WS 2011/12

Efficient Algorithms and Data Structures

Harald Räcke

Fakultät für Informatik
TU München

`http://www14.in.tum.de/lehre/2011WS/ea/`

Winter Term 2011/12

Part I

Organizational Matters

Part I

Organizational Matters

- Modul: IN2003
- Name: “Efficient Algorithms and Data Structures”
  “Effiziente Algorithmen und Datenstrukturen”
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 12:15–13:45 (Room 00.13.009A)
    Thu 10:15–11:45 (Room 00.04.011, HS2)
- Webpage: `http://www14.in.tum.de/lehre/2011WS/ea/`

Part I

Organizational Matters

- Modul: IN2003
- Name: "Efficient Algorithms and Data Structures"
  "Effiziente Algorithmen und Datenstrukturen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 12:15–13:45 (Room 00.13.009A)
    Thu 10:15–11:45 (Room 00.04.011, HS2)
- Webpage: `http://www14.in.tum.de/lehre/2011WS/ea/`

Part I

Organizational Matters

- Modul: IN2003
- Name: “Efficient Algorithms and Data Structures”
  “Effiziente Algorithmen und Datenstrukturen”
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 12:15–13:45 (Room 00.13.009A)
    Thu 10:15–11:45 (Room 00.04.011, HS2)
- Webpage: `http://www14.in.tum.de/lehre/2011WS/ea/`

Part I

Organizational Matters

- Modul: IN2003
- Name: "Efficient Algorithms and Data Structures"
  "Effiziente Algorithmen und Datenstrukturen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 12:15–13:45 (Room 00.13.009A)
    Thu 10:15–11:45 (Room 00.04.011, HS2)
- Webpage: `http://www14.in.tum.de/lehre/2011WS/ea/`

Part I

Organizational Matters

- Modul: IN2003
- Name: "Efficient Algorithms and Data Structures"
  "Effiziente Algorithmen und Datenstrukturen"
- ECTS: 8 Credit points
- Lectures:
  - 4 SWS
    Mon 12:15–13:45 (Room 00.13.009A)
    Thu 10:15–11:45 (Room 00.04.011, HS2)
- Webpage: `http://www14.in.tum.de/lehre/2011WS/ea/`

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

- Required knowledge:
  - IN0001, IN0003
    “Introduction to Informatics 1/2”
    “EinfÃijhrung in die Informatik 1/2”
  - IN0007
    “Fundamentals of Algorithms and Data Structures”
    “Grundlagen: Algorithmen und Datenstrukturen” (GAD)
  - IN0011
    “Basic Theoretic Informatics”
    “EinfÃijhrung in die Theoretische Informatik” (THEO)
  - IN0015
    “Discrete Structures”
    “Diskrete Strukturen” (DS)
  - IN0018
    “Discrete Probability Theory”
    “Diskrete Wahrscheinlichkeitstheorie” (DWT)

The Lecturer

- Harald RÃd’cke
- Email: raecke@in.tum.de
- Room: 03.09.044
- Office hours: (per appointment)

Tutorials

- Tutor:
    - Chintan Shah
    - chintan.shah@tum.de
    - Room: 03.09.059
    - Office hours: Wed 11:30–12:30
- Room: 00.08.038
- Time: Tue 14:14–15:45

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- In order to pass the module you need to
  1. pass an exam, and
  2. obtain at least 40% of the points in the assignment sheets.

- Exam:
  - Date will be announced shortly.
  - There are no resources allowed, apart from a hand-written piece of paper (A4).
  - Answers should be given in English, but German is also accepted.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Wednesday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Thursday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
  - Solutions have to be given in English.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.
  - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
    - An assignment sheet is usually made available on Wednesday on the module webpage.
    - Solutions have to be handed in in the following week before the lecture on Thursday.
    - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
    - Solutions have to be given in English.
    - Solutions will be discussed in the subsequent tutorial on Tuesday.
    - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
    - An assignment sheet is usually made available on Wednesday on the module webpage.
    - Solutions have to be handed in in the following week before the lecture on Thursday.
    - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
    - Solutions have to be given in English.
    - Solutions will be discussed in the subsequent tutorial on Tuesday.
    - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Wednesday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Thursday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
  - Solutions have to be given in English.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.
  - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
    - An assignment sheet is usually made available on Wednesday on the module webpage.
    - Solutions have to be handed in in the following week before the lecture on Thursday.
    - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
    - Solutions have to be given in English.
    - Solutions will be discussed in the subsequent tutorial on Tuesday.
    - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
  - An assignment sheet is usually made available on Wednesday on the module webpage.
  - Solutions have to be handed in in the following week before the lecture on Thursday.
  - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
  - Solutions have to be given in English.
  - Solutions will be discussed in the subsequent tutorial on Tuesday.
  - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

Assessment

- Assignment Sheets:
    - An assignment sheet is usually made available on Wednesday on the module webpage.
    - Solutions have to be handed in in the following week before the lecture on Thursday.
    - You can hand in your solutions by putting them in the right folder in front of room 03.09.052.
    - Solutions have to be given in English.
    - Solutions will be discussed in the subsequent tutorial on Tuesday.
    - We will probably have 12 assignment sheets. The first one will be out on Wednesday, 26 October.

1 Contents

- Foundations
  - Machine models
  - Efficiency measures
  - Asymptotic notation
  - Recursion
- Higher Data Structures
  - Search trees
  - Hashing
  - Priority queues
  - Union/Find data structures
- Cuts/Flows
- Matchings

1 Contents

- Foundations
    - Machine models
    - Efficiency measures
    - Asymptotic notation
    - Recursion
- Higher Data Structures
    - Search trees
    - Hashing
    - Priority queues
    - Union/Find data structures
- Cuts/Flows
- Matchings

1 Contents

- Foundations
  - Machine models
  - Efficiency measures
  - Asymptotic notation
  - Recursion
- Higher Data Structures
  - Search trees
  - Hashing
  - Priority queues
  - Union/Find data structures
- Cuts/Flows
- Matchings

1 Contents

- Foundations
    - Machine models
    - Efficiency measures
    - Asymptotic notation
    - Recursion
- Higher Data Structures
    - Search trees
    - Hashing
    - Priority queues
    - Union/Find data structures
- Cuts/Flows
- Matchings

2 Literatur I

- Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman:
  The design and analysis of computer algorithms,
  Addison-Wesley Publishing Company: Reading (MA), 1974
- Thomas H. Cormen, Charles E. Leiserson, Ron L. Rivest, Clifford Stein:
  Introduction to algorithms,
  McGraw-Hill, 1990
- Michael T. Goodrich, Roberto Tamassia:
  Algorithm design: Foundations, analysis, and internet examples,
  John Wiley & Sons, 2002

2 Literatur II

- Volker Heun:
  Grundlegende Algorithmen: EinfÃijhrung in den Entwurf und die Analyse effizienter Algorithmen,
  2. Auflage, Vieweg, 2003
- Jon Kleinberg, Eva Tardos:
  Algorithm Design,
  Addison-Wesley, 2005
- Donald E. Knuth:
  The art of computer programming. Vol. 1: Fundamental Algorithms,
  3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997

2 Literatur III

- Donald E. Knuth:
  The art of computer programming. Vol. 3: Sorting and Searching,
  3. Auflage, Addison-Wesley Publishing Company: Reading (MA), 1997
- Christos H. Papadimitriou, Kenneth Steiglitz:
  Combinatorial Optimization: Algorithms and Complexity,
  Prentice Hall, 1982
- Uwe SchÃČÂűning:
  Algorithmik,
  Spektrum Akademischer Verlag, 2001

2 Literatur IV

Steven S. Skiena:
The Algorithm Design Manual,
Springer, 1998

Part II

Foundations

3 Goals

- Gain knowledge about efficient algorithms for important problems, i.e., learn how to solve certain types of problems efficiently.
- Learn how to analyze and judge the efficiency of algorithms.
- Learn how to design efficient algorithms.

3 Goals

- Gain knowledge about efficient algorithms for important problems, i.e., learn how to solve certain types of problems efficiently.
- Learn how to analyze and judge the efficiency of algorithms.
- Learn how to design efficient algorithms.

3 Goals

- Gain knowledge about efficient algorithms for important problems, i.e., learn how to solve certain types of problems efficiently.
- Learn how to analyze and judge the efficiency of algorithms.
- Learn how to design efficient algorithms.

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- . . .

4 Modelling Issues

What do you measure?

- Memory requirement
- Running time
- Number of comparisons
- Number of multiplications
- Number of hard-disc accesses
- Program size
- Power consumption
- ...

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.
- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like "this algorithm always runs in time $\mathcal{O}(n^2)$".
  - Typically focuses on the worst case.
  - Can give lower bounds like "any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case".

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
    - How do you choose your inputs?
    - May be very time-consuming.
    - Very reliable results if done correctly.
    - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
    - Gives asymptotic bounds like "this algorithm always runs in time $O(n^2)$".
    - Typically focuses on the worst case.
    - Can give lower bounds like "any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case".

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like "this algorithm always runs in time $\mathcal{O}(n^2)$".
  - Typically focuses on the worst case.
  - Can give lower bounds like "any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case".

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like "this algorithm always runs in time $O(n^2)$".
  - Typically focuses on the worst case.
  - Can give lower bounds like "any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case".

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like "this algorithm always runs in time $O(n^2)$".
  - Typically focuses on the worst case.
  - Can give lower bounds like "any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case".

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like “this algorithm always runs in time $O(n^2)$”.
  - Typically focuses on the worst case.
  - Can give lower bounds like “any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case”.

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like “this algorithm always runs in time $O(n^2)$”.
  - Typically focuses on the worst case.
  - Can give lower bounds like “any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case”.

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like “this algorithm always runs in time $O(n^2)$”.
  - Typically focuses on the worst case.
  - Can give lower bounds like “any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case”.

4 Modelling Issues

How do you measure?

- Implementing and testing on representative inputs
  - How do you choose your inputs?
  - May be very time-consuming.
  - Very reliable results if done correctly.
  - Results only hold for a specific machine and for a specific set of inputs.

- Theoretical analysis in a specific model of computation.
  - Gives asymptotic bounds like “this algorithm always runs in time $O(n^2)$”.
  - Typically focuses on the worst case.
  - Can give lower bounds like “any comparison-based sorting algorithm needs at least $\Omega(n \log n)$ comparisons in the worst case”.

4 Modelling Issues

Input length

The theoretical bounds are usually given by a function $f : \mathbb{N} \to \mathbb{N}$ that maps the input length to the running time (or storage space, comparisons, multiplications, program size etc.).

The input length may e.g. be

- the size of the input (number of bits)
- the number of arguments

Suppose n numbers from the interval $\{1, \dots, N\}$ have to be sorted. In this case we usually say that the input length is n instead of e.g. $n \log N$, which would be the number of bits required to encode the input.

4 Modelling Issues

Input length

The theoretical bounds are usually given by a function $f : \mathbb{N} \to \mathbb{N}$ that maps the input length to the running time (or storage space, comparisons, multiplications, program size etc.).

The input length may e.g. be

- the size of the input (number of bits)
- the number of arguments

Suppose n numbers from the interval $\{1, \dots, N\}$ have to be sorted. In this case we usually say that the input length is n instead of e.g. $n \log N$, which would be the number of bits required to encode the input.

4 Modelling Issues

Input length

The theoretical bounds are usually given by a function $f : \mathbb{N} \to \mathbb{N}$ that maps the input length to the running time (or storage space, comparisons, multiplications, program size etc.).

The input length may e.g. be

- the size of the input (number of bits)
- the number of arguments

Example 1

Suppose n numbers from the interval $\{1, \dots, N\}$ have to be sorted. In this case we usually say that the input length is n instead of e.g. $n \log N$, which would be the number of bits required to encode the input.

4 Modelling Issues

Input length

The theoretical bounds are usually given by a function $f : \mathbb{N} \to \mathbb{N}$ that maps the input length to the running time (or storage space, comparisons, multiplications, program size etc.).

The input length may e.g. be

- the size of the input (number of bits)
- the number of arguments

Example 1

Suppose n numbers from the interval $\{1, \dots, N\}$ have to be sorted. In this case we usually say that the input length is n instead of e.g. $n \log N$, which would be the number of bits required to encode the input.

4 Modelling Issues

Input length

The theoretical bounds are usually given by a function $f : \mathbb{N} \to \mathbb{N}$ that maps the input length to the running time (or storage space, comparisons, multiplications, program size etc.).

The input length may e.g. be

- the size of the input (number of bits)
- the number of arguments

Example 1

Suppose n numbers from the interval $\{1, \ldots, N\}$ have to be sorted. In this case we usually say that the input length is n instead of e.g. $n \log N$, which would be the number of bits required to encode the input.

Model of Computation

How to measure performance

1. Calculate running time and storage space etc. on a simplified, idealized model of computation, e.g. Random Access Machine (RAM), Turing Machine (TM), ...
2. Calculate number of certain basic operations: comparisons, multiplications, harddisc accesses, ...

Version 2. is often easier, but focusing on one type of operation makes it more difficult to obtain meaningful results.

Model of Computation

How to measure performance

1. Calculate running time and storage space etc. on a simplified, idealized model of computation, e.g. Random Access Machine (RAM), Turing Machine (TM), . . .
2. Calculate number of certain basic operations: comparisons, multiplications, harddisc accesses, . . .

Version 2. is often easier, but focusing on one type of operation makes it more difficult to obtain meaningful results.

Model of Computation

How to measure performance

1. Calculate running time and storage space etc. on a simplified, idealized model of computation, e.g. Random Access Machine (RAM), Turing Machine (TM), ...
2. Calculate number of certain basic operations: comparisons, multiplications, harddisc accesses, ...

Version 2. is often easier, but focusing on one type of operation makes it more difficult to obtain meaningful results.

Model of Computation

How to measure performance

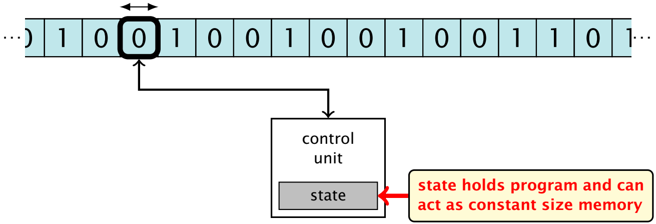
1. Calculate running time and storage space etc. on a simplified, idealized model of computation, e.g. Random Access Machine (RAM), Turing Machine (TM), ...
2. Calculate number of certain basic operations: comparisons, multiplications, harddisc accesses, ...

Version 2. is often easier, but focusing on one type of operation makes it more difficult to obtain meaningful results.

Turing Machine

- Very simple model of computation.
- Only the “current” memory location can be altered.
- Very good model for discussing computabiliy, or polynomial vs. exponential time.
- Some simple problems like recognizing whether input is of the form xx, where x is a string, have quadratic lower bound.

$\Rightarrow$ **Not a good model for developing efficient algorithms.**

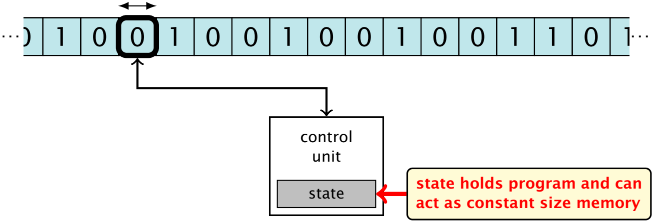


state holds program and can act as constant size memory

Turing Machine

- Very simple model of computation.
- Only the "current" memory location can be altered.
- Very good model for discussing computabiliy, or polynomial vs. exponential time.
- Some simple problems like recognizing whether input is of the form xx, where x is a string, have quadratic lower bound.

⇒ Not a good model for developing efficient algorithms.

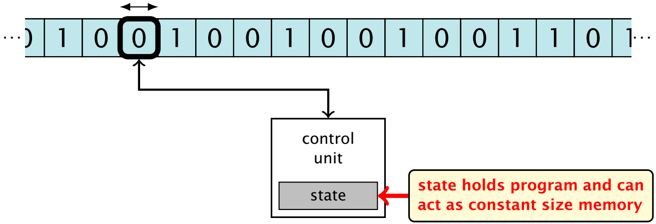


state holds program and can act as constant size memory

Turing Machine

- Very simple model of computation.
- Only the “current” memory location can be altered.
- Very good model for discussing computabiliy, or polynomial vs. exponential time.
- Some simple problems like recognizing whether input is of the form xx, where x is a string, have quadratic lower bound.

$\Longrightarrow$ **Not a good model for developing efficient algorithms.**

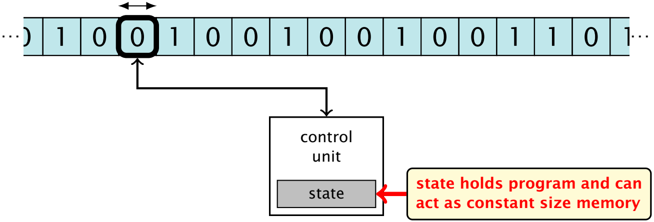


state holds program and can act as constant size memory

Turing Machine

- Very simple model of computation.
- Only the “current” memory location can be altered.
- Very good model for discussing computabiliy, or polynomial vs. exponential time.
- Some simple problems like recognizing whether input is of the form xx, where x is a string, have quadratic lower bound.

$\Longrightarrow$ Not a good model for developing efficient algorithms.

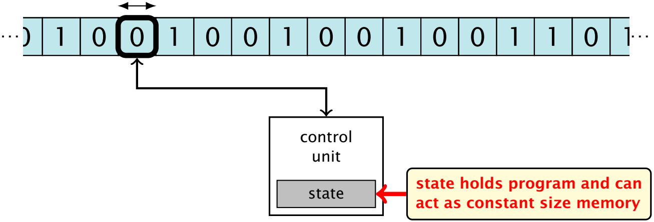


state holds program and can act as constant size memory

Turing Machine

- Very simple model of computation.
- Only the “current” memory location can be altered.
- Very good model for discussing computabiliy, or polynomial vs. exponential time.
- Some simple problems like recognizing whether input is of the form xx, where x is a string, have quadratic lower bound.

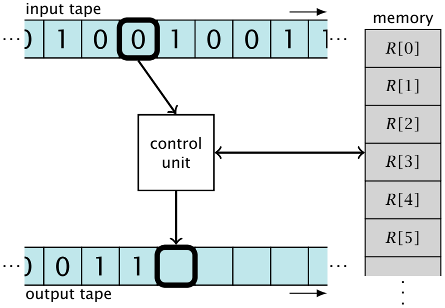
⇒ **Not a good model for developing efficient algorithms.**



state holds program and can act as constant size memory

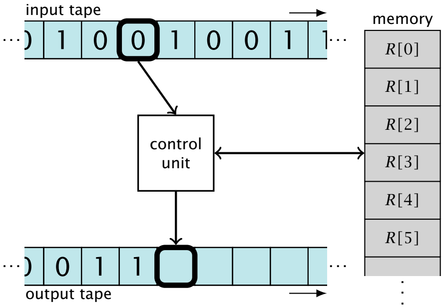
Random Access Machine (RAM)

- Input tape and output tape (sequences of zeros and ones; unbounded length).
- Memory unit: infinite but countable number of registers $R[0], R[1], R[2], \ldots$.
- Registers hold integers.
- Indirect addressing.



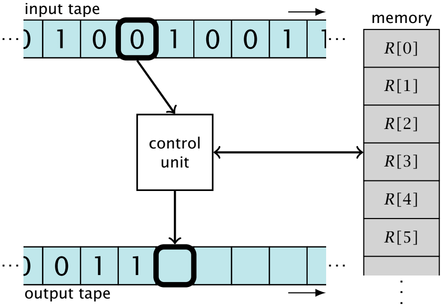
Random Access Machine (RAM)

- Input tape and output tape (sequences of zeros and ones; unbounded length).
- Memory unit: infinite but countable number of registers $R[0], R[1], R[2], \ldots$.
- Registers hold integers.
- Indirect addressing.



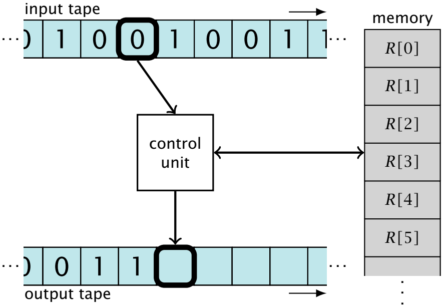
Random Access Machine (RAM)

- Input tape and output tape (sequences of zeros and ones; unbounded length).
- Memory unit: infinite but countable number of registers $R[0], R[1], R[2], \ldots$.
- Registers hold integers.
- Indirect addressing.



Random Access Machine (RAM)

- Input tape and output tape (sequences of zeros and ones; unbounded length).
- Memory unit: infinite but countable number of registers $R[0], R[1], R[2], \ldots$.
- Registers hold integers.
- Indirect addressing.



Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
    - READ i
- output operations ($R[i]$ → output tape)
    - WRITE i
- register-register transfers
    - $R[j] := R[k]$
    - $R[j] := 4$
- **indirect** addressing
    - $R[j] := R[R[i]]$
      loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - READ i
- output operations ($R[i]$ → output tape)
  - WRITE i
- register-register transfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirect** addressing
  - $R[j] := R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - READ i
- output operations ($R[i]$ → output tape)
  - WRITE i
- register-register transfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirect** addressing
  - $R[j] := R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
    - `READ` i
- output operations ($R[i]$ → output tape)
    - `WRITE` i
- register-register transfers
    - $R[j]$:= $R[i]$
    - $R[j]$:= 4
- **indirect** addressing
    - $R[j]$:= $R[R[i]]$
      loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - READ i
- output operations ($R[i]$ → output tape)
  - WRITE i
- register-register transfers
  - $R[j] := R[i]$
  - $R[j] := 4$
- **indirect** addressing
  - $R[j] := R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - READ i
- output operations ($R[i]$ → output tape)
  - WRITE i
- register-register transfers
  - $R[j]$:= $R[i]$
  - $R[j]$:= 4
- **indirect** addressing
  - $R[j]$:= $R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
    - `READ` i
- output operations ($R[i]$ → output tape)
    - `WRITE` i
- register-register transfers
    - $R[j]$ `:=` $R[i]$
    - $R[j]$ `:=` 4
- **indirect** addressing
    - $R[j]$ `:=` $R[R[i]]$
      loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - READ i
- output operations ($R[i]$ → output tape)
  - WRITE i
- register-register transfers
  - $R[j]$:= $R[i]$
  - $R[j]$:= 4
- **indirect** addressing
  - $R[j]$:= $R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- input operations (input tape → $R[i]$)
  - `READ` i
- output operations ($R[i]$ → output tape)
  - `WRITE` i
- register-register transfers
  - $R[j]$ `:=` $R[i]$
  - $R[j]$ `:=` 4
- **indirect** addressing
  - $R[j]$ `:=` $R[R[i]]$
    loads the content of the register number $R[i]$ into register number j

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+$, $-$, $\times$, $/$
  - $R[i] := R[j] + R[k]$;
    $R[i] := -R[k]$;

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+$, $-$, $\times$, $/$
  - `R[i] := R[j] + R[k];`
    `R[i] := -R[k];`

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+, -, \times, /$
  - $R[i] := R[j] + R[k]$;
    $R[i] := -R[k]$;

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+, -, \times, /$
  - $R[i] := R[j] + R[k];$
    $R[i] := -R[k];$

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+$, $-$, $\times$, $/$
  - $R[i]$:= $R[j]$ + $R[k]$;
    $R[i]$:= -$R[k]$;

Random Access Machine (RAM)

Operations

- branching (including loops) based on comparisons
  - `jump` x
    jumps to position x in the program;
    sets instruction counter to x;
    reads the next operation to perform from register $R[x]$
  - `jumpz` x $R[i]$
    jump to x if $R[i] = 0$
    if not the instruction counter is increased by 1;
  - `jumpi` i
    jump to $R[i]$ (indirect jump);
- arithmetic instructions: $+$, $-$, $\times$, $/$
  - $R[i]$ `:=` $R[j]$ `+` $R[k]$`;`
    $R[i]$ `:=` `-`$R[k]$`;`

Model of Computation

- uniform cost model
  Every operation takes time 1.
- logarithmic cost model
  The cost depends on the content of memory cells:
  - The time for a step is equal to the largest operand involved;
  - The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

Bounded word RAM model: cost is uniform but the largest value stored in a register may not exceed w, where usually $w = \log_2 n$.

Model of Computation

- uniform cost model
  Every operation takes time 1.
- logarithmic cost model
  The cost depends on the content of memory cells:
  - The time for a step is equal to the largest operand involved;
  - The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

Bounded word RAM model: cost is uniform but the largest value stored in a register may not exceed w, where usually $w = \log_2 n$.

Model of Computation

- uniform cost model
  Every operation takes time 1.
- logarithmic cost model
  The cost depends on the content of memory cells:
  - The time for a step is equal to the largest operand involved;
  - The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

Bounded word RAM model: cost is uniform but the largest value stored in a register may not exceed w, where usually $w = \log_2 n$.

Model of Computation

- uniform cost model

  Every operation takes time 1.

- logarithmic cost model

  The cost depends on the content of memory cells:

  - The time for a step is equal to the largest operand involved;
  - The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

Bounded word RAM model: cost is uniform but the largest value stored in a register may not exceed w, where usually $w = \log_2 n$.

Model of Computation

- uniform cost model

  Every operation takes time 1.
- logarithmic cost model

  The cost depends on the content of memory cells:
  - The time for a step is equal to the largest operand involved;
  - The storage space of a register is equal to the length (in bits) of the largest value ever stored in it.

Bounded word RAM model: cost is uniform but the largest value stored in a register may not exceed w, where usually $w = \log_2 n$.

4 Modelling Issues

Example 2

```
Algorithm 1 RepeatedSquaring($n$)
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

running time

space requirement

4 Modelling Issues

Example 2

Algorithm 1 RepeatedSquaring(n)

```
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

4 Modelling Issues

Example 2

Algorithm 1 RepeatedSquaring(n)

```
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

4 Modelling Issues

Example 2

```
**Algorithm 1** RepeatedSquaring($n$)
$r \leftarrow 2$;
**for** $i = 1 \to n$ **do**
    $r \leftarrow r^2$
**return** $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

4 Modelling Issues

Example 2

```
Algorithm 1 RepeatedSquaring($n$)
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

4 Modelling Issues

Example 2

```
Algorithm 1 RepeatedSquaring($n$)
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

4 Modelling Issues

Example 2

```
Algorithm 1 RepeatedSquaring($n$)
$r \leftarrow 2$;
for $i = 1 \rightarrow n$ do
    $r \leftarrow r^2$
return $r$
```

- running time:
  - uniform model: n steps
  - logarithmic model: $1 + 2 + 4 + \cdots + 2^n = 2^{n+1} - 1 = \Theta(2^n)$
- space requirement:
  - uniform model: $\mathcal{O}(1)$
  - logarithmic model: $\mathcal{O}(2^n)$

There are different types of complexity bounds:

- best-case complexity:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Usually easy to analyze, but not very meaningful.

- worst-case complexity:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Usually moderately easy to analyze; sometimes too pessimistic.

- average case complexity:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  more general: probability measure μ

  $$C_{\mathrm{avg}}(n) := \sum_{x \in I_n} \mu(x) \cdot C(x)$$

There are different types of complexity bounds:

- best-case complexity:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Usually easy to analyze, but not very meaningful.
- worst-case complexity:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Usually moderately easy to analyze; sometimes too pessimistic.
- average case complexity:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  more general: probability measure μ

  $$C_{\mathrm{avg}}(n) := \sum_{x \in I_n} \mu(x) \cdot C(x)$$

There are different types of complexity bounds:

- best-case complexity:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Usually easy to analyze, but not very meaningful.
- worst-case complexity:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Usually moderately easy to analyze; sometimes too pessimistic.
- average case complexity:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  more general: probability measure μ

  $$C_{\mathrm{avg}}(n) := \sum_{x \in I_n} \mu(x) \cdot C(x)$$

There are different types of complexity bounds:

- best-case complexity:

  $$C_{\mathrm{bc}}(n) := \min\{C(x) \mid |x| = n\}$$

  Usually easy to analyze, but not very meaningful.
- worst-case complexity:

  $$C_{\mathrm{wc}}(n) := \max\{C(x) \mid |x| = n\}$$

  Usually moderately easy to analyze; sometimes too pessimistic.
- average case complexity:

  $$C_{\mathrm{avg}}(n) := \frac{1}{|I_n|} \sum_{|x|=n} C(x)$$

  more general: probability measure μ

  $$C_{\mathrm{avg}}(n) := \sum_{x \in I_n} \mu(x) \cdot C(x)$$

There are different types of complexity bounds:

- amortized complexity:
  The average cost of data structure operations over a worst case sequence of operations.
- randomized complexity:
  The algorithm may use random bits. Expected running time (over all possible choices of random bits) for a fixed input x. Then take the worst-case over all x with $|x| = n$.

There are different types of complexity bounds:

- amortized complexity:
  The average cost of data structure operations over a worst case sequence of operations.
- randomized complexity:
  The algorithm may use random bits. Expected running time (over all possible choices of random bits) for a fixed input x. Then take the worst-case over all x with $|x| = n$.

5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymtotic classification of the running time, that ignores constant factors and constant additive offsets.

- We are usually interested in the running times for large values of n. Then constant additive terms do not play an important role.
- An exact analysis (e.g. exactly counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- Running time should be expressed by simple functions.

5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymtotic classification of the running time, that ignores constant factors and constant additive offsets.

- We are usually interested in the running times for large values of n. Then constant additive terms do not play an important role.
- An exact analysis (e.g. *exactly* counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- Running time should be expressed by simple functions.

5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymtotic classification of the running time, that ignores constant factors and constant additive offsets.

- We are usually interested in the running times for large values of n. Then constant additive terms do not play an important role.
- An exact analysis (e.g. *exactly* counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- Running time should be expressed by simple functions.

5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymtotic classification of the running time, that ignores constant factors and constant additive offsets.

- We are usually interested in the running times for large values of n. Then constant additive terms do not play an important role.
- An exact analysis (e.g. *exactly* counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- Running time should be expressed by simple functions.

5 Asymptotic Notation

We are usually not interested in exact running times, but only in an asymtotic classification of the running time, that ignores constant factors and constant additive offsets.

- We are usually interested in the running times for large values of n. Then constant additive terms do not play an important role.
- An exact analysis (e.g. *exactly* counting the number of operations in a RAM) may be hard, but wouldn't lead to more precise results as the computational model is already quite a distance from reality.
- A linear speed-up (i.e., by a constant factor) is always possible by e.g. implementing the algorithm on a faster machine.
- Running time should be expressed by simple functions.

Asymptotic Notation

Formal Definition

Let f denote functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not faster than f)

Asymptotic Notation

Formal Definition

Let f denote functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not faster than f)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not slower than f)

Asymptotic Notation

Formal Definition

Let f denote functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not faster than f)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not slower than f)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (functions that asymptotically have the same growth as f)

Asymptotic Notation

Formal Definition

Let f denote functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not faster than f)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not slower than f)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (functions that asymptotically have the same growth as f)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow slower than f)

Asymptotic Notation

Formal Definition

Let f denote functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $\mathcal{O}(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not faster than f)
- $\Omega(f) = \{g \mid \exists c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (set of functions that asymptotically grow not slower than f)
- $\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$
  (functions that asymptotically have the same growth as f)
- $o(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \leq c \cdot f(n)]\}$
  (set of functions that asymptotically grow slower than f)
- $\omega(f) = \{g \mid \forall c > 0\ \exists n_0 \in \mathbb{N}_0\ \forall n \geq n_0 : [g(n) \geq c \cdot f(n)]\}$
  (set of functions that asymptotically grow faster than f)

Asymptotic Notation

There is an equivalent definition using limes notation. f and g are functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotic Notation

There is an equivalent definition using limes notation. f and g are functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$

Asymptotic Notation

There is an equivalent definition using limes notation. f and g are functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \le \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \le \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$

Asymptotic Notation

There is an equivalent definition using limes notation. f and g are functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim\limits_{n\to\infty} \dfrac{g(n)}{f(n)} = 0$

Asymptotic Notation

There is an equivalent definition using limes notation. f and g are functions from $\mathbb{N}$ to $\mathbb{R}^+$.

- $g \in \mathcal{O}(f)$: $\quad 0 \leq \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in \Omega(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} \leq \infty$
- $g \in \Theta(f)$: $\quad 0 < \lim_{n\to\infty} \frac{g(n)}{f(n)} < \infty$
- $g \in o(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
- $g \in \omega(f)$: $\quad \lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$

Asymptotic Notation

Abuse of notation

1. People write $f = \mathcal{O}(g)$, when they mean $f \in \mathcal{O}(g)$. This is **not** an equality (how could a function be equal to a set of functions).
2. People write $f(n) = \mathcal{O}(g(n))$, when they mean $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. People write e.g. $h(n) = f(n) + o(g(n))$ when they mean that there exists a function $z : \mathbb{N} \to \mathbb{R}^+, n \mapsto z(n), z \in o(g)$ such that $h(n) \le f(n) + z(n)$.
4. People write $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, when they mean $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Again this is not an equality.

Asymptotic Notation

Abuse of notation

1. People write $f = \mathcal{O}(g)$, when they mean $f \in \mathcal{O}(g)$. This is **not** an equality (how could a function be equal to a set of functions).
2. People write $f(n) = \mathcal{O}(g(n))$, when they mean $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. People write e.g. $h(n) = f(n) + o(g(n))$ when they mean that there exists a function $z : \mathbb{N} \to \mathbb{R}^+, n \mapsto z(n), z \in o(g)$ such that $h(n) \leq f(n) + z(n)$.
4. People write $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, when they mean $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Again this is not an equality.

Asymptotic Notation

Abuse of notation

1. People write $f = \mathcal{O}(g)$, when they mean $f \in \mathcal{O}(g)$. This is **not** an equality (how could a function be equal to a set of functions).
2. People write $f(n) = \mathcal{O}(g(n))$, when they mean $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. People write e.g. $h(n) = f(n) + o(g(n))$ when they mean that there exists a function $z : \mathbb{N} \to \mathbb{R}^+, n \mapsto z(n), z \in o(g)$ such that $h(n) \leq f(n) + z(n)$.
4. People write $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, when they mean $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Again this is not an equality.

Asymptotic Notation

Abuse of notation

1. People write $f = \mathcal{O}(g)$, when they mean $f \in \mathcal{O}(g)$. This is **not** an equality (how could a function be equal to a set of functions).
2. People write $f(n) = \mathcal{O}(g(n))$, when they mean $f \in \mathcal{O}(g)$, with $f : \mathbb{N} \to \mathbb{R}^+, n \mapsto f(n)$, and $g : \mathbb{N} \to \mathbb{R}^+, n \mapsto g(n)$.
3. People write e.g. $h(n) = f(n) + o(g(n))$ when they mean that there exists a function $z : \mathbb{N} \to \mathbb{R}^+, n \mapsto z(n), z \in o(g)$ such that $h(n) \leq f(n) + z(n)$.
4. People write $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$, when they mean $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Again this is not an equality.

Asymptotic Notation

Lemma 3

Let f, g be functions with the property $\exists n_0 > 0\ \forall n \geq n_0 : f(n) > 0$ (the same for g). Then

- $c \cdot f(n) \in \Theta(f(n))$ *for any constant c*
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

The expressions also hold for Ω. Note that this means that $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$.

Asymptotic Notation

Lemma 3

Let f, g be functions with the property $\exists n_0 > 0\ \forall n \geq n_0 : f(n) > 0$ (the same for g). Then

- $c \cdot f(n) \in \Theta(f(n))$ *for any constant c*
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

The expressions also hold for Ω. Note that this means that $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$.

Asymptotic Notation

Lemma 3

Let f, g be functions with the property
$\exists n_0 > 0\ \forall n \geq n_0 : f(n) > 0$ *(the same for g). Then*

- $c \cdot f(n) \in \Theta(f(n))$ *for any constant c*
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

The expressions also hold for Ω. Note that this means that
$f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$.

Asymptotic Notation

Lemma 3

Let f, g be functions with the property $\exists n_0 > 0\ \forall n \geq n_0 : f(n) > 0$ (the same for g). Then

- $c \cdot f(n) \in \Theta(f(n))$ *for any constant c*
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

The expressions also hold for Ω. Note that this means that $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$.

Asymptotic Notation

Lemma 3

Let f, g *be functions with the property* $\exists n_0 > 0\ \forall n \geq n_0 : f(n) > 0$ *(the same for* g*). Then*

- $c \cdot f(n) \in \Theta(f(n))$ *for any constant* c
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(f(n) + g(n))$
- $\mathcal{O}(f(n)) \cdot \mathcal{O}(g(n)) = \mathcal{O}(f(n) \cdot g(n))$
- $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

The expressions also hold for Ω*. Note that this means that* $f(n) + g(n) \in \Theta(\max\{f(n), g(n)\})$*.*

Asymptotic Notation

Comments

- Do not use asymptotic notation within induction proofs.
- For any constants a, b we have $\log_a n = \Theta(\log_b n)$. Therefore, we will usually ignore the base of a logarithm within asymptotic notation.
- In general $\log n = \log_2 n$, i.e., we use 2 as the default base for the logarithm.

Asymptotic Notation

Comments

- Do not use asymptotic notation within induction proofs.
- For any constants a, b we have $\log_a n = \Theta(\log_b n)$. Therefore, we will usually ignore the base of a logarithm within asymptotic notation.
- In general $\log n = \log_2 n$, i.e., we use 2 as the default base for the logarithm.

Asymptotic Notation

Comments

- Do not use asymptotic notation within induction proofs.
- For any constants a, b we have $\log_a n = \Theta(\log_b n)$. Therefore, we will usually ignore the base of a logarithm within asymptotic notation.
- In general $\log n = \log_2 n$, i.e., we use 2 as the default base for the logarithm.

6 Recurrences

Algorithm 2 mergesort(list L)

```
$s \leftarrow \text{size}(L)$
**if** $s \leq 1$ **return** $L$
$L_1 \leftarrow L[1 \cdots \lfloor \frac{s}{2} \rfloor]$
$L_2 \leftarrow L[\lceil \frac{s}{2} \rceil \cdots n]$
$\text{mergesort}(L_1)$
$\text{mergesort}(L_2)$
$L \leftarrow \text{merge}(L_1, L_2)$
**return** $L$
```

This algorithm requires

$$T(n) \leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n)$$

comparisons when $n > 1$ and 0 comparisons when $n \leq 1$.

6 Recurrences

```
Algorithm 2 mergesort(list $L$)
$s \leftarrow \mathrm{size}(L)$
if $s \le 1$ return $L$
$L_1 \leftarrow L[1 \cdots \lfloor \frac{s}{2} \rfloor]$
$L_2 \leftarrow L[\lceil \frac{s}{2} \rceil \cdots n]$
$\mathrm{mergesort}(L_1)$
$\mathrm{mergesort}(L_2)$
$L \leftarrow \mathrm{merge}(L_1, L_2)$
return $L$
```

This algorithm requires

$$T(n) \le 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \mathcal{O}(n)$$

comparisons when $n > 1$ and 0 comparisons when $n \le 1$.

Recurrences

How do we bring the expression for the number of comparisons ($\approx$ running time) into a closed form?

For this we need to solve the recurrence.

Recurrences

How do we bring the expression for the number of comparisons ($\approx$ running time) into a closed form?

For this we need to solve the recurrence.

Methods for Solving Recurrences

1. **Guessing+Induction**
   Guess the right solution and prove that it is correct via induction. It needs experience to make the right guess.
2. **Master Theorem**
   For a lot of recurrences that appear in the analysis of algorithms this theorem can be used to obtain tight asymptotic bounds. It does not provide exact solutions.
3. **Characteristic Polynomial**
   Linear homogenous recurrences can be solved via this method.

6.1 Guessing+Induction

First we need to get rid of the $\mathcal{O}$-notation in our recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Assume that instead we had

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

One way of solving such a recurrence is to guess a solution, and check that it is correct by plugging it in.

6.1 Guessing+Induction

First we need to get rid of the $\mathcal{O}$-notation in our recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Assume that instead we had

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

One way of solving such a recurrence is to guess a solution, and check that it is correct by plugging it in.

6.1 Guessing+Induction

First we need to get rid of the $\mathcal{O}$-notation in our recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Assume that instead we had

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

One way of solving such a recurrence is to guess a solution, and check that it is correct by plugging it in.

6.1 Guessing+Induction

Suppose we guess $T(n) \leq dn\log n$ for a constant d.

6.1 Guessing+Induction

Suppose we guess $T(n) \leq dn\log n$ for a constant d. Then

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

6.1 Guessing+Induction

Suppose we guess $T(n) \leq dn \log n$ for a constant d. Then

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \end{aligned}$$

6.1 Guessing+Induction

Suppose we guess $T(n) \leq dn\log n$ for a constant d. Then

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \end{aligned}$$

6.1 Guessing+Induction

Suppose we guess $T(n) \le dn \log n$ for a constant d. Then

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn \log n + (c-d)n \end{aligned}$$

6.1 Guessing+Induction

Suppose we guess $T(n) \leq dn \log n$ for a constant d. Then

$$\begin{aligned}
T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\
&\leq 2\left(\frac{n}{2} \log \frac{n}{2}\right) + cn \\
&= dn(\log n - 1) + cn \\
&= dn \log n + (c - d)n \\
&= dn \log n
\end{aligned}$$

if we choose $d \geq c$.

6.1 Guessing+Induction

Suppose we guess $T(n) \le dn \log n$ for a constant d. Then

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn\log n + (c - d)n \\ &= dn\log n \end{aligned}$$

if we choose $d \ge c$.

Formally one would make an induction proof, where the above is the induction step. The base case is usually trivial.

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn \log n$.

Proof. (by induction)

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn \log n$.

Proof. (by induction)

- **base case** $(2 \leq n < 16)$:

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

Proof. (by induction)

- **base case** $(2 \le n < 16)$: true if we choose $d \ge b$.

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

Proof. (by induction)

- **base case** $(2 \le n < 16)$: true if we choose $d \ge b$.
- **induction step** $2 \ldots n-1 \to n$:

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn \log n$.

Proof. (by induction)

- **base case** $(2 \leq n < 16)$: true if we choose $d \geq b$.
- **induction step** $2 \ldots n-1 \rightarrow n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \geq 16$.
  We prove it for n:

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn \log n$.

Proof. (by induction)

- **base case** $(2 \leq n < 16)$: true if we choose $d \geq b$.
- **induction step** $2 \ldots n-1 \rightarrow n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \geq 16$.
  We prove it for n:

$$T(n) \leq 2T\left(\frac{n}{2}\right) + cn$$

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

Proof. (by induction)

- **base case** $(2 \le n < 16)$: true if we choose $d \ge b$.
- **induction step** $2 \ldots n-1 \to n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \ge 16$.
  We prove it for n:

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{n}{2} \log \frac{n}{2}\right) + cn \end{aligned}$$

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn\log n$.

Proof. (by induction)

- **base case** $(2 \leq n < 16)$: true if we choose $d \geq b$.
- **induction step** $2 \dots n-1 \rightarrow n$:

  Suppose statem. is true for $n' \in \{2, \dots, n-1\}$, and $n \geq 16$.
  We prove it for n:

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \end{aligned}$$

6.1 Guessing+Induction

$$T(n) \leq \begin{cases} 2T(\frac{n}{2}) + cn & n \geq 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \leq dn \log n$.

Proof. (by induction)

- **base case** $(2 \leq n < 16)$: true if we choose $d \geq b$.
- **induction step** $2 \ldots n-1 \to n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \geq 16$.
  We prove it for n:

$$\begin{aligned} T(n) &\leq 2T\left(\frac{n}{2}\right) + cn \\ &\leq 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn\log n + (c-d)n \end{aligned}$$

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

Proof. (by induction)

- **base case** $(2 \le n < 16)$: true if we choose $d \ge b$.
- **induction step** $2 \ldots n-1 \to n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \ge 16$.
  We prove it for n:

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{n}{2} \log \frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn \log n + (c-d)n \\ &= dn \log n \end{aligned}$$

6.1 Guessing+Induction

$$T(n) \le \begin{cases} 2T(\frac{n}{2}) + cn & n \ge 16 \\ b & \text{otw.} \end{cases}$$

Guess: $T(n) \le dn \log n$.

Proof. (by induction)

- **base case** $(2 \le n < 16)$: true if we choose $d \ge b$.
- **induction step** $2 \ldots n-1 \to n$:

  Suppose statem. is true for $n' \in \{2, \ldots, n-1\}$, and $n \ge 16$.
  We prove it for n:

$$\begin{aligned} T(n) &\le 2T\left(\frac{n}{2}\right) + cn \\ &\le 2\left(\frac{n}{2}\log\frac{n}{2}\right) + cn \\ &= dn(\log n - 1) + cn \\ &= dn\log n + (c-d)n \\ &= dn\log n \end{aligned}$$

  Hence, statement is true if we choose $d \ge c$.

6.1 Guessing+Induction

Why did we change the recurrence by getting rid of the ceiling?

6.1 Guessing+Induction

Why did we change the recurrence by getting rid of the ceiling?

If we do not do this we instead consider the following recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 16 \\ b & \text{otherwise} \end{cases}$$

6.1 Guessing+Induction

Why did we change the recurrence by getting rid of the ceiling?

If we do not do this we instead consider the following recurrence:

$$T(n) \leq \begin{cases} 2T(\lceil \frac{n}{2} \rceil) + cn & n \geq 16 \\ b & \text{otherwise} \end{cases}$$

Note that we can do this as for constant-sized inputs the running time is always some constant (b in the above case).

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn \log n$ and get

$$T(n)$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn \log n$ and get

$$T(n) \le 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn$$

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn\log n$ and get

$$
\begin{aligned}
T(n) &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\
&\leq 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn
\end{aligned}
$$

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn\log n$ and get

$$\begin{aligned} T(n) &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\ &\leq 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \end{aligned}$$

$$\boxed{\left\lceil \frac{n}{2} \right\rceil \leq \frac{n}{2} + 1}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn \log n$ and get

$$\begin{aligned} T(n) &\le 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\ &\le 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \\ \boxed{\left\lceil \tfrac{n}{2} \right\rceil \le \tfrac{n}{2} + 1} \quad &\le 2\big(d(n/2+1)\log(n/2+1)\big) + cn \end{aligned}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn\log n$ and get

$$\begin{aligned} T(n) &\leq 2T\left(\left\lceil\frac{n}{2}\right\rceil\right) + cn \\ &\leq 2\left(d\left\lceil\frac{n}{2}\right\rceil\log\left\lceil\frac{n}{2}\right\rceil\right) + cn \\ \boxed{\left\lceil\tfrac{n}{2}\right\rceil \leq \tfrac{n}{2}+1} \quad &\leq 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\ \boxed{\tfrac{n}{2}+1 \leq \tfrac{9}{16}n} \quad & \end{aligned}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn\log n$ and get

$$\begin{aligned}
T(n) &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\
&\leq 2\left(d\left\lceil \frac{n}{2} \right\rceil \log\left\lceil \frac{n}{2} \right\rceil\right) + cn \\
\boxed{\left\lceil \tfrac{n}{2} \right\rceil \leq \tfrac{n}{2} + 1} \quad &\leq 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\
\boxed{\tfrac{n}{2} + 1 \leq \tfrac{9}{16}n} \quad &\leq dn\log\left(\frac{9}{16}n\right) + 2d\log n + cn
\end{aligned}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn \log n$ and get

$$\begin{aligned} T(n) &\le 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\ &\le 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \\ \boxed{\left\lceil \tfrac{n}{2} \right\rceil \le \tfrac{n}{2} + 1} \quad &\le 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\ \boxed{\tfrac{n}{2} + 1 \le \tfrac{9}{16} n} \quad &\le dn \log\left(\frac{9}{16} n\right) + 2d \log n + cn \end{aligned}$$

$$\boxed{\log \tfrac{9}{16} n = \log n + (\log 9 - 4)}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn\log n$ and get

$$\begin{aligned}
T(n) &\le 2T\left(\left\lceil \frac{n}{2}\right\rceil\right) + cn \\
&\le 2\left(d\left\lceil \frac{n}{2}\right\rceil \log\left\lceil \frac{n}{2}\right\rceil\right) + cn \\
\boxed{\left\lceil \tfrac{n}{2}\right\rceil \le \tfrac{n}{2}+1} \quad &\le 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\
\boxed{\tfrac{n}{2}+1 \le \tfrac{9}{16}n} \quad &\le dn\log\left(\frac{9}{16}n\right) + 2d\log n + cn \\
\boxed{\log\tfrac{9}{16}n = \log n + (\log 9 - 4)} \quad &= dn\log n + (\log 9 - 4)dn + 2d\log n + cn
\end{aligned}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn\log n$ and get

$$\begin{aligned}
T(n) &\le 2T\left(\left\lceil \frac{n}{2}\right\rceil\right) + cn \\
&\le 2\left(d\left\lceil \frac{n}{2}\right\rceil \log\left\lceil \frac{n}{2}\right\rceil\right) + cn \\
\boxed{\left\lceil \tfrac{n}{2}\right\rceil \le \tfrac{n}{2}+1} \quad &\le 2\left(d(n/2+1)\log(n/2+1)\right) + cn \\
\boxed{\tfrac{n}{2}+1 \le \tfrac{9}{16}n} \quad &\le dn\log\left(\frac{9}{16}n\right) + 2d\log n + cn \\
\boxed{\log \tfrac{9}{16}n = \log n + (\log 9 - 4)} \quad &= dn\log n + (\log 9 - 4)dn + 2d\log n + cn
\end{aligned}$$

$$\boxed{\log n \le \tfrac{n}{4}}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn\log n$ and get

$$\begin{aligned}
T(n) &\le 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\
&\le 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \\
\boxed{\left\lceil \tfrac{n}{2} \right\rceil \le \tfrac{n}{2} + 1} \quad &\le 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\
\boxed{\tfrac{n}{2} + 1 \le \tfrac{9}{16}n} \quad &\le dn\log\left(\frac{9}{16}n\right) + 2d\log n + cn \\
\boxed{\log \tfrac{9}{16}n = \log n + (\log 9 - 4)} \quad &= dn\log n + (\log 9 - 4)dn + 2d\log n + cn \\
\boxed{\log n \le \tfrac{n}{4}} \quad &= dn\log n + (\log 9 - 3.5)dn + cn
\end{aligned}$$

6.1 Guessing+Induction

We also make a guess of $T(n) \leq dn \log n$ and get

$$
\begin{aligned}
& & T(n) &\leq 2T\left(\left\lceil \frac{n}{2} \right\rceil\right) + cn \\
& & &\leq 2\left(d\left\lceil \frac{n}{2} \right\rceil \log \left\lceil \frac{n}{2} \right\rceil\right) + cn \\
&\boxed{\left\lceil \tfrac{n}{2} \right\rceil \leq \tfrac{n}{2} + 1} & &\leq 2\big(d(n/2+1)\log(n/2+1)\big) + cn \\
&\boxed{\tfrac{n}{2} + 1 \leq \tfrac{9}{16}n} & &\leq dn \log\left(\frac{9}{16}n\right) + 2d\log n + cn \\
&\boxed{\log \tfrac{9}{16}n = \log n + (\log 9 - 4)} & &= dn\log n + (\log 9 - 4)dn + 2d\log n + cn \\
&\boxed{\log n \leq \tfrac{n}{4}} & &= dn\log n + (\log 9 - 3.5)dn + cn \\
& & &\leq dn\log n - 0.33dn + cn
\end{aligned}
$$

6.1 Guessing+Induction

We also make a guess of $T(n) \le dn\log n$ and get

$$
\begin{aligned}
& T(n) && \le 2T\left(\left\lceil\frac{n}{2}\right\rceil\right) + cn \\
& && \le 2\left(d\left\lceil\frac{n}{2}\right\rceil\log\left\lceil\frac{n}{2}\right\rceil\right) + cn \\
& \boxed{\left\lceil\tfrac{n}{2}\right\rceil \le \tfrac{n}{2} + 1} && \le 2\bigl(d(n/2+1)\log(n/2+1)\bigr) + cn \\
& \boxed{\tfrac{n}{2} + 1 \le \tfrac{9}{16}n} && \le dn\log\left(\frac{9}{16}n\right) + 2d\log n + cn \\
& \boxed{\log\tfrac{9}{16}n = \log n + (\log 9 - 4)} && = dn\log n + (\log 9 - 4)dn + 2d\log n + cn \\
& \boxed{\log n \le \tfrac{n}{4}} && = dn\log n + (\log 9 - 3.5)dn + cn \\
& && \le dn\log n - 0.33dn + cn \\
& && \le dn\log n
\end{aligned}
$$

for a suitable choice of d.

6.2 Master Theorem

Lemma 4

Let $a \geq 1, b \geq 1$ and $\epsilon > 0$ denote constants. Consider the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \ .$$

Case 1.

If $f(n) = \mathcal{O}(n^{\log_b(a)-\epsilon})$ then $T(n) = \Theta(n^{\log_b a})$.

Case 2.

If $f(n) = \Theta(n^{\log_b(a)} \log^k n)$ then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.

Case 3.

If $f(n) = \Omega(n^{\log_b(a)+\epsilon})$ and for sufficiently large n $af(\frac{n}{b}) \leq cf(n)$ for some constant $c < 1$ then $T(n) = \Theta(f(n))$.

6.2 Master Theorem

We prove the Master Theorem for the case that n is of the form b^ℓ, and we assume that the non-recursive case occurs for problem size 1 and incurs cost 1.

The Recursion Tree

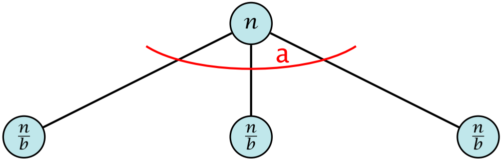
The running time of a recursive algorithm can be visualized by a recursion tree:

The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:

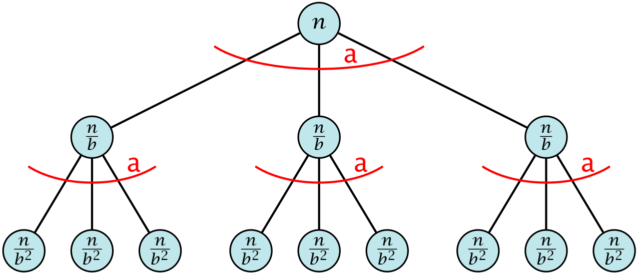
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



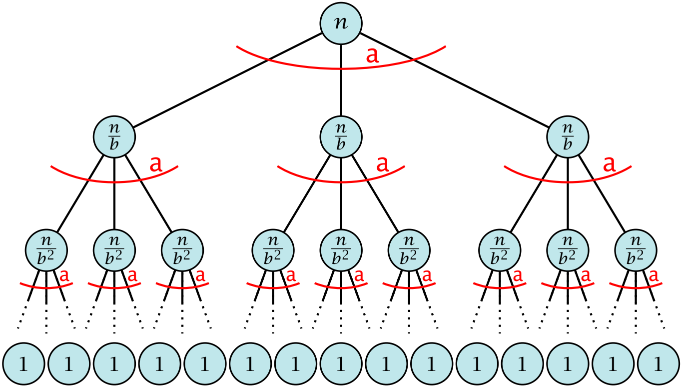
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



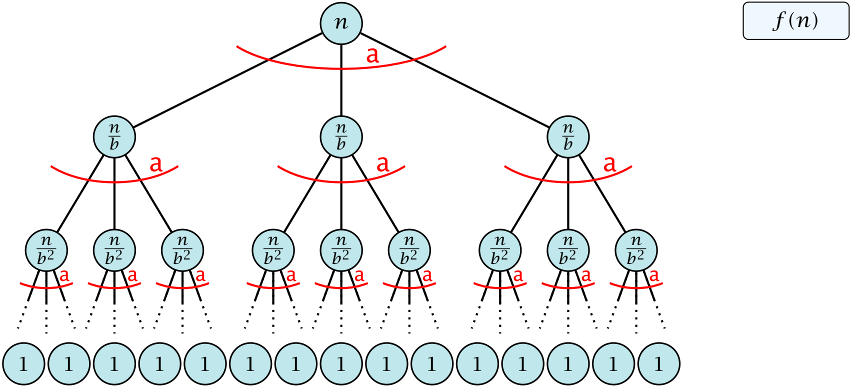
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



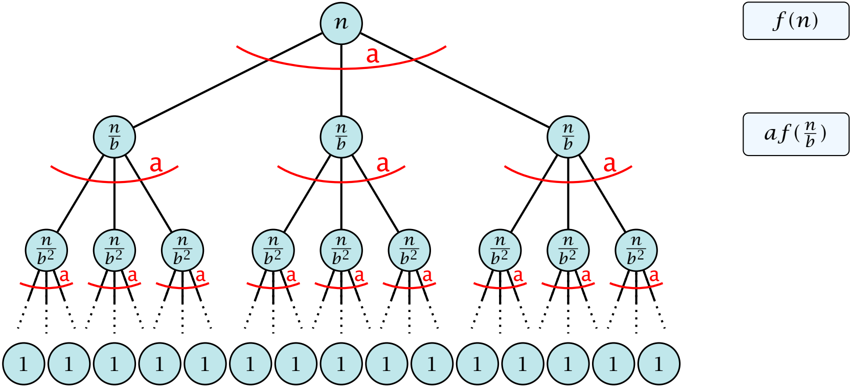
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:

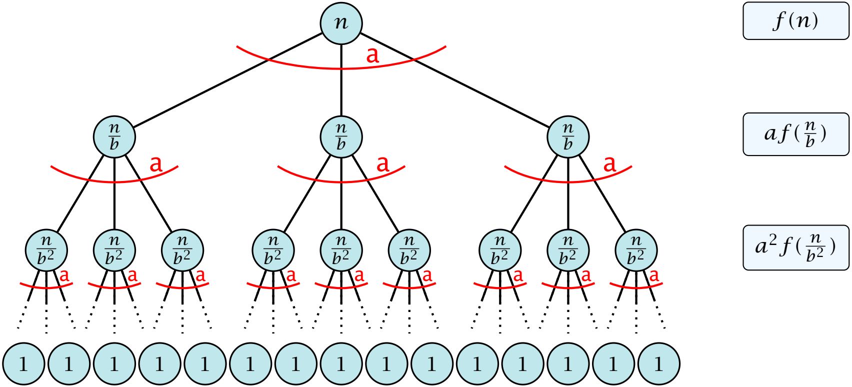


$f(n)$

$af(\frac{n}{b})$

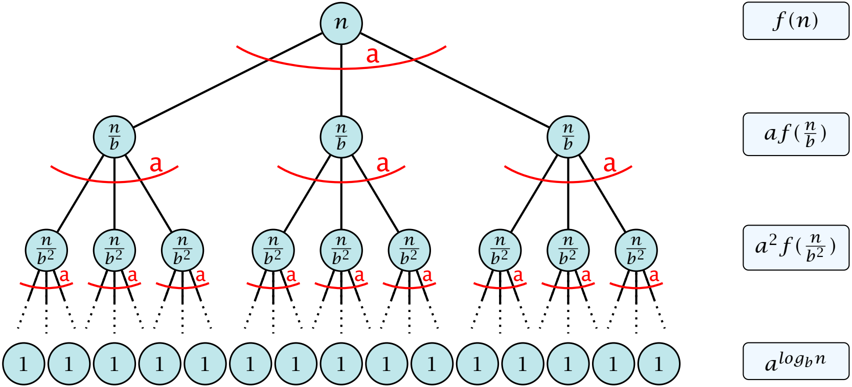
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



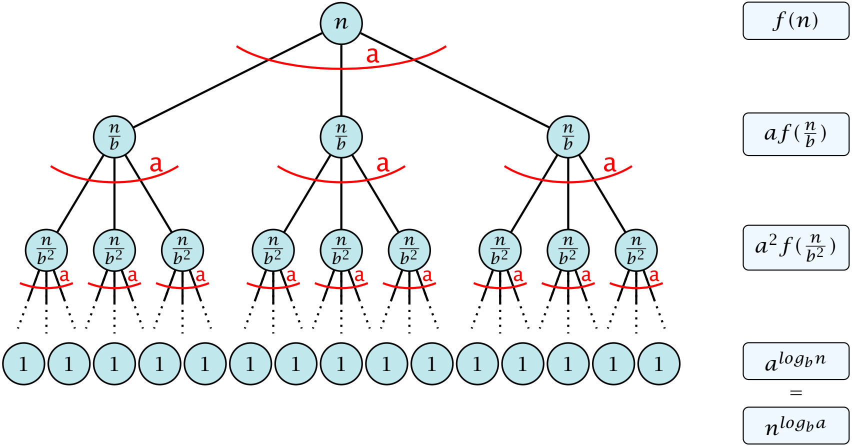
The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



The Recursion Tree

The running time of a recursive algorithm can be visualized by a recursion tree:



$f(n)$

$af(\frac{n}{b})$

$a^2f(\frac{n}{b^2})$

$a^{\log_b n}$
$=$
$n^{\log_b a}$

6.2 Master Theorem

This gives

$$T(n) = n^{\log_b a} + \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) .$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a - \epsilon}$.

Case 1. Now suppose that $f(n) \leq cn^{\log_b a - \epsilon}$.

$$T(n) - n^{\log_b a}$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a-\epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \end{aligned}$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a - \epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}}$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
&= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^{\epsilon}\right)^i
\end{aligned}
$$

$\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}}$

Case 1. Now suppose that $f(n) \le cn^{\log_b a-\epsilon}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon}$$

$$\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}} \quad = cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i$$

$$\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}}$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}} \quad &= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^{\epsilon})^i \\
\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a-\epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1)
\end{aligned}
$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} \quad &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \\
\boxed{\textstyle\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1)
\end{aligned}
$$

Case 1. Now suppose that $f(n) \le cn^{\log_b a-\epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \\
\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}} \quad &= cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} (b^\epsilon)^i \\
\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a-\epsilon}(b^{\epsilon\log_b n} - 1)/(b^\epsilon - 1) \\
&= cn^{\log_b a-\epsilon}(n^\epsilon - 1)/(b^\epsilon - 1) \\
&= \frac{c}{b^\epsilon - 1} n^{\log_b a}(n^\epsilon - 1)/(n^\epsilon)
\end{aligned}
$$

Case 1. Now suppose that $f(n) \leq cn^{\log_b a - \epsilon}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a - \epsilon} \\
\boxed{b^{-i(\log_b a - \epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i} a^{-i}} \quad &= cn^{\log_b a - \epsilon} \sum_{i=0}^{\log_b n - 1} \left(b^{\epsilon}\right)^i \\
\boxed{\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad &= cn^{\log_b a - \epsilon}(b^{\epsilon \log_b n} - 1)/(b^{\epsilon} - 1) \\
&= cn^{\log_b a - \epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1) \\
&= \frac{c}{b^{\epsilon} - 1} n^{\log_b a}(n^{\epsilon} - 1)/(n^{\epsilon})
\end{aligned}
$$

Hence,

$$
T(n) \leq \left(\frac{c}{b^{\epsilon} - 1} + 1\right) n^{\log_b (a)}
$$

Case 1. Now suppose that $f(n) \le cn^{\log_b a-\epsilon}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a-\epsilon} \end{aligned}$$

$$\boxed{b^{-i(\log_b a-\epsilon)} = b^{\epsilon i}(b^{\log_b a})^{-i} = b^{\epsilon i}a^{-i}} \quad = cn^{\log_b a-\epsilon} \sum_{i=0}^{\log_b n-1} \left(b^{\epsilon}\right)^i$$

$$\boxed{\textstyle\sum_{i=0}^{k} q^i = \frac{q^{k+1}-1}{q-1}} \quad = cn^{\log_b a-\epsilon}(b^{\epsilon\log_b n} - 1)/(b^{\epsilon} - 1)$$

$$\begin{aligned} &= cn^{\log_b a-\epsilon}(n^{\epsilon} - 1)/(b^{\epsilon} - 1) \\ &= \frac{c}{b^{\epsilon} - 1} n^{\log_b a}(n^{\epsilon} - 1)/(n^{\epsilon}) \end{aligned}$$

Hence,

$$T(n) \le \left(\frac{c}{b^{\epsilon} - 1} + 1\right) n^{\log_b (a)} \qquad \boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a}).}$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}$.

Case 2. Now suppose that $f(n) \le cn^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \end{aligned}$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \end{aligned}$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\ &= cn^{\log_b a} \log_b n \end{aligned}$$

Hence,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n)$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\ &= cn^{\log_b a} \log_b n \end{aligned}$$

Hence,

$$T(n) = \mathcal{O}(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log n).}$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$T(n) - n^{\log_b a}$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right)$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a}
\end{aligned}
$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\geq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\ &= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \end{aligned}$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n - 1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Hence,

$$
T(n) = \Omega(n^{\log_b a} \log_b n)
$$

Case 2. Now suppose that $f(n) \geq cn^{\log_b a}$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\geq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \\
&= cn^{\log_b a} \sum_{i=0}^{\log_b n-1} 1 \\
&= cn^{\log_b a} \log_b n
\end{aligned}
$$

Hence,

$$T(n) = \Omega(n^{\log_b a} \log_b n) \qquad \boxed{\Rightarrow T(n) = \Omega(n^{\log_b a} \log n).}$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a}$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

Case 2. Now suppose that $f(n) \le c n^{\log_b a} (\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k
\end{aligned}
$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \end{aligned}$$

$$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

$$= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
\boxed{n = b^\ell \Rightarrow \ell = \log_b n} \quad &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k
\end{aligned}
$$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
\boxed{n = b^\ell \Rightarrow \ell = \log_b n} \quad &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k
\end{aligned}$$

Case 2. Now suppose that $f(n) \le c n^{\log_b a} (\log_b(n))^k$.

$$\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n - 1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n - 1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= c n^{\log_b a} \sum_{i=0}^{\ell - 1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= c n^{\log_b a} \sum_{i=0}^{\ell - 1} (\ell - i)^k \\
&= c n^{\log_b a} \boxed{\sum_{i=1}^{\ell} i^k} \approx \tfrac{1}{k} \ell^{k+1}
\end{aligned}$$

$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$

Case 2. Now suppose that $f(n) \leq cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\leq c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
\boxed{n = b^\ell \Rightarrow \ell = \log_b n} \quad &= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

Case 2. Now suppose that $f(n) \le cn^{\log_b a}(\log_b(n))^k$.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&\le c \sum_{i=0}^{\log_b n-1} a^i \left(\frac{n}{b^i}\right)^{\log_b a} \cdot \left(\log_b\left(\frac{n}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} \left(\log_b\left(\frac{b^\ell}{b^i}\right)\right)^k \\
&= cn^{\log_b a} \sum_{i=0}^{\ell-1} (\ell - i)^k \\
&= cn^{\log_b a} \sum_{i=1}^{\ell} i^k \\
&\approx \frac{c}{k} n^{\log_b a} \ell^{k+1}
\end{aligned}
$$

$$\boxed{n = b^\ell \Rightarrow \ell = \log_b n}$$

$$\boxed{\Rightarrow T(n) = \mathcal{O}(n^{\log_b a} \log^{k+1} n).}$$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &= \sum_{i=0}^{\log_b n-1} c^i f(n) + \mathcal{O}(n^{\log_b a}) \end{aligned}$$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &= \sum_{i=0}^{\log_b n-1} c^i f(n) + \mathcal{O}(n^{\log_b a}) \end{aligned}$$

$q < 1 : \sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q}$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$\begin{aligned} T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\ &= \sum_{i=0}^{\log_b n-1} c^i f(n) + \mathcal{O}(n^{\log_b a}) \\ &\leq \frac{1}{1-c} f(n) + \mathcal{O}(n^{\log_b a}) \end{aligned}$$

$q < 1 : \sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q}$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$
\begin{aligned}
T(n) - n^{\log_b a} &= \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right) \\
&= \sum_{i=0}^{\log_b n-1} c^i f(n) + \mathcal{O}(n^{\log_b a}) \\
&\leq \frac{1}{1-c} f(n) + \mathcal{O}(n^{\log_b a})
\end{aligned}
$$

$q < 1 : \sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q}$

Hence,

$$T(n) \leq \mathcal{O}(f(n))$$

Case 3. Now suppose that $f(n) \geq dn^{\log_b a+\epsilon}$, and that for sufficiently large n: $af(n/b) \leq cf(n)$, for $c < 1$.

From this we get $a^i f(n/b^i) \leq c^i f(n)$, where we assume that $n/b^{i-1} \geq n_0$ is still sufficiently large.

$$T(n) - n^{\log_b a} = \sum_{i=0}^{\log_b n-1} a^i f\left(\frac{n}{b^i}\right)$$

$$= \sum_{i=0}^{\log_b n-1} c^i f(n) + \mathcal{O}(n^{\log_b a})$$

$$\boxed{q < 1 : \sum_{i=0}^{n} q^i = \frac{1-q^{n+1}}{1-q} \leq \frac{1}{1-q}} \quad \leq \frac{1}{1-c} f(n) + \mathcal{O}(n^{\log_b a})$$

Hence,

$$T(n) \leq \mathcal{O}(f(n)) \qquad \boxed{\Rightarrow T(n) = \Theta(f(n)).}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

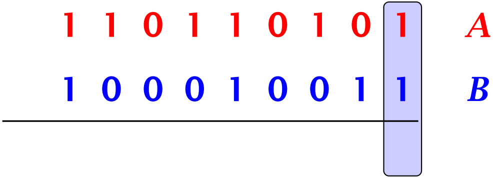
For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
\hline
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:



$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & \boxed{1} & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \boxed{1} & & B \\
\hline
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

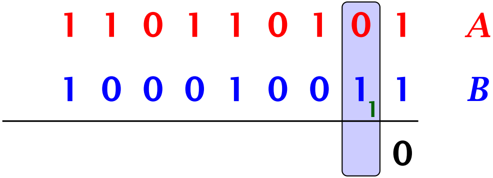
For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccc|c|c}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1_{1} & 1 & & B \\
\hline
 & & & & & & & & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

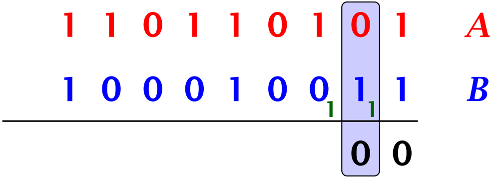


$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & \boxed{0} & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1_{1} & 1 & & B \\
\hline
 & & & & & & & & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

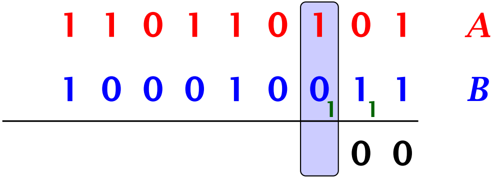


$$\begin{array}{cccccccccccc} 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & & A \\ 1 & 0 & 0 & 0 & 1 & 0 & 0_{1} & 1_{1} & 1 & & & B \\ \hline & & & & & & & 0 & 0 & & & \end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

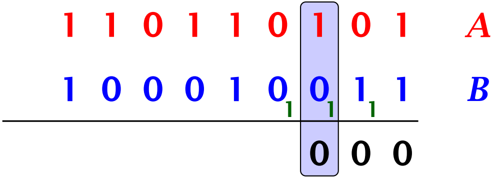


$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & \boxed{1} & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0_{1} & 1_{1} & 1 & & B \\
\hline
 & & & & & & & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

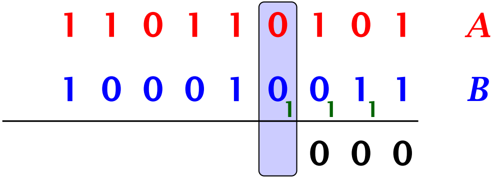


$$\begin{array}{cccccccccc} 1 & 1 & 0 & 1 & 1 & 0 & \boxed{1} & 0 & 1 & A \\ 1 & 0 & 0 & 0 & 1 & 0_1 & \boxed{0_1} & 1_1 & 1 & B \\ \hline & & & & & & \boxed{0} & 0 & 0 & \end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

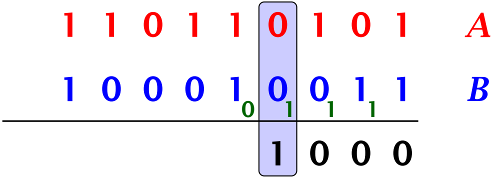


$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0_1 & 0_1 & 1_1 & 1 & & B \\
\hline
 & & & & & & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

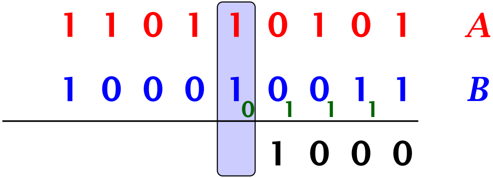
For this we first need to be able to add two integers A and B:



Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

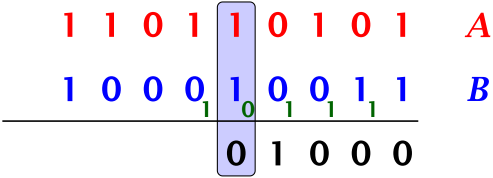


$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & & & {}_{0} & {}_{1} & {}_{1} & {}_{1} & & & \\
\hline
 & & & & & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

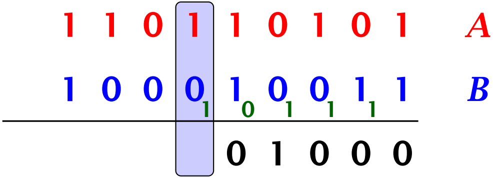
For this we first need to be able to add two integers A and B:



Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:



Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

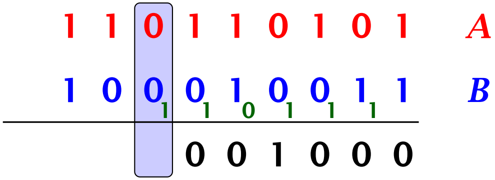
For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
 & & {}_1 & {}_1 & {}_0 & {}_1 & {}_1 & {}_1 & & & \\
\hline
 & & & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:



Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

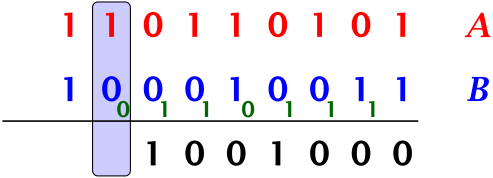
For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccccc}
1 & 1 & \boxed{0} & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0_0 & \boxed{0_1} & 0_1 & 1_0 & 0_1 & 0_1 & 1_1 & 1 & & B \\
\hline
 & & \boxed{1} & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:



$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1 & 0_0 & 0_1 & 0_1 & 1_0 & 0_1 & 0_1 & 1_1 & 1 & & B \\
\hline
 & & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

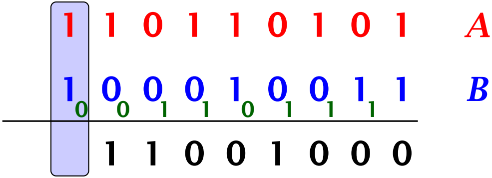
For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccccc}
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
1_0 & 0_0 & 0_1 & 0_1 & 1_0 & 0_1 & 0_1 & 1_1 & 1 & & B \\
\hline
 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:



Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

$$\begin{array}{cccccccccccc}
 & \boxed{1} & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & & A \\
 & \boxed{1} & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & & B \\
{\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & & \\
\hline
 & \boxed{0} & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & &
\end{array}$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

$$
\begin{array}{c|ccccccccc|c}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & & \\
\hline
 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

$$
\begin{array}{c|ccccccccc|l}
 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & A \\
 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & B \\
{\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & & \\
\hline
1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 &
\end{array}
$$

Example: Multiplying Two Integers

Suppose we want to multiply two n-bit Integers, but our registers can only perform operations on integers of constant size.

For this we first need to be able to add two integers A and B:

$$\begin{array}{ccccccccccl} & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & \quad A \\ & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & \quad B \\ _1 & _0 & _0 & _1 & _1 & _0 & _1 & _1 & _1 & & \\ \hline 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \end{array}$$

This gives that two n-bit integers can be added in time $\mathcal{O}(n)$.

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \leq n$).

$$\underline{1\;0\;0\;0\;1 \times 1\;0\;1\;1}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\underline{1\;0\;0\;0\;1 \times 1\;0\;1\;\boxed{1}}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc} 1&0&0&0&1&\times&1&0&1&\boxed{1}\\ \hline &&&&&1&0&0&0&1 \end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\ \hline & & & & & 1 & 0 & 0 & 0 & 1 \end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & & & & & & 0
\end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc} 1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & \boxed{1} & 1 \\ \hline & & & & & 1 & 0 & 0 & 0 & 1 \\ & & & & 1 & 0 & 0 & 0 & 1 & 0 \end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0
\end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & & & & & & 0 & 0
\end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \leq n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & \boxed{0} & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & & & & & & 0 & 0 & 0
\end{array}
$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \leq n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & \boxed{1} & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0
\end{array}
$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
\end{array}$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

Time requirement:

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B ($m \le n$).

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

Time requirement:

- Computing intermediate results: $\mathcal{O}(nm)$.

Example: Multiplying Two Integers

Suppose that we want to multiply an n-bit integer A and an m-bit integer B $(m \leq n)$.

$$
\begin{array}{cccccccccc}
1 & 0 & 0 & 0 & 1 & \times & 1 & 0 & 1 & 1 \\
\hline
 & & & & & 1 & 0 & 0 & 0 & 1 \\
 & & & & 1 & 0 & 0 & 0 & 1 & 0 \\
 & & & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 & & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\hline
 & & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
$$

Time requirement:

- Computing intermediate results: $\mathcal{O}(nm)$.
- Adding m numbers of length $\leq 2n$: $\mathcal{O}((m+n)m) = \mathcal{O}(nm)$.

Example: Multiplying Two Integers

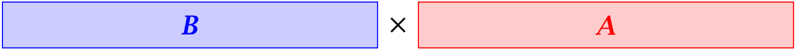
A recursive approach:

Suppose that integers A and B are of length $n = 2^k$, for some k.

Example: Multiplying Two Integers

A recursive approach:

Suppose that integers A and B are of length $n = 2^k$, for some k.



$B \times A$

Example: Multiplying Two Integers

A recursive approach:

Suppose that integers A and B are of length $n = 2^k$, for some k.

$$b_n \quad \cdots \quad b_0 \times a_n \quad \cdots \quad a_0$$

Example: Multiplying Two Integers

A recursive approach:

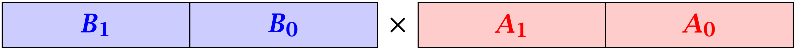
Suppose that integers A and B are of length $n = 2^k$, for some k.

$$\boxed{b_n \quad \cdots \quad b_{\frac{n}{2}} \; b_{\frac{n}{2}-1} \quad \cdots \quad b_0} \times \boxed{a_n \quad \cdots \quad a_{\frac{n}{2}} \; a_{\frac{n}{2}-1} \quad \cdots \quad a_0}$$

Example: Multiplying Two Integers

A recursive approach:

Suppose that integers A and B are of length $n = 2^k$, for some k.

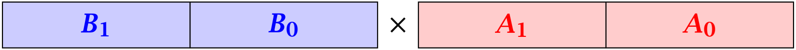


B_1 B_0 × A_1 A_0

Example: Multiplying Two Integers

A recursive approach:

Suppose that integers A and B are of length $n = 2^k$, for some k.



B_1 B_0 × A_1 A_0

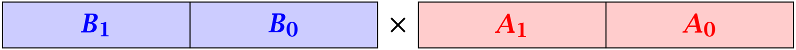
Then it holds that

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ and } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Example: Multiplying Two Integers

A recursive approach:

Suppose that integers **A** and **B** are of length $n = 2^k$, for some k.



Then it holds that

$$A = A_1 \cdot 2^{\frac{n}{2}} + A_0 \text{ and } B = B_1 \cdot 2^{\frac{n}{2}} + B_0$$

Hence,

$$A \cdot B = A_1 B_1 \cdot 2^n + (A_1 B_0 + A_0 B_1) \cdot 2^{\frac{n}{2}} + A_0 \cdot B_0$$

Example: Multiplying Two Integers

Algorithm 3 mult(A, B)

```
if $|A| = |B| = 1$ then
    return $a_0 \cdot b_0$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

```
Algorithm 3 mult(A, B)
if $|A| = |B| = 1$ then
    return $a_0 \cdot b_0$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

$\mathcal{O}(1)$

Example: Multiplying Two Integers

```
Algorithm 3 mult($A, B$)
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                               $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

Algorithm 3 mult(A, B)

```
if $|A| = |B| = 1$ then                           $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                        $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                    $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

```
**Algorithm 3** mult$(A, B)$
if $|A| = |B| = 1$ then                          $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                       $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                       $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

Algorithm 3 mult(A, B)

```
if $|A| = |B| = 1$ then                          $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                       $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                   $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                   $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$         $T(\frac{n}{2})$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

Algorithm 3 mult(A, B)

```
if $|A| = |B| = 1$ then                              $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                       $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                       $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$             $T(\frac{n}{2})$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$    $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

Algorithm 3 mult(A, B)

```
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                           $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                           $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$                 $T(\frac{n}{2})$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$    $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$                 $T(\frac{n}{2})$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

```
Algorithm 3 mult$(A, B)$
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                            $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                            $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                            $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$                  $T(\frac{n}{2})$
$Z_1 \leftarrow \mathrm{mult}(A_1, B_0) + \mathrm{mult}(A_0, B_1)$    $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$                  $T(\frac{n}{2})$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$  $\mathcal{O}(n)$
```

Example: Multiplying Two Integers

```
Algorithm 3 mult($A, B$)
if $|A| = |B| = 1$ then                              $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                       $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                       $\mathcal{O}(n)$
$Z_2 \leftarrow \text{mult}(A_1, B_1)$               $T(\frac{n}{2})$
$Z_1 \leftarrow \text{mult}(A_1, B_0) + \text{mult}(A_0, B_1)$   $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_0 \leftarrow \text{mult}(A_0, B_0)$               $T(\frac{n}{2})$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$         $\mathcal{O}(n)$
```

We get the following recurrence:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Example: Multiplying Two Integers

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a}\log^k n)$ $\quad T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\quad T(n) = \Theta(f(n))$

Example: Multiplying Two Integers

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $T(n) = \Theta(f(n))$

In our case $a = 4$, $b = 2$, and $f(n) = \Theta(n)$. Hence, we are in Case 1, since $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

Example: Multiplying Two Integers

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a-\epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a}\log^k n)$ $\quad T(n) = \Theta(n^{\log_b a}\log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a+\epsilon})$ $\quad T(n) = \Theta(f(n))$

In our case $a = 4$, $b = 2$, and $f(n) = \Theta(n)$. Hence, we are in Case 1, since $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a-\epsilon})$.

We get a running time of $\mathcal{O}(n^2)$ for our algorithm.

Example: Multiplying Two Integers

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\quad T(n) = \Theta(f(n))$

In our case $a = 4$, $b = 2$, and $f(n) = \Theta(n)$. Hence, we are in Case 1, since $n = \mathcal{O}(n^{2-\epsilon}) = \mathcal{O}(n^{\log_b a - \epsilon})$.

We get a running time of $\mathcal{O}(n^2)$ for our algorithm.

⇒ Not better then the “school method”.

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$Z_1 = A_1B_0 + A_0B_1$$

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - A_1B_1 - A_0B_0 \end{aligned}$$

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

Algorithm 4 mult(A, B)

```
if $|A| = |B| = 1$ then
    return $a_0 \cdot b_0$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult(A, B)
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
    return $a_0 \cdot b_0$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult($A, B$)
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \text{mult}(A_1, B_1)$
$Z_0 \leftarrow \text{mult}(A_0, B_0)$
$Z_1 \leftarrow \text{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{=Z_2} - \overbrace{A_0B_0}^{=Z_0} \end{aligned}$$

Hence,

Algorithm 4 mult(A,B)

```
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                                $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                            $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1) \cdot (B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult($A, B$)
if $|A| = |B| = 1$ then                                   $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                            $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                            $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                            $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

Algorithm 4 mult(A, B)

```
if $|A| = |B| = 1$ then                              $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                        $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                        $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                        $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$              $T(\frac{n}{2})$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult($A, B$)
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                             $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                           $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                           $\mathcal{O}(n)$
$Z_2 \leftarrow \mathrm{mult}(A_1, B_1)$                  $T(\frac{n}{2})$
$Z_0 \leftarrow \mathrm{mult}(A_0, B_0)$                  $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_1 \leftarrow \mathrm{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{= Z_2} - \overbrace{A_0B_0}^{= Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult(A, B)
if $|A| = |B| = 1$ then                                  $\mathcal{O}(1)$
        return $a_0 \cdot b_0$                           $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                           $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                           $\mathcal{O}(n)$
$Z_2 \leftarrow \text{mult}(A_1, B_1)$                   $T(\frac{n}{2})$
$Z_0 \leftarrow \text{mult}(A_0, B_0)$                   $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_1 \leftarrow \text{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$   $T(\frac{n}{2})$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$
```

Example: Multiplying Two Integers

We can use the following identity to compute Z_1:

$$\begin{aligned} Z_1 &= A_1B_0 + A_0B_1 \\ &= (A_0 + A_1)\cdot(B_0 + B_1) - \overbrace{A_1B_1}^{=\,Z_2} - \overbrace{A_0B_0}^{=\,Z_0} \end{aligned}$$

Hence,

```
Algorithm 4 mult($A, B$)
if $|A| = |B| = 1$ then                                   $\mathcal{O}(1)$
    return $a_0 \cdot b_0$                                 $\mathcal{O}(1)$
split $A$ into $A_0$ and $A_1$                            $\mathcal{O}(n)$
split $B$ into $B_0$ and $B_1$                            $\mathcal{O}(n)$
$Z_2 \leftarrow \text{mult}(A_1, B_1)$                    $T(\frac{n}{2})$
$Z_0 \leftarrow \text{mult}(A_0, B_0)$                    $2T(\frac{n}{2}) + \mathcal{O}(n)$
$Z_1 \leftarrow \text{mult}(A_0 + A_1, B_0 + B_1) - Z_2 - Z_0$   $T(\frac{n}{2})$
return $Z_2 \cdot 2^n + Z_1 \cdot 2^{\frac{n}{2}} + Z_0$   $\mathcal{O}(n)$
```

Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \quad .$$

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\quad T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

A huge improvement over the "school method".

Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \enspace .$$

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\qquad T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

A huge improvement over the "school method".

Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \enspace .$$

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\quad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\quad T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

A huge improvement over the “school method”.

Example: Multiplying Two Integers

We get the following recurrence:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n) \ .$$

Master Theorem: Recurrence: $T[n] = aT(\frac{n}{b}) + f(n)$.

- Case 1: $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ $\qquad T(n) = \Theta(n^{\log_b a})$
- Case 2: $f(n) = \Theta(n^{\log_b a} \log^k n)$ $\quad T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
- Case 3: $f(n) = \Omega(n^{\log_b a + \epsilon})$ $\qquad T(n) = \Theta(f(n))$

Again we are in Case 1. We get a running time of $\Theta(n^{\log_2 3}) \approx \Theta(n^{1.59})$.

A huge improvement over the “school method”.

6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

This is the general form of a linear recurrence relation of order k with constant coefficients ($c_0, c_k \neq 0$).

- $T(n)$ only depends on the k preceding values. This means the recurrence relation is of order k.
- The recurrence is linear as there are no products of $T[n]$'s.
- If $f(n) = 0$ then the recurrence relation becomes a linear, homogenous recurrence relation of order k.

6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

This is the general form of a linear recurrence relation of order k with constant coefficients ($c_0, c_k \neq 0$).

- $T(n)$ only depends on the k preceding values. This means the recurrence relation is of order k.
- The recurrence is linear as there are no products of $T[n]$'s.
- If $f(n) = 0$ then the recurrence relation becomes a linear, homogenous recurrence relation of order k.

6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

This is the general form of a linear recurrence relation of order k with constant coefficients ($c_0, c_k \neq 0$).

- $T(n)$ only depends on the k preceding values. This means the recurrence relation is of order k.
- The recurrence is linear as there are no products of $T[n]$'s.
- If $f(n) = 0$ then the recurrence relation becomes a linear, homogenous recurrence relation of order k.

6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

This is the general form of a linear recurrence relation of order k with constant coefficients ($c_0, c_k \neq 0$).

- $T(n)$ only depends on the k preceding values. This means the recurrence relation is of order k.
- The recurrence is linear as there are no products of $T[n]$'s.
- If $f(n) = 0$ then the recurrence relation becomes a linear, homogenous recurrence relation of order k.

6.3 The Characteristic Polynomial

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

This is the general form of a linear recurrence relation of order k with constant coefficients ($c_0, c_k \neq 0$).

- $T(n)$ only depends on the k preceding values. This means the recurrence relation is of order k.
- The recurrence is linear as there are no products of $T[n]$'s.
- If $f(n) = 0$ then the recurrence relation becomes a linear, homogenous recurrence relation of order k.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \ldots$ is completely determined by a set of boundary conditions that specify values for $T[0], \ldots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

6.3 The Characteristic Polynomial

Observations:

- The solution $T[0], T[1], T[2], \dots$ is completely determined by a set of boundary conditions that specify values for $T[0], \dots, T[k-1]$.
- In fact, any k consecutive values completely determine the solution.
- k non-concecutive values might not be an appropriate set of boundary conditions (depends on the problem).

Approach:

- First determine all solutions that satisfy recurrence relation.
- Then pick the right one by analyzing boundary conditions.
- First consider the homogenous case.

The Homogenous Case

The solution space

$$S = \Big\{T = T[0], T[1], T[2], \dots \;\Big|\; T \text{ fulfills recurrence relation}\Big\}$$

is a vector space. This means that if $T_1, T_2 \in S$, then also $\alpha T_1 + \beta T_2 \in S$, for arbitrary constants α, β.

How do we find a non-trivial solution?

We guess that the solution is of the form λ^n, $\lambda \neq 0$, and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \cdots + c_k \cdot \lambda^{n-k} = 0$$

for all $n \geq k$.

The Homogenous Case

The solution space

$$S = \Big\{T = T[0], T[1], T[2], \dots \;\Big|\; T \text{ fulfills recurrence relation}\Big\}$$

is a vector space. This means that if $T_1, T_2 \in S$, then also $\alpha T_1 + \beta T_2 \in S$, for arbitrary constants α, β.

How do we find a non-trivial solution?

We guess that the solution is of the form λ^n, $\lambda \neq 0$, and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \cdots + c_k \cdot \lambda^{n-k} = 0$$

for all $n \geq k$.

The Homogenous Case

The solution space

$$S = \Big\{T = T[0], T[1], T[2], \dots \;\Big|\; T \text{ fulfills recurrence relation}\Big\}$$

is a vector space. This means that if $T_1, T_2 \in S$, then also $\alpha T_1 + \beta T_2 \in S$, for arbitrary constants α, β.

How do we find a non-trivial solution?

We guess that the solution is of the form λ^n, $\lambda \neq 0$, and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \cdots + c_k \cdot \lambda^{n-k} = 0$$

for all $n \geq k$.

The Homogenous Case

The solution space

$$S = \Big\{T = T[0], T[1], T[2], \ldots \;\Big|\; T \text{ fulfills recurrence relation}\Big\}$$

is a vector space. This means that if $T_1, T_2 \in S$, then also $\alpha T_1 + \beta T_2 \in S$, for arbitrary constants α, β.

How do we find a non-trivial solution?

We guess that the solution is of the form λ^n, $\lambda \neq 0$, and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \cdots + c_k \cdot \lambda^{n-k} = 0$$

for all $n \geq k$.

The Homogenous Case

The solution space

$$S = \Big\{T = T[0], T[1], T[2], \dots \;\Big|\; T \text{ fulfills recurrence relation}\Big\}$$

is a vector space. This means that if $T_1, T_2 \in S$, then also $\alpha T_1 + \beta T_2 \in S$, for arbitrary constants α, β.

How do we find a non-trivial solution?

We guess that the solution is of the form λ^n, $\lambda \neq 0$, and see what happens. In order for this guess to fulfill the recurrence we need

$$c_0\lambda^n + c_1\lambda^{n-1} + c_2 \cdot \lambda^{n-2} + \cdots + c_k \cdot \lambda^{n-k} = 0$$

for all $n \geq k$.

The Homogenous Case

Dividing by λ^{n-k} gives that all these constraints are identical to

$$c_0\lambda^k + c_1\lambda^{k-1} + c_2 \cdot \lambda^{k-2} + \cdots + c_k = 0$$

This means that if λ_i is a root (Nullstelle) of $P[\lambda]$ then $T[n] = \lambda_i^n$ is a solution to the recurrence relation.

Let $\lambda_1, \dots, \lambda_k$ be the k (complex) roots of $P[\lambda]$. Then, because of the vector space property

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n$$

is a solution for arbitrary values α_i.

The Homogenous Case

Dividing by λ^{n-k} gives that all these constraints are identical to

$$\underbrace{c_0\lambda^k + c_1\lambda^{k-1} + c_2\cdot\lambda^{k-2} + \cdots + c_k}_{\text{characteristic polynomial } P[\lambda]} = 0$$

This means that if λ_i is a root (Nullstelle) of $P[\lambda]$ then $T[n] = \lambda_i^n$ is a solution to the recurrence relation.

Let $\lambda_1, \dots, \lambda_k$ be the k (complex) roots of $P[\lambda]$. Then, because of the vector space property

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n$$

is a solution for arbitrary values α_i.

The Homogenous Case

Dividing by λ^{n-k} gives that all these constraints are identical to

$$\underbrace{c_0\lambda^k + c_1\lambda^{k-1} + c_2 \cdot \lambda^{k-2} + \cdots + c_k}_{\text{characteristic polynomial } P[\lambda]} = 0$$

This means that if λ_i is a root (Nullstelle) of $P[\lambda]$ then $T[n] = \lambda_i^n$ is a solution to the recurrence relation.

Let $\lambda_1, \dots, \lambda_k$ be the k (complex) roots of $P[\lambda]$. Then, because of the vector space property

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n$$

is a solution for arbitrary values α_i.

The Homogenous Case

Dividing by λ^{n-k} gives that all these constraints are identical to

$$\underbrace{c_0\lambda^k + c_1\lambda^{k-1} + c_2 \cdot \lambda^{k-2} + \cdots + c_k}_{\text{characteristic polynomial } P[\lambda]} = 0$$

This means that if λ_i is a root (Nullstelle) of $P[\lambda]$ then $T[n] = \lambda_i^n$ is a solution to the recurrence relation.

Let $\lambda_1, \ldots, \lambda_k$ be the k (complex) roots of $P[\lambda]$. Then, because of the vector space property

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n$$

is a solution for arbitrary values α_i.

The Homogenous Case

Lemma 5

Assume that the characteristic polynomial has k distinct roots $\lambda_1, \dots, \lambda_k$. Then all solutions to the recurrence relation are of the form

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n \enspace .$$

Proof.

There is one solution for every possible choice of boundary conditions for $T[1], \dots, T[k]$.

We show that the above set of solutions contains one solution for every choice of boundary conditions.

The Homogenous Case

Lemma 5

Assume that the characteristic polynomial has k distinct roots $\lambda_1,\dots,\lambda_k$. Then all solutions to the recurrence relation are of the form

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n \ .$$

Proof.

There is one solution for every possible choice of boundary conditions for $T[1],\dots,T[k]$.

We show that the above set of solutions contains one solution for every choice of boundary conditions.

The Homogenous Case

Lemma 5

Assume that the characteristic polynomial has k distinct roots $\lambda_1, \dots, \lambda_k$. Then all solutions to the recurrence relation are of the form

$$\alpha_1\lambda_1^n + \alpha_2\lambda_2^n + \cdots + \alpha_k\lambda_k^n \ .$$

Proof.

There is one solution for every possible choice of boundary conditions for $T[1], \dots, T[k]$.

We show that the above set of solutions contains one solution for every choice of boundary conditions.

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\alpha_1 \cdot \lambda_1 \quad + \quad \alpha_2 \cdot \lambda_2 \quad + \quad \cdots \quad + \quad \alpha_k \cdot \lambda_k \quad = \quad T[1]$$

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\begin{array}{ccccccccc}
\alpha_1\cdot\lambda_1 & + & \alpha_2\cdot\lambda_2 & + & \cdots & + & \alpha_k\cdot\lambda_k & = & T[1] \\
\alpha_1\cdot\lambda_1^2 & + & \alpha_2\cdot\lambda_2^2 & + & \cdots & + & \alpha_k\cdot\lambda_k^2 & = & T[2]
\end{array}$$

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\begin{array}{ccccccccc} \alpha_1 \cdot \lambda_1 & + & \alpha_2 \cdot \lambda_2 & + & \cdots & + & \alpha_k \cdot \lambda_k & = & T[1] \\ \alpha_1 \cdot \lambda_1^2 & + & \alpha_2 \cdot \lambda_2^2 & + & \cdots & + & \alpha_k \cdot \lambda_k^2 & = & T[2] \\ & & & & \vdots & & & & \end{array}$$

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\begin{array}{ccccccccc}
\alpha_1 \cdot \lambda_1 & + & \alpha_2 \cdot \lambda_2 & + & \cdots & + & \alpha_k \cdot \lambda_k & = & T[1] \\
\alpha_1 \cdot \lambda_1^2 & + & \alpha_2 \cdot \lambda_2^2 & + & \cdots & + & \alpha_k \cdot \lambda_k^2 & = & T[2] \\
 & & & & \vdots & & & & \\
\alpha_1 \cdot \lambda_1^k & + & \alpha_2 \cdot \lambda_2^k & + & \cdots & + & \alpha_k \cdot \lambda_k^k & = & T[k]
\end{array}$$

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\begin{pmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \cdots & \lambda_k^2 \\ & & \vdots & \\ \lambda_1^k & \lambda_2^k & \cdots & \lambda_k^k \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{pmatrix} = \begin{pmatrix} T[1] \\ T[2] \\ \vdots \\ T[k] \end{pmatrix}$$

The Homogenous Case

Proof (cont.).

Suppose I am given boundary conditions $T[i]$ and I want to see whether I can choose the $\alpha_i's$ such that these conditions are met:

$$\begin{pmatrix} \lambda_1 & \lambda_2 & \cdots & \lambda_k \\ \lambda_1^2 & \lambda_2^2 & \cdots & \lambda_k^2 \\ & & \vdots & \\ \lambda_1^k & \lambda_2^k & \cdots & \lambda_k^k \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_k \end{pmatrix} = \begin{pmatrix} T[1] \\ T[2] \\ \vdots \\ T[k] \end{pmatrix}$$

We show that the column vectors are linearly independent. Then the above equation has a solution.

The Homogenous Case

Proof (cont.).

The Homogenous Case

Proof (cont.).

This we show by induction:

☐

The Homogenous Case

Proof (cont.).

This we show by induction:

- **base case** ($k = 1$):
  A vector (λ_i), $\lambda_i \neq 0$ is linearly independent.

□

The Homogenous Case

Proof (cont.).

This we show by induction:

- **base case** $(k = 1)$:
  A vector (λ_i), $\lambda_i \neq 0$ is linearly independent.
- **induction step** $(k \to k+1)$:

□

The Homogenous Case

Proof (cont.).

This we show by induction:

- **base case** $(k = 1)$:
  A vector (λ_i), $\lambda_i \neq 0$ is linearly independent.
- **induction step** $(k \to k + 1)$:
  assume for contradiction that there exist α_i's with

$$\alpha_1 \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_1^{k-1} \\ \lambda_1^k \end{pmatrix} + \cdots + \alpha_k \begin{pmatrix} \lambda_k \\ \vdots \\ \lambda_k^{k-1} \\ \lambda_k^k \end{pmatrix} = 0$$

and not all $\alpha_i = 0$.

☐

The Homogenous Case

Proof (cont.).

This we show by induction:

- **base case** ($k = 1$):
  A vector (λ_i), $\lambda_i \neq 0$ is linearly independent.
- **induction step** ($k \to k+1$):
  assume for contradiction that there exist α_i's with

$$\alpha_1 \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_1^{k-1} \\ \lambda_1^k \end{pmatrix} + \cdots + \alpha_k \begin{pmatrix} \lambda_k \\ \vdots \\ \lambda_k^{k-1} \\ \lambda_k^k \end{pmatrix} = 0$$

and not all $\alpha_i = 0$. **Then all $\boldsymbol{\alpha_i \neq 0}$!** ☐

The Homogeneous Case

$$\alpha_1 \begin{pmatrix} \lambda_1 \\ \lambda_1^2 \\ \vdots \\ \lambda_1^{k-1} \\ \lambda_1^k \end{pmatrix} + \cdots + \alpha_k \begin{pmatrix} \lambda_1 \\ \lambda_k^2 \\ \vdots \\ \lambda_k^{k-1} \\ \lambda_k^k \end{pmatrix} = 0$$

The Homogeneous Case

$$\alpha_1 \begin{pmatrix} v_1 := \boxed{\begin{matrix} \lambda_1 \\ \lambda_1^2 \\ \vdots \\ \lambda_1^{k-1} \end{matrix}} \\ \lambda_1^k \end{pmatrix} + \cdots + \alpha_k \begin{pmatrix} v_k := \boxed{\begin{matrix} \lambda_1 \\ \lambda_k^2 \\ \vdots \\ \lambda_k^{k-1} \end{matrix}} \\ \lambda_k^k \end{pmatrix} = 0$$

The Homogeneous Case

$$\begin{array}{r} v_1 := \\ \\ \\ \\ \lambda_1 v_1 = \end{array} \alpha_1 \begin{pmatrix} \lambda_1 \\ \lambda_1^2 \\ \vdots \\ \lambda_1^{k-1} \\ \lambda_1^k \end{pmatrix} + \cdots + \begin{array}{r} v_k := \\ \\ \\ \\ \lambda_k v_k = \end{array} \alpha_k \begin{pmatrix} \lambda_1 \\ \lambda_k^2 \\ \vdots \\ \lambda_k^{k-1} \\ \lambda_k^k \end{pmatrix} = 0$$

The Homogeneous Case

$$\alpha_1 \left(\begin{array}{r|c} v_1 := & \lambda_1 \\ & \lambda_1^2 \\ & \vdots \\ & \lambda_1^{k-1} \\ \lambda_1 v_1 = & \lambda_1^k \end{array}\right) + \cdots + \alpha_k \left(\begin{array}{r|c} v_k := & \lambda_1 \\ & \lambda_k^2 \\ & \vdots \\ & \lambda_k^{k-1} \\ \lambda_k v_k = & \lambda_k^k \end{array}\right) = 0$$

This means that

$$\sum_{i=1}^{k} \alpha_i v_i = 0 \text{ and } \sum_{i=1}^{k} \lambda_i \alpha_i v_i = 0$$

The Homogeneous Case

$$\alpha_1 \left(\begin{array}{rc} v_1 := & \lambda_1 \\ & \lambda_1^2 \\ & \vdots \\ & \lambda_1^{k-1} \\ \lambda_1 v_1 = & \lambda_1^k \end{array}\right) + \cdots + \alpha_k \left(\begin{array}{rc} v_k := & \lambda_1 \\ & \lambda_k^2 \\ & \vdots \\ & \lambda_k^{k-1} \\ \lambda_k v_k = & \lambda_k^k \end{array}\right) = 0$$

This means that

$$\sum_{i=1}^{k} \alpha_i v_i = 0 \text{ and } \sum_{i=1}^{k} \lambda_i \alpha_i v_i = 0$$

Hence,

$$\sum_{i=1}^{k-1} \alpha_i v_i + \alpha_k v_k = 0 \text{ and } -\frac{1}{\lambda_k} \sum_{i=1}^{k-1} \lambda_i \alpha_i v_i = \alpha_k v_k$$

The Homogeneous Case

This gives that

$$\sum_{i=1}^{k-1}(1-\frac{\lambda_i}{\lambda_k})\alpha_i v_i = 0 \enspace .$$

The Homogeneous Case

This gives that

$$\sum_{i=1}^{k-1}(1-\frac{\lambda_i}{\lambda_k})\alpha_i v_i = 0 \enspace .$$

This is a contradiction as the v_i's are linearly independent because of induction hypothesis.

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$c_0\underbrace{n\lambda_i^n}_{T[n]} + c_1\underbrace{(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + c_k\underbrace{(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0 n\lambda_i^{n}}_{T[n]} + c_1\underbrace{(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + c_k\underbrace{(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda-\lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0 n\lambda_i^n}_{T[n]} + \underbrace{c_1(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + \underbrace{c_k(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0 n\lambda_i^n}_{T[n]} + \underbrace{c_1(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + \underbrace{c_k(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0 n\lambda_i^n}_{T[n]} + \underbrace{c_1(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + \underbrace{c_k(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$\underbrace{c_0 n\lambda_i^{n}}_{T[n]} + \underbrace{c_1(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + \underbrace{c_k(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

What happens if the roots are not all distinct?

Suppose we have a root λ_i with multiplicity (Vielfachheit) at least 2. Then not only is λ_i^n a solution to the recurrence but also $n\lambda_i^n$.

To see this consider the polynomial

$$P(\lambda)\lambda^{n-k} = c_0\lambda^n + c_1\lambda^{n-1} + c_2\lambda^{n-2} + \cdots + c_k\lambda^{n-k}$$

Since λ_i is a root we can write this as $Q(\lambda)(\lambda - \lambda_i)^2$. Calculating the derivative gives a polynomial that still has root λ_i.

This means

$$c_0 n\lambda_i^{n-1} + c_1(n-1)\lambda_i^{n-2} + \cdots + c_k(n-k)\lambda_i^{n-k-1} = 0$$

Hence,

$$c_0 \underbrace{n\lambda_i^n}_{T[n]} + c_1 \underbrace{(n-1)\lambda_i^{n-1}}_{T[n-1]} + \cdots + c_k \underbrace{(n-k)\lambda_i^{n-k}}_{T[n-k]} = 0$$

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell\lambda_i^n$ is a solution for $\ell \in 0,\ldots,j-1$.

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell\lambda_i^n$ is a solution for $\ell \in 0,\ldots,j-1$.

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell\lambda_i^n$ is a solution for $\ell \in 0,\dots,j-1$.

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell \lambda_i^n$ is a solution for $\ell \in 0,\ldots,j-1$.

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell\lambda_i^n$ is a solution for $\ell \in 0,\dots,j-1$.

The Homogeneous Case

Suppose λ_i has multiplicity j. We know that

$$c_0 n\lambda_i^n + c_1(n-1)\lambda_i^{n-1} + \cdots + c_k(n-k)\lambda_i^{n-k} = 0$$

(after taking the derivative; multiplying with λ; plugging in λ_i)

Doing this again gives

$$c_0 n^2\lambda_i^n + c_1(n-1)^2\lambda_i^{n-1} + \cdots + c_k(n-k)^2\lambda_i^{n-k} = 0$$

We can continue $j-1$ times.

Hence, $n^\ell\lambda_i^n$ is a solution for $\ell \in 0,\dots,j-1$.

The Homogeneous Case

Lemma 6

Let $P[\lambda]$ denote the characteristic polynomial to the recurrence

$$c_0T[n] + c_1T[n-1] + \cdots + c_kT[n-k] = 0$$

Let λ_i, $i = 1, \dots, m$ be the (complex) roots of $P[\lambda]$ with multiplicities ℓ_i. Then the general solution to the recurrence is given by

$$T[n] = \sum_{i=1}^{m} \sum_{j=0}^{\ell_i - 1} \alpha_{ij} \cdot (n^j \lambda_i^n) \ .$$

The full proof is omitted. We have only shown that any choice of α_{ij}'s is a solution to the recurrence.

Example: Fibonacci Sequence

$$
\begin{aligned}
T[0] &= 0\\
T[1] &= 1\\
T[n] &= T[n-1] + T[n-2] \text{ for } n \geq 2
\end{aligned}
$$

The characteristic polynomial is

$$\lambda^2 - \lambda - 1$$

Finding the roots, gives

$$\lambda_{1/2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1}{2}\left(1 \pm \sqrt{5}\right)$$

Example: Fibonacci Sequence

$$\begin{aligned} T[0] &= 0 \\ T[1] &= 1 \\ T[n] &= T[n-1] + T[n-2] \text{ for } n \geq 2 \end{aligned}$$

The characteristic polynomial is

$$\lambda^2 - \lambda - 1$$

Finding the roots, gives

$$\lambda_{1/2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1}{2}\left(1 \pm \sqrt{5}\right)$$

Example: Fibonacci Sequence

$$\begin{aligned} T[0] &= 0 \\ T[1] &= 1 \\ T[n] &= T[n-1] + T[n-2] \text{ for } n \geq 2 \end{aligned}$$

The characteristic polynomial is

$$\lambda^2 - \lambda - 1$$

Finding the roots, gives

$$\lambda_{1/2} = \frac{1}{2} \pm \sqrt{\frac{1}{4} + 1} = \frac{1}{2}\left(1 \pm \sqrt{5}\right)$$

Example: Fibonacci Sequence

Hence, the solution is of the form

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

Example: Fibonacci Sequence

Hence, the solution is of the form

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$T[0]=0$ gives $\alpha+\beta=0$.

Example: Fibonacci Sequence

Hence, the solution is of the form

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$T[0]=0$ gives $\alpha+\beta=0$.

$T[1]=1$ gives

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)+\beta\left(\frac{1-\sqrt{5}}{2}\right)=1$$

Example: Fibonacci Sequence

Hence, the solution is of the form

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)^n+\beta\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$T[0]=0$ gives $\alpha+\beta=0$.

$T[1]=1$ gives

$$\alpha\left(\frac{1+\sqrt{5}}{2}\right)+\beta\left(\frac{1-\sqrt{5}}{2}\right)=1\Longrightarrow\alpha-\beta=\frac{2}{\sqrt{5}}$$

Example: Fibonacci Sequence

Hence, the solution is

$$\frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

The Inhomogeneous Case

Consider the recurrence relation:

$$c_0T(n) + c_1T(n-1) + c_2T(n-2) + \cdots + c_kT(n-k) = f(n)$$

with $f(n) \neq 0$.

While we have a fairly general technique for solving homogeneous, linear recurrence relations the inhomogeneous case is different.

The Inhomogeneous Case

The general solution of the recurrence relation is

$$T(n) = T_h(n) + T_p(n) \ ,$$

where T_h is any solution to the homogeneous equation, and T_p is one particular solution to the inhomogeneous equation.

There is no general method to find a particular solution.

The Inhomogeneous Case

The general solution of the recurrence relation is

$$T(n) = T_h(n) + T_p(n) \ ,$$

where T_h is any solution to the homogeneous equation, and T_p is one particular solution to the inhomogeneous equation.

There is no general method to find a particular solution.

The Inhomogeneous Case

Example:

$$T[n] = T[n-1] + 1 \qquad T[0] = 1$$

Then,

$$T[n-1] = T[n-2] + 1 \qquad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n-1] = T[n-1] - T[n-2] \qquad (n \geq 2)$$

or

$$T[n] = 2T[n-1] - T[n-2] \qquad (n \geq 2)$$

I get a completely determined recurrence if I add $T[0] = 1$ and $T[1] = 2$.

The Inhomogeneous Case

Example:

$$T[n] = T[n-1] + 1 \qquad T[0] = 1$$

Then,

$$T[n-1] = T[n-2] + 1 \qquad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n-1] = T[n-1] - T[n-2] \qquad (n \geq 2)$$

or

$$T[n] = 2T[n-1] - T[n-2] \qquad (n \geq 2)$$

I get a completely determined recurrence if I add $T[0] = 1$ and $T[1] = 2$.

The Inhomogeneous Case

Example:

$$T[n] = T[n-1] + 1 \qquad T[0] = 1$$

Then,

$$T[n-1] = T[n-2] + 1 \qquad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n-1] = T[n-1] - T[n-2] \qquad (n \geq 2)$$

or

$$T[n] = 2T[n-1] - T[n-2] \qquad (n \geq 2)$$

I get a completely determined recurrence if I add $T[0] = 1$ and $T[1] = 2$.

The Inhomogeneous Case

Example:

$$T[n] = T[n-1] + 1 \qquad T[0] = 1$$

Then,

$$T[n-1] = T[n-2] + 1 \qquad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n-1] = T[n-1] - T[n-2] \qquad (n \geq 2)$$

or

$$T[n] = 2T[n-1] - T[n-2] \qquad (n \geq 2)$$

I get a completely determined recurrence if I add $T[0] = 1$ and $T[1] = 2$.

The Inhomogeneous Case

Example:

$$T[n] = T[n-1] + 1 \qquad T[0] = 1$$

Then,

$$T[n-1] = T[n-2] + 1 \qquad (n \geq 2)$$

Subtracting the first from the second equation gives,

$$T[n] - T[n-1] = T[n-1] - T[n-2] \qquad (n \geq 2)$$

or

$$T[n] = 2T[n-1] - T[n-2] \qquad (n \geq 2)$$

I get a completely determined recurrence if I add $T[0] = 1$ and $T[1] = 2$.

The Inhomogeneous Case

Example: Characteristic polynomial:

$$\lambda^2 - 2\lambda + 1 = 0$$

The Inhomogeneous Case

Example: Characteristic polynomial:

$$\underbrace{\lambda^2 - 2\lambda + 1}_{(\lambda-1)^2} = 0$$

The Inhomogeneous Case

Example: Characteristic polynomial:

$$\underbrace{\lambda^2 - 2\lambda + 1}_{(\lambda-1)^2} = 0$$

Then the solution is of the form

$$T[n] = \alpha 1^n + \beta n 1^n = \alpha + \beta n$$

The Inhomogeneous Case

Example: Characteristic polynomial:

$$\underbrace{\lambda^2 - 2\lambda + 1}_{(\lambda-1)^2} = 0$$

Then the solution is of the form

$$T[n] = \alpha 1^n + \beta n 1^n = \alpha + \beta n$$

$T[0] = 1$ gives $\alpha = 1$.

The Inhomogeneous Case

Example: Characteristic polynomial:

$$\underbrace{\lambda^2 - 2\lambda + 1}_{(\lambda-1)^2} = 0$$

Then the solution is of the form

$$T[n] = \alpha 1^n + \beta n 1^n = \alpha + \beta n$$

$T[0] = 1$ gives $\alpha = 1$.

$T[1] = 2$ gives $1 + \beta = 2 \Longrightarrow \beta = 1$.

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

The Inhomogeneous Case

If $f(n)$ is a polynomial of degree r this method can be applied $r+1$ times to obtain a homogeneous equation:

$$T[n] = T[n-1] + n^2$$

Shift:

$$T[n-1] = T[n-2] + (n-1)^2 = T[n-2] + n^2 - 2n + 1$$

Difference:

$$T[n] - T[n-1] = T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

Shift:

$$T[n-1] = 2T[n-2] - T[n-3] + 2(n-1) - 1$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

Shift:

$$\begin{aligned} T[n-1] &= 2T[n-2] - T[n-3] + 2(n-1) - 1 \\ &= 2T[n-2] - T[n-3] + 2n - 3 \end{aligned}$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

Shift:

$$\begin{aligned} T[n-1] &= 2T[n-2] - T[n-3] + 2(n-1) - 1 \\ &= 2T[n-2] - T[n-3] + 2n - 3 \end{aligned}$$

Difference:

$$\begin{aligned} T[n] - T[n-1] =& 2T[n-1] - T[n-2] + 2n - 1 \\ & - 2T[n-2] + T[n-3] - 2n + 3 \end{aligned}$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

Shift:

$$\begin{aligned} T[n-1] &= 2T[n-2] - T[n-3] + 2(n-1) - 1 \\ &= 2T[n-2] - T[n-3] + 2n - 3 \end{aligned}$$

Difference:

$$\begin{aligned} T[n] - T[n-1] =& 2T[n-1] - T[n-2] + 2n - 1 \\ & - 2T[n-2] + T[n-3] - 2n + 3 \end{aligned}$$

$$T[n] = 3T[n-1] - 3T[n-2] + T[n-3] + 2$$

$$T[n] = 2T[n-1] - T[n-2] + 2n - 1$$

Shift:

$$\begin{aligned} T[n-1] &= 2T[n-2] - T[n-3] + 2(n-1) - 1 \\ &= 2T[n-2] - T[n-3] + 2n - 3 \end{aligned}$$

Difference:

$$\begin{aligned} T[n] - T[n-1] =& 2T[n-1] - T[n-2] + 2n - 1 \\ & - 2T[n-2] + T[n-3] - 2n + 3 \end{aligned}$$

$$T[n] = 3T[n-1] - 3T[n-2] + T[n-3] + 2$$

and so on...

6.4 Generating Functions

Definition 7 (Generating Function)

Let $(a_n)_{n\geq 0}$ be a sequence. The corresponding

- generating function (Erzeugendenfunktion) is

$$F(z) := \sum_{n=0}^{\infty} a_n z^n;$$

- exponential generating function (exponentielle Erzeugendenfunktion) is

$$F(z) = \sum_{n\geq 0} \frac{a_n}{n!} z^n.$$

6.4 Generating Functions

Definition 7 (Generating Function)

Let $(a_n)_{n\geq 0}$ be a sequence. The corresponding

- generating function (Erzeugendenfunktion) is
$$F(z) := \sum_{n=0}^{\infty} a_n z^n;$$
- exponential generating function (exponentielle Erzeugendenfunktion) is
$$F(z) = \sum_{n\geq 0} \frac{a_n}{n!} z^n.$$

6.4 Generating Functions

Example 8

1. The generating function of the sequence $(1, 0, 0, \ldots)$ is
$$F(z) = 1 \,.$$

2. The generating function of the sequence $(1, 1, 1, \ldots)$ is
$$F(z) = \frac{1}{1-z} \,.$$

6.4 Generating Functions

Example 8

1. The generating function of the sequence $(1, 0, 0, \ldots)$ is

$$F(z) = 1 \, .$$

2. The generating function of the sequence $(1, 1, 1, \ldots)$ is

$$F(z) = \frac{1}{1 - z} \, .$$

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty} (a_n + b_n) z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

There are two different views:

A generating function is a formal power series (formale Potenzreihe).

Then the generating function is an algebraic object.

Let $f = \sum_{n=0}^{\infty} a_n z^n$ and $g = \sum_{n=0}^{\infty} b_n z^n$.

- **Equality:** f and g are equal if $a_n = b_n$ for all n.
- **Addition:** $f + g := \sum_{n=0}^{\infty}(a_n + b_n)z^n$.
- **Multiplication:** $f \cdot g := \sum_{n=0}^{\infty} c_n z^n$ with $c = \sum_{p=0}^{n} a_p b_{n-p}$.

There are no convergence issues here.

6.4 Generating Functions

The arithmetic view:

We view a power series as a function $f : \mathbb{C} \to \mathbb{C}$.

Then, it is important to think about convergence/convergence radius etc.

6.4 Generating Functions

The arithmetic view:

We view a power series as a function $f : \mathbb{C} \to \mathbb{C}$.

Then, it is important to think about convergence/convergence radius etc.

6.4 Generating Functions

The arithmetic view:

We view a power series as a function $f : \mathbb{C} \to \mathbb{C}$.

Then, it is important to think about convergence/convergence radius etc.

6.4 Generating Functions

What does $\sum_{n=0}^{\infty} z^n = \frac{1}{1-z}$ mean in the algebraic view?

It means that the power series $1 - z$ and the power series $\sum_{n=0}^{\infty} z^n$ are invers, i.e.,

$$\big(1 - z\big) \cdot \Big(\sum_{n=0}^{\infty} z^n \Big) = 1 \ .$$

This is well-defined.

6.4 Generating Functions

What does $\sum_{n=0}^{\infty} z^n = \frac{1}{1-z}$ mean in the algebraic view?

It means that the power series $1 - z$ and the power series $\sum_{n=0}^{\infty} z^n$ are invers, i.e.,

$$\left(1 - z\right) \cdot \Big(\sum_{n=0}^{\infty} z^n \Big) = 1 \ .$$

This is well-defined.

6.4 Generating Functions

What does $\sum_{n=0}^{\infty} z^n = \frac{1}{1-z}$ mean in the algebraic view?

It means that the power series $1 - z$ and the power series $\sum_{n=0}^{\infty} z^n$ are invers, i.e.,

$$\left(1 - z\right) \cdot \left(\sum_{n=0}^{\infty} z^n \right) = 1 \ .$$

This is well-defined.

6.4 Generating Functions

Suppose we are given the generating function

$$\sum_{n=0}^{\infty} z^n = \frac{1}{1-z} \ .$$

6.4 Generating Functions

Suppose we are given the generating function

$$\sum_{n=0}^{\infty} z^n = \frac{1}{1-z} \ .$$

We can compute the derivative:

$$\sum_{n\geq 1} nz^{n-1} = \frac{1}{(1-z)^2}$$

6.4 Generating Functions

Suppose we are given the generating function

$$\sum_{n=0}^{\infty} z^n = \frac{1}{1-z} \ .$$

We can compute the derivative:

$$\underbrace{\sum_{n\geq 1} n z^{n-1}}_{\sum_{n=0}^{\infty}(n+1)z^n} = \frac{1}{(1-z)^2}$$

6.4 Generating Functions

Suppose we are given the generating function

$$\sum_{n=0}^{\infty} z^n = \frac{1}{1-z} \ .$$

We can compute the derivative:

$$\underbrace{\sum_{n\geq 1} nz^{n-1}}_{\sum_{n=0}^{\infty}(n+1)z^n} = \frac{1}{(1-z)^2}$$

Hence, the generating function of the sequence $a_n = n + 1$ is $1/(1-z)^2$.

6.4 Generating Functions

We can repeat this

6.4 Generating Functions

We can repeat this

$$\sum_{n=0}^{\infty}(n+1)z^n = \frac{1}{(1-z)^2}\ .$$

6.4 Generating Functions

We can repeat this

$$\sum_{n=0}^{\infty}(n+1)z^n = \frac{1}{(1-z)^2} \ .$$

Derivative:

$$\sum_{n\geq 1} n(n+1)z^{n-1} = \frac{2}{(1-z)^3}$$

6.4 Generating Functions

We can repeat this

$$\sum_{n=0}^{\infty}(n+1)z^n = \frac{1}{(1-z)^2}\ .$$

Derivative:

$$\underbrace{\sum_{n\ge 1} n(n+1)z^{n-1}}_{\sum_{n=0}^{\infty}(n+1)(n+2)z^n} = \frac{2}{(1-z)^3}$$

6.4 Generating Functions

We can repeat this

$$\sum_{n=0}^{\infty}(n+1)z^n = \frac{1}{(1-z)^2} \enspace .$$

Derivative:

$$\underbrace{\sum_{n\geq 1} n(n+1)z^{n-1}}_{\sum_{n=0}^{\infty}(n+1)(n+2)z^n} = \frac{2}{(1-z)^3}$$

Hence, the generating function of the sequence $a_n = (n+1)(n+2)$ is $\frac{2}{(1-z)^2}$.

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

$$\sum_{n\geq k} n(n-1)\dots(n-k+1)z^{n-k}$$

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

$$\sum_{n\ge k} n(n-1)\dots(n-k+1)z^{n-k} = \sum_{n\ge 0}(n+k)\dots(n+1)z^n$$

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

$$\begin{aligned}\sum_{n\geq k} n(n-1)\dots(n-k+1)z^{n-k} &= \sum_{n\geq 0}(n+k)\dots(n+1)z^n\\ &= \frac{k!}{(1-z)^{k+1}}\ .\end{aligned}$$

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

$$\sum_{n\geq k} n(n-1)\dots(n-k+1)z^{n-k} = \sum_{n\geq 0}(n+k)\dots(n+1)z^n$$

$$= \frac{k!}{(1-z)^{k+1}} \ .$$

Hence:

$$\sum_{n\geq 0}\binom{n+k}{k}z^n = \frac{1}{(1-z)^{k+1}} \ .$$

6.4 Generating Functions

Computing the k-th derivative of $\sum z^n$.

$$\sum_{n\geq k} n(n-1)\dots(n-k+1)z^{n-k} = \sum_{n\geq 0}(n+k)\dots(n+1)z^n$$
$$= \frac{k!}{(1-z)^{k+1}} \ .$$

Hence:

$$\sum_{n\geq 0}\binom{n+k}{k}z^n = \frac{1}{(1-z)^{k+1}} \ .$$

The generating function of the sequence $a_n = \binom{n+k}{k}$ is $\frac{1}{(1-z)^{k+1}}$.

6.4 Generating Functions

$$\sum_{n\geq 0} nz^n = \sum_{n\geq 0} (n+1)z^n - \sum_{n\geq 0} z^n$$

6.4 Generating Functions

$$
\begin{aligned}
\sum_{n\geq 0} nz^n &= \sum_{n\geq 0} (n+1)z^n - \sum_{n\geq 0} z^n \\
&= \frac{1}{(1-z)^2} - \frac{1}{1-z}
\end{aligned}
$$

6.4 Generating Functions

$$\begin{aligned}\sum_{n\geq 0} nz^n &= \sum_{n\geq 0}(n+1)z^n - \sum_{n\geq 0} z^n \\ &= \frac{1}{(1-z)^2} - \frac{1}{1-z} \\ &= \frac{z}{(1-z)^2}\end{aligned}$$

6.4 Generating Functions

$$
\begin{aligned}
\sum_{n\geq 0} nz^n &= \sum_{n\geq 0}(n+1)z^n - \sum_{n\geq 0} z^n \\
&= \frac{1}{(1-z)^2} - \frac{1}{1-z} \\
&= \frac{z}{(1-z)^2}
\end{aligned}
$$

6.4 Generating Functions

$$\begin{aligned}\sum_{n\geq 0} nz^n &= \sum_{n\geq 0}(n+1)z^n - \sum_{n\geq 0} z^n \\ &= \frac{1}{(1-z)^2} - \frac{1}{1-z} \\ &= \frac{z}{(1-z)^2}\end{aligned}$$

The generating function of the sequence $a_n = n$ is $\frac{z}{(1-z)^2}$.

6.4 Generating Functions

We know

$$\sum_{n\geq 0} y^n = \frac{1}{1-y}$$

Hence,

$$\sum_{n\geq 0} a^n z^n = \frac{1}{1-az}$$

The generating function of the sequence $f_n = a^n$ is $\frac{1}{1-az}$.

6.4 Generating Functions

We know

$$\sum_{n\ge 0} y^n = \frac{1}{1-y}$$

Hence,

$$\sum_{n\ge 0} a^n z^n = \frac{1}{1-az}$$

The generating function of the sequence $f_n = a^n$ is $\frac{1}{1-az}$.

6.4 Generating Functions

We know

$$\sum_{n\geq 0} y^n = \frac{1}{1-y}$$

Hence,

$$\sum_{n\geq 0} a^n z^n = \frac{1}{1-az}$$

The generating function of the sequence $f_n = a^n$ is $\frac{1}{1-az}$.

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$A(z)$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$A(z) = \sum_{n \geq 0} a_n z^n$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$
\begin{aligned}
A(z) &= \sum_{n\geq 0} a_n z^n \\
&= a_0 + \sum_{n\geq 1} (a_{n-1} + 1) z^n
\end{aligned}
$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} (a_{n-1} + 1) z^n \\ &= 1 + z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} z^n \end{aligned}$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$\begin{aligned}
A(z) &= \sum_{n\geq 0} a_n z^n \\
&= a_0 + \sum_{n\geq 1} (a_{n-1} + 1) z^n \\
&= 1 + z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} z^n \\
&= z \sum_{n\geq 0} a_n z^n + \sum_{n\geq 0} z^n
\end{aligned}$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} (a_{n-1} + 1) z^n \\ &= 1 + z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} z^n \\ &= z \sum_{n\geq 0} a_n z^n + \sum_{n\geq 0} z^n \\ &= zA(z) + \sum_{n\geq 0} z^n \end{aligned}$$

6.4 Generating Functions

Suppose we have again the recurrence $a_n = a_{n-1} + 1$ for $n \geq 1$ and $a_0 = 1$.

$$\begin{aligned}
A(z) &= \sum_{n\geq 0} a_n z^n \\
&= a_0 + \sum_{n\geq 1} (a_{n-1} + 1) z^n \\
&= 1 + z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} z^n \\
&= z \sum_{n\geq 0} a_n z^n + \sum_{n\geq 0} z^n \\
&= zA(z) + \sum_{n\geq 0} z^n \\
&= zA(z) + \frac{1}{1-z}
\end{aligned}$$

6.4 Generating Functions

Solving for $A(z)$ gives

6.4 Generating Functions

Solving for $A(z)$ gives

$$A(z) = \frac{1}{(1-z)^2}$$

6.4 Generating Functions

Solving for $A(z)$ gives

$$\sum_{n\geq 0} a_n z^n = A(z) = \frac{1}{(1-z)^2}$$

6.4 Generating Functions

Solving for $A(z)$ gives

$$\sum_{n\geq 0} a_n z^n = A(z) = \frac{1}{(1-z)^2} = \sum_{n\geq 0} (n+1) z^n$$

6.4 Generating Functions

Solving for $A(z)$ gives

$$\sum_{n\ge 0} a_n z^n = A(z) = \frac{1}{(1-z)^2} = \sum_{n\ge 0} (n+1) z^n$$

Hence, $a_n = n + 1$.

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	generating function
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
1	$\frac{1}{1-z}$
$n+1$	$\frac{1}{(1-z)^2}$
$\binom{n+k}{n}$	$\frac{1}{(1-z)^{k+1}}$
n	$\frac{z}{(1-z)^2}$
a^n	$\frac{1}{1-az}$
n^2	$\frac{z(1+z)}{(1-z)^3}$
$\frac{1}{n!}$	$\frac{z(1+z)}{(1-z)^3}$

Some Generating Functions

n-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k);\ \ 0$ otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\dfrac{F(z)}{1-z}$
nf_n	$z\dfrac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

n-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k);\quad$ 0 otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\frac{F(z)}{1-z}$
nf_n	$z\frac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

$\boldsymbol{n}$-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k);\quad 0$ otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\dfrac{F(z)}{1-z}$
nf_n	$z\dfrac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

n-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
f_{n-k} $(n \geq k)$; 0 otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\frac{F(z)}{1-z}$
nf_n	$z\frac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

n-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
f_{n-k} $(n \geq k)$; 0 otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\dfrac{F(z)}{1-z}$
nf_n	$z\dfrac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

$\boldsymbol{n}$-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k); \quad 0$ otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\dfrac{F(z)}{1-z}$
nf_n	$z\dfrac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

$\boldsymbol{n}$-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k); \quad 0$ otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\dfrac{F(z)}{1-z}$
nf_n	$z\dfrac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Some Generating Functions

$\boldsymbol{n}$-th sequence element	**generating function**
cf_n	cF
$f_n + g_n$	$F + G$
$\sum_{i=0}^{n} f_i g_{n-i}$	$F \cdot G$
$f_{n-k} \quad (n \geq k); \quad 0$ otw.	$z^k F$
$\sum_{i=0}^{n} f_i$	$\frac{F(z)}{1-z}$
nf_n	$z\frac{\mathrm{d}F(z)}{\mathrm{d}z}$
$c^n f_n$	$F(cz)$

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.
4. Solving for $A(z)$ gives an equation of the form $A(z) = f(z)$, where hopefully $f(z)$ is a simple function.

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.
4. Solving for $A(z)$ gives an equation of the form $A(z) = f(z)$, where hopefully $f(z)$ is a simple function.
5. Write $f(z)$ as a formal power series.
   Techniques:

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.
4. Solving for $A(z)$ gives an equation of the form $A(z) = f(z)$, where hopefully $f(z)$ is a simple function.
5. Write $f(z)$ as a formal power series.
   Techniques:
   - partial fraction decomposition (Partialbruchzerlegung)

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\ge 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.
4. Solving for $A(z)$ gives an equation of the form $A(z) = f(z)$, where hopefully $f(z)$ is a simple function.
5. Write $f(z)$ as a formal power series.
   Techniques:
   - partial fraction decomposition (Partialbruchzerlegung)
   - lookup in tables

Solving Recursions with Generating Functions

1. Set $A(z) = \sum_{n\geq 0} a_n z^n$.
2. Transform the right hand side so that boundary condition and recurrence relation can be plugged in.
3. Do further transformations so that the infinite sums on the right hand side can be replaced by $A(z)$.
4. Solving for $A(z)$ gives an equation of the form $A(z) = f(z)$, where hopefully $f(z)$ is a simple function.
5. Write $f(z)$ as a formal power series.
   Techniques:
   - partial fraction decomposition (Partialbruchzerlegung)
   - lookup in tables
6. The coefficients of the resulting power series are the a_n.

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n \geq 0} a_n z^n$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

2. Transform right hand side so that recurrence can be plugged in:

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

2. Transform right hand side so that recurrence can be plugged in:

$$A(z) = a_0 + \sum_{n\geq 1} a_n z^n$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

2. Transform right hand side so that recurrence can be plugged in:

$$A(z) = a_0 + \sum_{n\geq 1} a_n z^n$$

2. Plug in:

Example: $a_n = 2a_{n-1}, a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

2. Transform right hand side so that recurrence can be plugged in:

$$A(z) = a_0 + \sum_{n\geq 1} a_n z^n$$

2. Plug in:

$$A(z) = 1 + \sum_{n\geq 1} (2a_{n-1}) z^n$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

Example: $a_n = 2a_{n-1}, a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

Example: $a_n = 2a_{n-1}$, $a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$A(z) = 1 + \sum_{n\geq 1}(2a_{n-1})z^n$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$
\begin{aligned}
A(z) &= 1 + \sum_{n\geq 1}(2a_{n-1})z^n \\
&= 1 + 2z\sum_{n\geq 1} a_{n-1}z^{n-1}
\end{aligned}
$$

Example: $a_n = 2a_{n-1}$, $a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$\begin{aligned} A(z) &= 1 + \sum_{n\geq 1} (2a_{n-1})z^n \\ &= 1 + 2z \sum_{n\geq 1} a_{n-1}z^{n-1} \\ &= 1 + 2z \sum_{n\geq 0} a_n z^n \end{aligned}$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$\begin{aligned} A(z) &= 1 + \sum_{n\geq 1}(2a_{n-1})z^n \\ &= 1 + 2z\sum_{n\geq 1}a_{n-1}z^{n-1} \\ &= 1 + 2z\sum_{n\geq 0}a_n z^n \\ &= 1 + 2z\cdot A(z) \end{aligned}$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$\begin{aligned} A(z) &= 1 + \sum_{n\geq 1}(2a_{n-1})z^n \\ &= 1 + 2z\sum_{n\geq 1} a_{n-1}z^{n-1} \\ &= 1 + 2z\sum_{n\geq 0} a_n z^n \\ &= 1 + 2z \cdot A(z) \end{aligned}$$

4. Solve for $A(z)$.

Example: $a_n = 2a_{n-1}, a_0 = 1$

3. Transform right hand side so that infinite sums can be replaced by $A(z)$ or by simple function.

$$\begin{aligned} A(z) &= 1 + \sum_{n\geq 1}(2a_{n-1})z^n \\ &= 1 + 2z\sum_{n\geq 1}a_{n-1}z^{n-1} \\ &= 1 + 2z\sum_{n\geq 0}a_n z^n \\ &= 1 + 2z\cdot A(z) \end{aligned}$$

4. Solve for $A(z)$.

$$A(z) = \frac{1}{1-2z}$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

5. Rewrite $f(n)$ as a power series:

$$A(z) = \frac{1}{1-2z}$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

5. Rewrite $f(n)$ as a power series:

$$\sum_{n\geq 0} a_n z^n = A(z) = \frac{1}{1-2z}$$

Example: $a_n = 2a_{n-1}, a_0 = 1$

5. Rewrite $f(n)$ as a power series:

$$\sum_{n\geq 0} a_n z^n = A(z) = \frac{1}{1-2z} = \sum_{n\geq 0} 2^n z^n$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n \geq 0} a_n z^n$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

1. Set up generating function:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$A(z) = \sum_{n\geq 0} a_n z^n$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} a_n z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} a_n z^n \\ &= 1 + \sum_{n\geq 1} (3a_{n-1} + n) z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} a_n z^n \\ &= 1 + \sum_{n\geq 1} (3a_{n-1} + n) z^n \\ &= 1 + 3z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} n z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$\begin{aligned} A(z) &= \sum_{n\geq 0} a_n z^n \\ &= a_0 + \sum_{n\geq 1} a_n z^n \\ &= 1 + \sum_{n\geq 1} (3a_{n-1} + n) z^n \\ &= 1 + 3z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} n z^n \\ &= 1 + 3z \sum_{n\geq 0} a_n z^n + \sum_{n\geq 0} n z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

2./3. Transform right hand side:

$$\begin{aligned}
A(z) &= \sum_{n\geq 0} a_n z^n \\
&= a_0 + \sum_{n\geq 1} a_n z^n \\
&= 1 + \sum_{n\geq 1} (3a_{n-1} + n) z^n \\
&= 1 + 3z \sum_{n\geq 1} a_{n-1} z^{n-1} + \sum_{n\geq 1} n z^n \\
&= 1 + 3z \sum_{n\geq 0} a_n z^n + \sum_{n\geq 0} n z^n \\
&= 1 + 3zA(z) + \frac{z}{(1-z)^2}
\end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

4. Solve for $A(z)$:

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

4. Solve for $A(z)$:

$$A(z) = 1 + 3zA(z) + \frac{z}{(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

4. Solve for $A(z)$:

$$A(z) = 1 + 3zA(z) + \frac{z}{(1-z)^2}$$

gives

$$A(z) = \frac{(1-z)^2 + z}{(1-3z)(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

4. Solve for $A(z)$:

$$A(z) = 1 + 3zA(z) + \frac{z}{(1-z)^2}$$

gives

$$A(z) = \frac{(1-z)^2 + z}{(1-3z)(1-z)^2} = \frac{z^2 - z + 1}{(1-3z)(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

   We use partial fraction decomposition:

Example: $a_n = 3a_{n-1} + n, a_0 = 1$

5. Write $f(z)$ as a formal power series:

   We use partial fraction decomposition:

$$\frac{z^2 - z + 1}{(1-3z)(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n, a_0 = 1$

5. Write $f(z)$ as a formal power series:

   We use partial fraction decomposition:

$$\frac{z^2 - z + 1}{(1-3z)(1-z)^2} \overset{!}{=} \frac{A}{1-3z} + \frac{B}{1-z} + \frac{C}{(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

   We use partial fraction decomposition:

$$\frac{z^2 - z + 1}{(1-3z)(1-z)^2} \overset{!}{=} \frac{A}{1-3z} + \frac{B}{1-z} + \frac{C}{(1-z)^2}$$

   This leads to the following conditions:

$$\begin{aligned} A + B + C &= 1 \\ 2A + 4B + 3C &= 1 \\ A + 3B &= 1 \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

We use partial fraction decomposition:

$$\frac{z^2 - z + 1}{(1-3z)(1-z)^2} \overset{!}{=} \frac{A}{1-3z} + \frac{B}{1-z} + \frac{C}{(1-z)^2}$$

This leads to the following conditions:

$$\begin{aligned} A + B + C &= 1 \\ 2A + 4B + 3C &= 1 \\ A + 3B &= 1 \end{aligned}$$

which gives

$$A = \frac{7}{4} \quad B = -\frac{1}{4} \quad C = -\frac{1}{2}$$

Example: $a_n = 3a_{n-1} + n, a_0 = 1$

5. Write $f(z)$ as a formal power series:

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

$$A(z) = \frac{7}{4} \cdot \frac{1}{1-3z} - \frac{1}{4} \cdot \frac{1}{1-z} - \frac{1}{2} \cdot \frac{1}{(1-z)^2}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

$$\begin{aligned} A(z) &= \frac{7}{4} \cdot \frac{1}{1-3z} - \frac{1}{4} \cdot \frac{1}{1-z} - \frac{1}{2} \cdot \frac{1}{(1-z)^2} \\ &= \frac{7}{4} \cdot \sum_{n\ge 0} 3^n z^n - \frac{1}{4} \cdot \sum_{n\ge 0} z^n - \frac{1}{2} \cdot \sum_{n\ge 0} (n+1) z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n,\ a_0 = 1$

5. Write $f(z)$ as a formal power series:

$$\begin{aligned} A(z) &= \frac{7}{4}\cdot\frac{1}{1-3z} - \frac{1}{4}\cdot\frac{1}{1-z} - \frac{1}{2}\cdot\frac{1}{(1-z)^2} \\ &= \frac{7}{4}\cdot\sum_{n\geq 0} 3^n z^n - \frac{1}{4}\cdot\sum_{n\geq 0} z^n - \frac{1}{2}\cdot\sum_{n\geq 0}(n+1)z^n \\ &= \sum_{n\geq 0}\Big(\frac{7}{4}\cdot 3^n - \frac{1}{4} - \frac{1}{2}(n+1)\Big)z^n \end{aligned}$$

Example: $a_n = 3a_{n-1} + n$, $a_0 = 1$

5. Write $f(z)$ as a formal power series:

$$
\begin{aligned}
A(z) &= \frac{7}{4} \cdot \frac{1}{1-3z} - \frac{1}{4} \cdot \frac{1}{1-z} - \frac{1}{2} \cdot \frac{1}{(1-z)^2} \\
&= \frac{7}{4} \cdot \sum_{n\geq 0} 3^n z^n - \frac{1}{4} \cdot \sum_{n\geq 0} z^n - \frac{1}{2} \cdot \sum_{n\geq 0} (n+1) z^n \\
&= \sum_{n\geq 0} \left(\frac{7}{4} \cdot 3^n - \frac{1}{4} - \frac{1}{2}(n+1) \right) z^n
\end{aligned}
$$

6. This means $a_n = \frac{7}{4}3^n - \frac{1}{2}n - \frac{3}{4}$.

6.5 Transformation of the Recurrence

Example 9

$$
\begin{aligned}
f_0 &= 1\\
f_1 &= 2\\
f_n &= f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,.
\end{aligned}
$$

6.5 Transformation of the Recurrence

Example 9

$$f_0 = 1$$
$$f_1 = 2$$
$$f_n = f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,.$$

Define

$$g_n := \log f_n \,.$$

6.5 Transformation of the Recurrence

Example 9

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 2 \\ f_n &= f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,. \end{aligned}$$

Define

$$g_n := \log f_n \,.$$

Then

$$g_n = g_{n-1} + g_{n-2} \text{ for } n \geq 2$$

6.5 Transformation of the Recurrence

Example 9

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 2 \\ f_n &= f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2\,. \end{aligned}$$

Define

$$g_n := \log f_n\,.$$

Then

$$\begin{aligned} g_n &= g_{n-1} + g_{n-2} \text{ for } n \geq 2 \\ g_1 &= \log 2 = 1,\ g_0 = 0 \text{ (fÃČÅŠr } \log = \log_2\text{)} \end{aligned}$$

6.5 Transformation of the Recurrence

Example 9

$$
\begin{aligned}
f_0 &= 1\\
f_1 &= 2\\
f_n &= f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,.
\end{aligned}
$$

Define

$$g_n := \log f_n \,.$$

Then

$$
\begin{aligned}
g_n &= g_{n-1} + g_{n-2} \text{ for } n \geq 2\\
g_1 &= \log 2 = 1,\ g_0 = 0 \text{ (fÃČÅŠr } \log = \log_2\text{)}\\
g_n &= F_n \text{ (n-th Fibonacci number)}
\end{aligned}
$$

6.5 Transformation of the Recurrence

Example 9

$$\begin{aligned} f_0 &= 1 \\ f_1 &= 2 \\ f_n &= f_{n-1} \cdot f_{n-2} \text{ for } n \geq 2 \,. \end{aligned}$$

Define

$$g_n := \log f_n \,.$$

Then

$$\begin{aligned} g_n &= g_{n-1} + g_{n-2} \text{ for } n \geq 2 \\ g_1 &= \log 2 = 1,\; g_0 = 0 \text{ (fÃČÅŠr } \log = \log_2 \text{)} \\ g_n &= F_n \text{ (n-th Fibonacci number)} \\ f_n &= 2^{F_n} \end{aligned}$$

6.5 Transformation of the Recurrence

Example 10

$$f_1 = 1$$
$$f_n = 3f_{\frac{n}{2}} + n;\ \text{for } n = 2^k\ ;$$

6.5 Transformation of the Recurrence

Example 10

$$\begin{aligned} f_1 &= 1 \\ f_n &= 3f_{\frac{n}{2}} + n;\ \text{for } n = 2^k\ ; \end{aligned}$$

Define

$$g_k := f_{2^k}\ .$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$
\begin{aligned}
g_0 &= 1 \\
g_k &= 3g_{k-1} + 2^k,\ k \geq 1
\end{aligned}
$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$\begin{aligned} g_0 &= 1 \\ g_k &= 3g_{k-1} + 2^k,\ k \geq 1 \end{aligned}$$

We get,

$$g_k = 3^{k+1} - 2^{k+1},$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$\begin{aligned} g_0 &= 1 \\ g_k &= 3g_{k-1} + 2^k,\ k \geq 1 \end{aligned}$$

We get,

$$\begin{aligned} g_k &= 3^{k+1} - 2^{k+1},\ \text{hence} \\ f_n &= 3 \cdot 3^k - 2 \cdot 2^k \end{aligned}$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$\begin{aligned} g_0 &= 1 \\ g_k &= 3g_{k-1} + 2^k,\ k \geq 1 \end{aligned}$$

We get,

$$\begin{aligned} g_k &= 3^{k+1} - 2^{k+1},\ \text{hence} \\ f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\ &= 3(2^{\log 3})^k - 2 \cdot 2^k \end{aligned}$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$\begin{aligned} g_0 &= 1 \\ g_k &= 3g_{k-1} + 2^k,\ k \geq 1 \end{aligned}$$

We get,

$$\begin{aligned} g_k &= 3^{k+1} - 2^{k+1},\ \text{hence} \\ f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\ &= 3(2^{\log 3})^k - 2 \cdot 2^k \\ &= 3(2^k)^{\log 3} - 2 \cdot 2^k \end{aligned}$$

6.5 Transformation of the Recurrence

Example 10

Then:

$$\begin{aligned} g_0 &= 1 \\ g_k &= 3g_{k-1} + 2^k,\ k \geq 1 \end{aligned}$$

We get,

$$\begin{aligned} g_k &= 3^{k+1} - 2^{k+1},\ \text{hence} \\ f_n &= 3 \cdot 3^k - 2 \cdot 2^k \\ &= 3(2^{\log 3})^k - 2 \cdot 2^k \\ &= 3(2^k)^{\log 3} - 2 \cdot 2^k \\ &= 3n^{\log 3} - 2n\,. \end{aligned}$$

Part III

Data Structures

Abstract Data Type

An abstract data type (ADT) is defined by an interface of operations or methods that can be performed and that have a defined behavior

The data types in this lecture all operate on objects that are represented by a [key, value] pair.

- The key comes from a totally ordered set, and we assume that there is an efficient comparison function.
- The value can be anything; it usually carries satellite information important for the application that uses the ADT.

Dynamic Set Operations

- $\boldsymbol{S.\,\mathrm{search}(k)}$: Returns pointer to object x from S with $\mathrm{key}[x] = k$ or null.
- $\boldsymbol{S.\,\mathrm{insert}(x)}$: Inserts object x into set S. $\mathrm{key}[x]$ must not currently exist in the data-structure.
- $\boldsymbol{S.\,\mathrm{delete}(x)}$: Given pointer to object x from S, delete x from the set.
- $\boldsymbol{S.\,\mathrm{minimum}()}$: Return pointer to object with smallest key-value in S.
- $\boldsymbol{S.\,\mathrm{maximum}()}$: Return pointer to object with largest key-value in S.
- $\boldsymbol{S.\,\mathrm{successor}(x)}$: Return pointer to the next larger element in S or null if S is maximum.
- $\boldsymbol{S.\,\mathrm{predecessor}(x)}$: Return pointer to the next smaller element in S or null if S is minimum.

Dynamic Set Operations

- **S. union(S'):** Sets $S := S \cup S'$. The set S' is destroyed.
- **S. merge(S'):** Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- **S. split(k, S'):**
  $S := \{x \in S \mid \mathrm{key}[x] \le k\}$, $S' := \{x \in S \mid \mathrm{key}[x] > k\}$.
- **S. concatenate(S'):** $S := S \cup S'$.
  Requires $S.\mathrm{maximum}() \le S'.\mathrm{minimum}()$.

- **S. decrease-key(x, k):** Replace $\mathrm{key}[x]$ by $k \le \mathrm{key}[x]$.

Examples of ADTs

Stack:

- ***S*.push(*x*)**: Insert an element.
- ***S*.pop()**: Return the element from S that was inserted most recently; delete it from S.
- ***S*.empty()**: Tell if S contains any object.

Queue:

- ***S*.enqueue(*x*)**: Insert an element.
- ***S*.dequeue()**: Return the element that is longest in the structure; delete it from S.
- ***S*.empty()**: Tell if S contains any object.

Priority-Queue:

- ***S*.insert(*x*)**: Insert an element.
- ***S*.delete-min()**: Return the element with lowest key-value; delete it from S.

7 Dictionary

Dictionary:

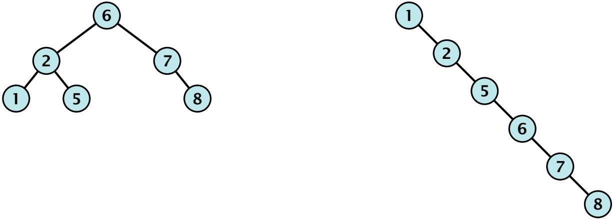
- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

7.1 Binary Search Trees

An (internal) binary search tree stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node v have a smaller key-value than $\mathrm{key}[v]$ and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:

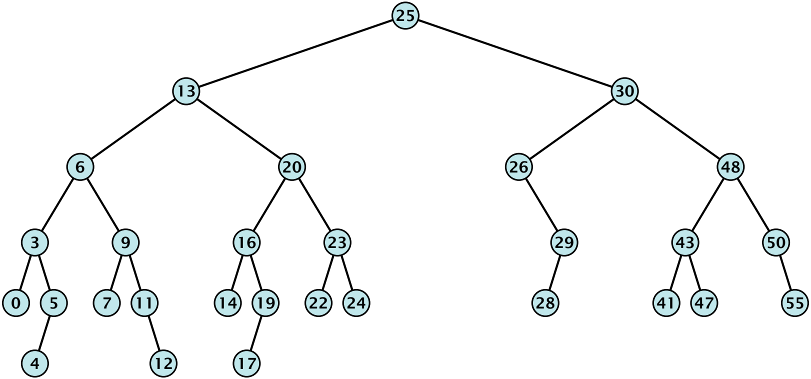


7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- $T.\,\textbf{insert}(x)$
- $T.\,\textbf{delete}(x)$
- $T.\,\textbf{search}(k)$
- $T.\,\textbf{successor}(x)$
- $T.\,\textbf{predecessor}(x)$
- $T.\,\textbf{minimum}()$
- $T.\,\textbf{maximum}()$

Binary Search Trees: Searching

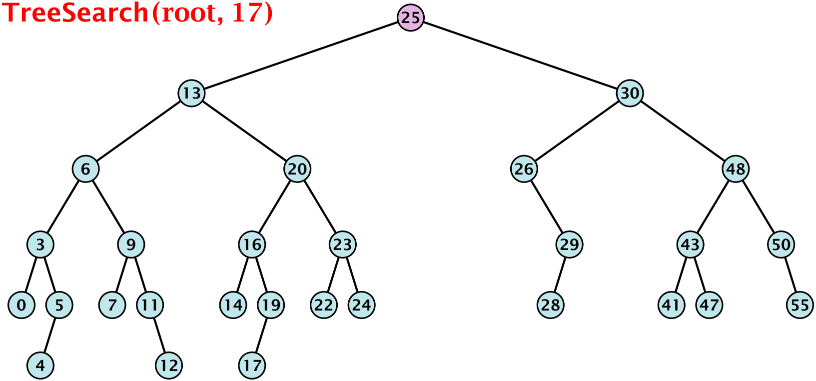


Algorithm 5 TreeSearch(x, k)

```
1: if $x$ = null or $k$ = key[$x$] return $x$
2: if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
3: else return TreeSearch(right[$x$], $k$)
```

Binary Search Trees: Searching

TreeSearch(root, 17)

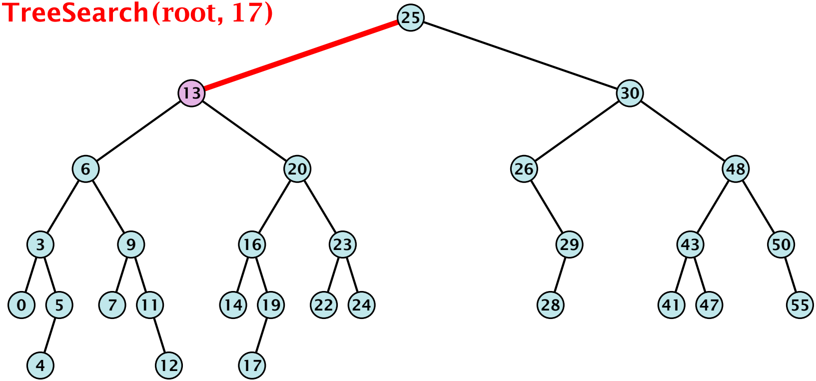


Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

Binary Search Trees: Searching

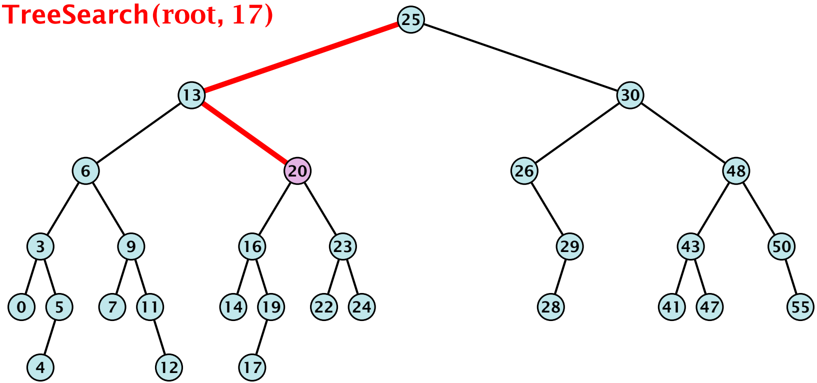
TreeSearch(root, 17)



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k$ < key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

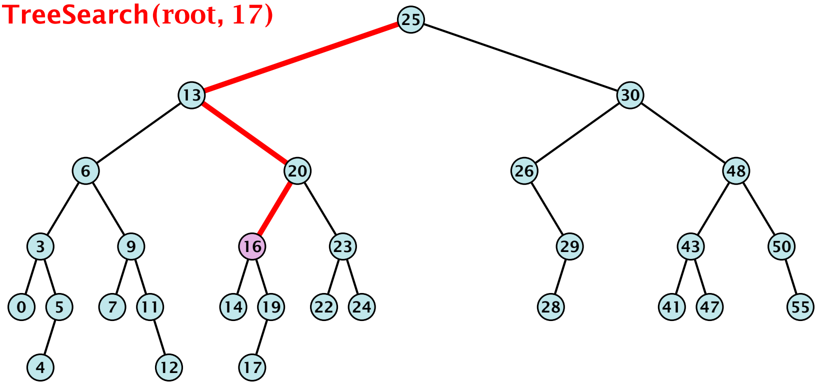
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

Binary Search Trees: Searching

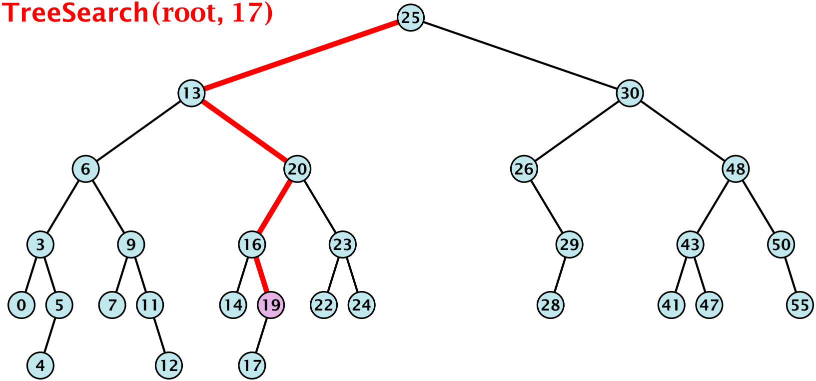


Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k$ < key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

Binary Search Trees: Searching

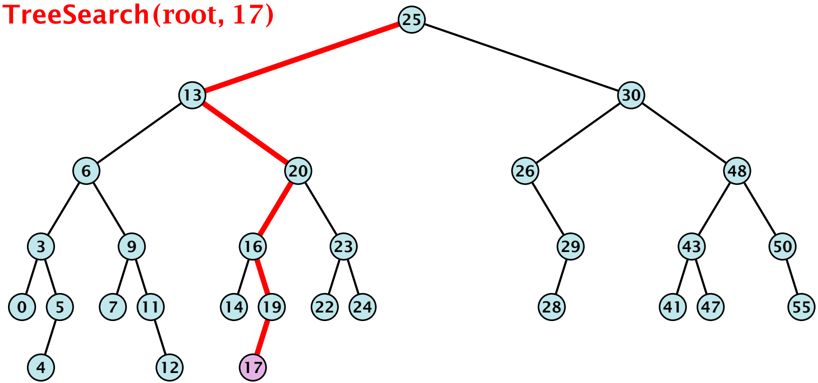
TreeSearch(root, 17)



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k$ < key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

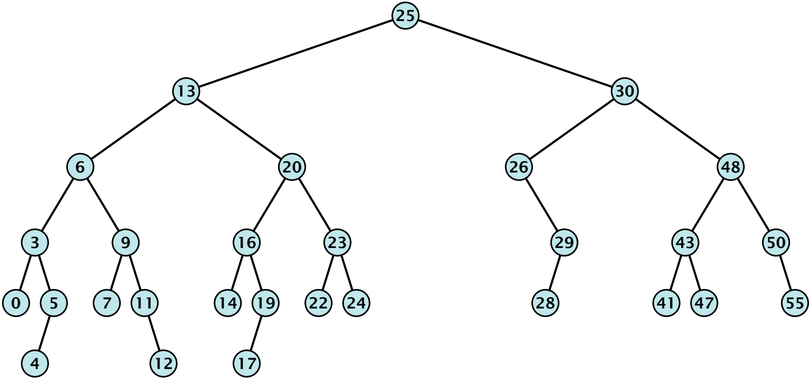
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k = \text{key}[x]$ return $x$
if $k < \text{key}[x]$ return TreeSearch$(\text{left}[x], k)$
else return TreeSearch$(\text{right}[x], k)$
```

Binary Search Trees: Searching

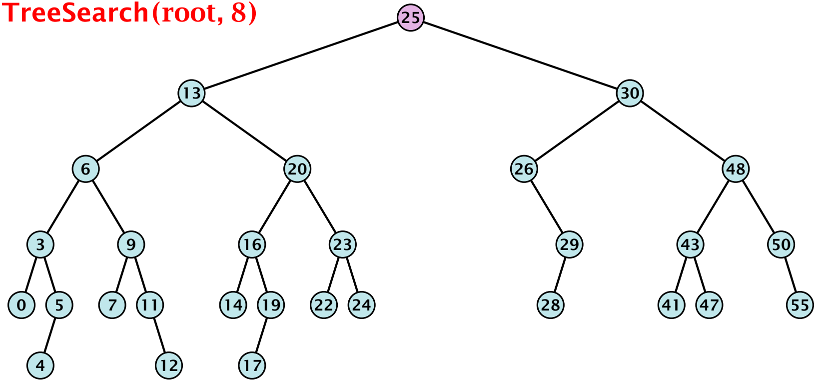


Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k = \text{key}[x]$ return $x$
if $k < \text{key}[x]$ return TreeSearch$(\text{left}[x], k)$
else return TreeSearch$(\text{right}[x], k)$
```

Binary Search Trees: Searching

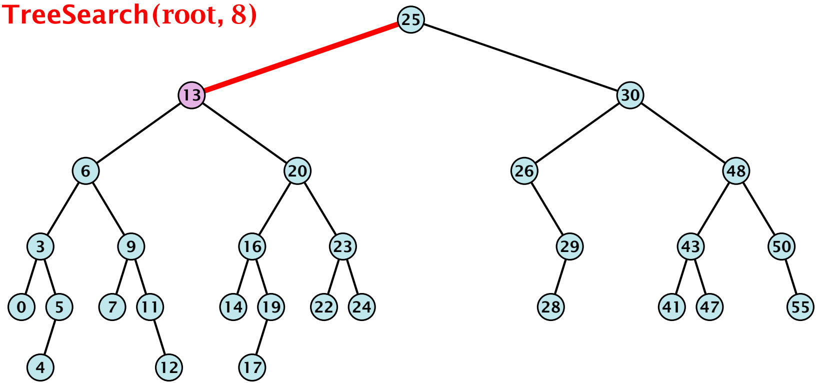
TreeSearch(root, 8)



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key$[x]$ return $x$
if $k <$ key$[x]$ return TreeSearch(left$[x]$, $k$)
else return TreeSearch(right$[x]$, $k$)
```

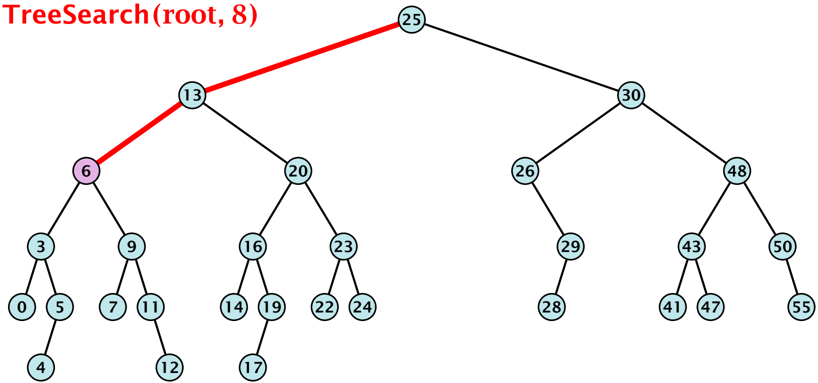
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k = \text{key}[x]$ return $x$
if $k < \text{key}[x]$ return TreeSearch(left$[x], k$)
else return TreeSearch(right$[x], k$)
```

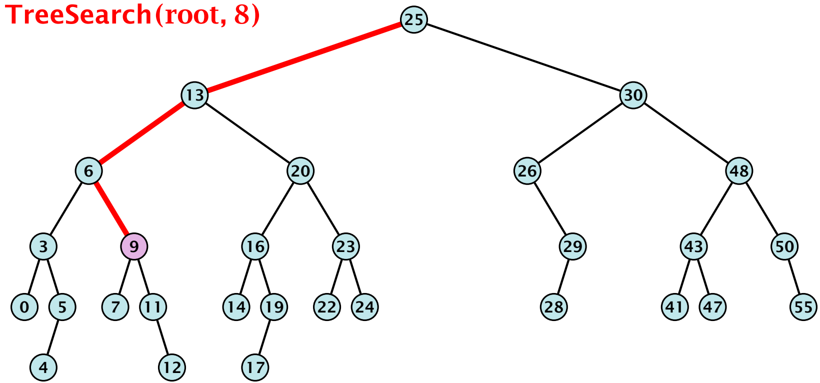
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k = \text{key}[x]$ return $x$
if $k < \text{key}[x]$ return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

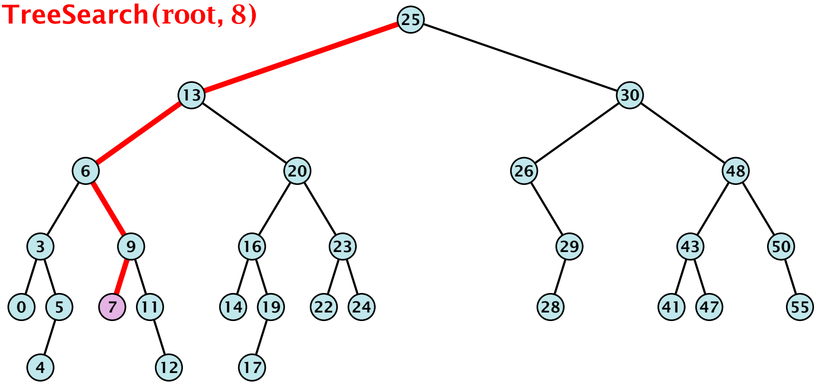
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k = \text{key}[x]$ return $x$
if $k < \text{key}[x]$ return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

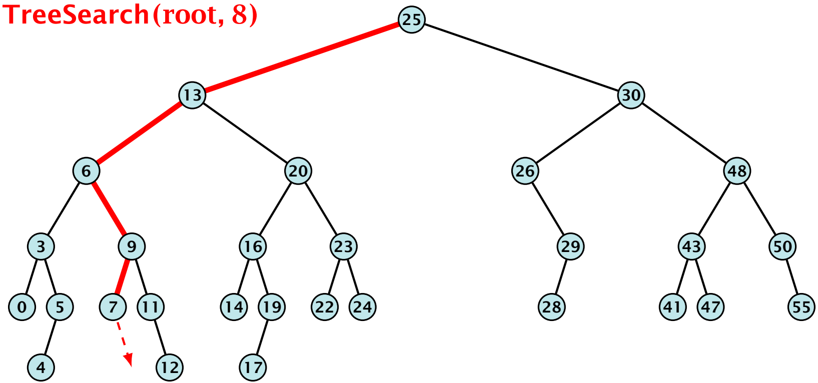
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key[$x$] return $x$
if $k$ < key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeSearch(right[$x$], $k$)
```

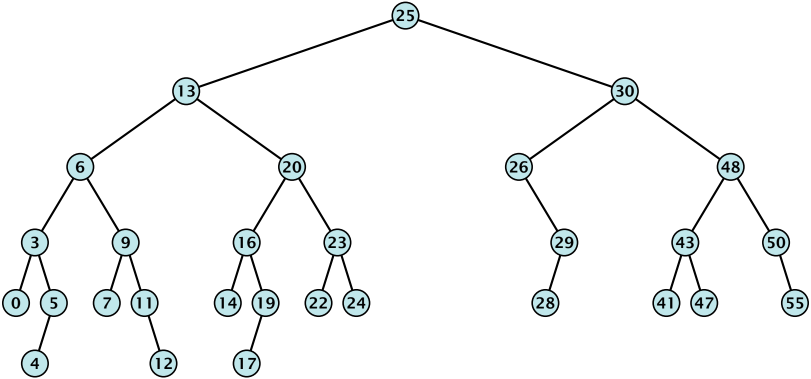
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

```
if $x$ = null or $k$ = key$[x]$ return $x$
if $k <$ key$[x]$ return TreeSearch(left$[x]$, $k$)
else return TreeSearch(right$[x]$, $k$)
```

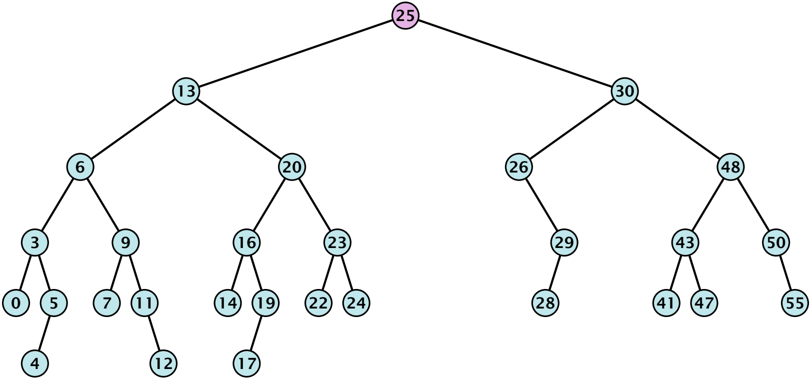
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left[$x$] = null return $x$
if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeMin(left[$x$])
```

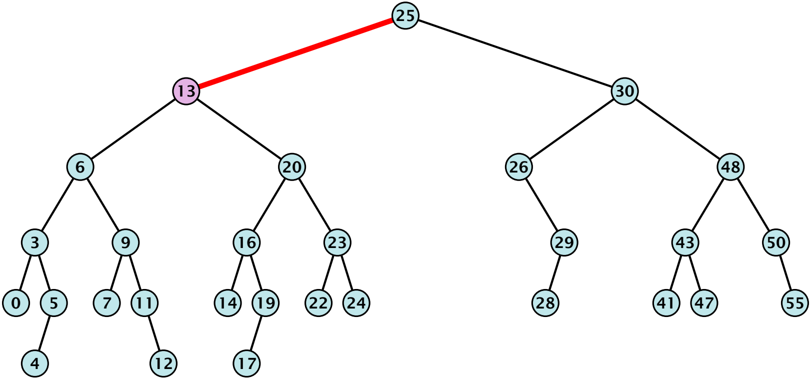
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left[$x$] = null return $x$
if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeMin(left[$x$])
```

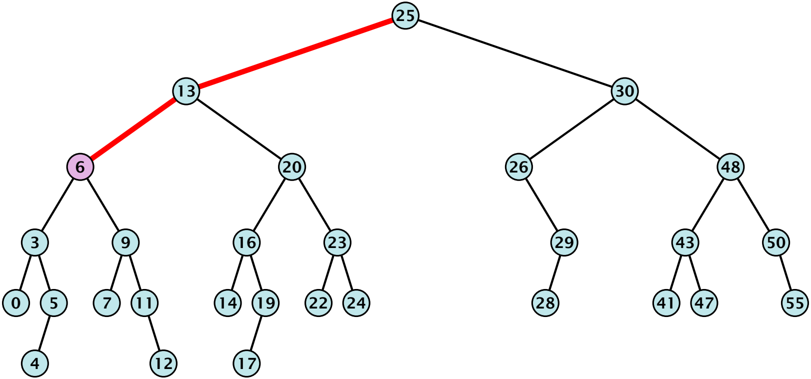
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left[$x$] = null return $x$
if $k < \text{key}[x]$ return TreeSearch(left[$x$], $k$)
else return TreeMin(left[$x$])
```

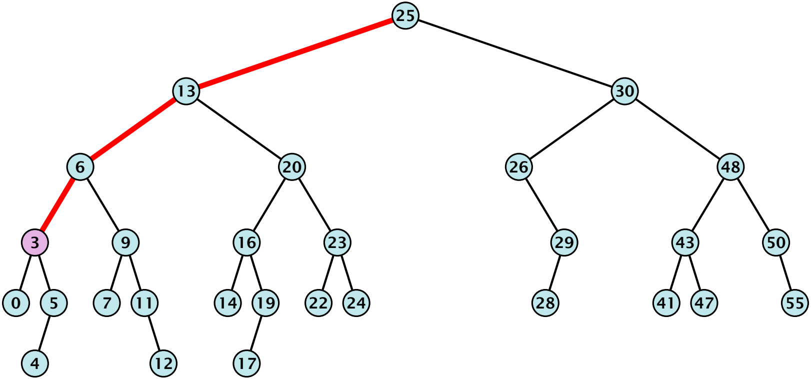
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left$[x]$ = null return $x$
if $k <$ key$[x]$ return TreeSearch(left$[x], k$)
else return TreeMin(left$[x]$)
```

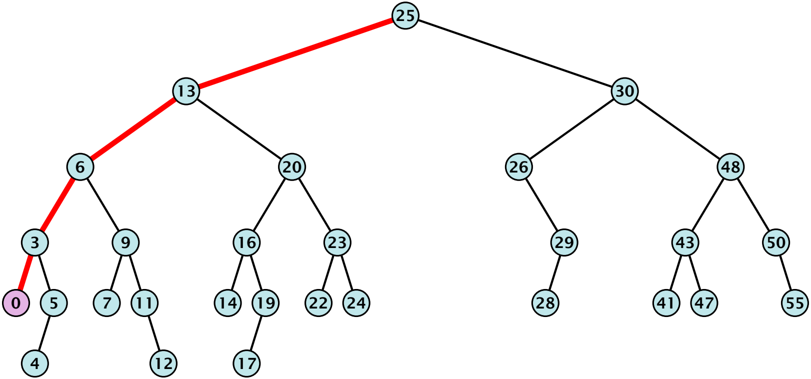
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left$[x]$ = null return $x$
if $k <$ key$[x]$ return TreeSearch(left$[x]$, $k$)
else return TreeMin(left$[x]$)
```

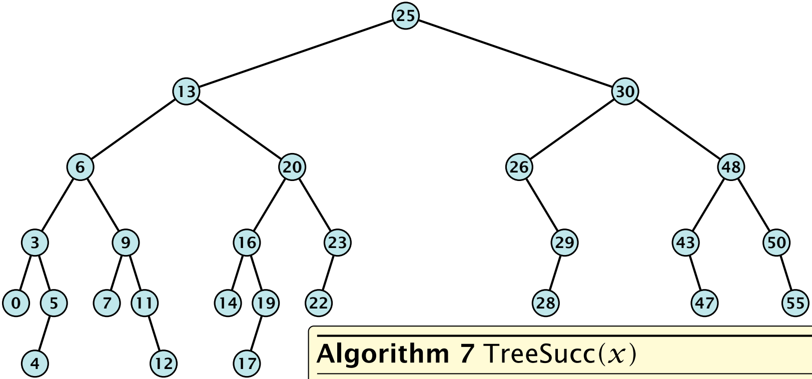
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

```
if $x$ = null or left[$x$] = null return $x$
if $k <$ key[$x$] return TreeSearch(left[$x$], $k$)
else return TreeMin(left[$x$])
```

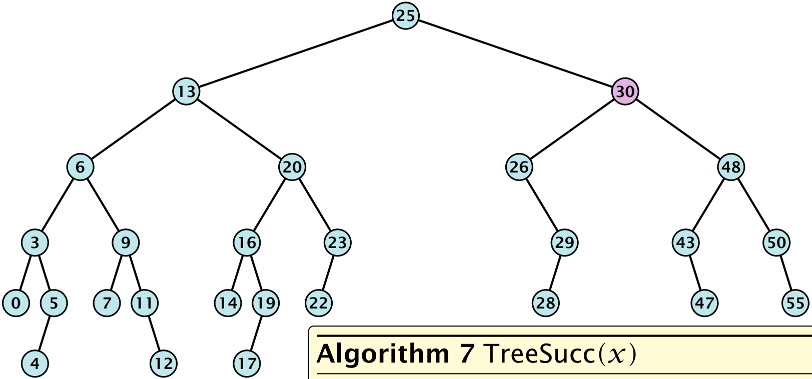
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x$ = right[$y$] do
    $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

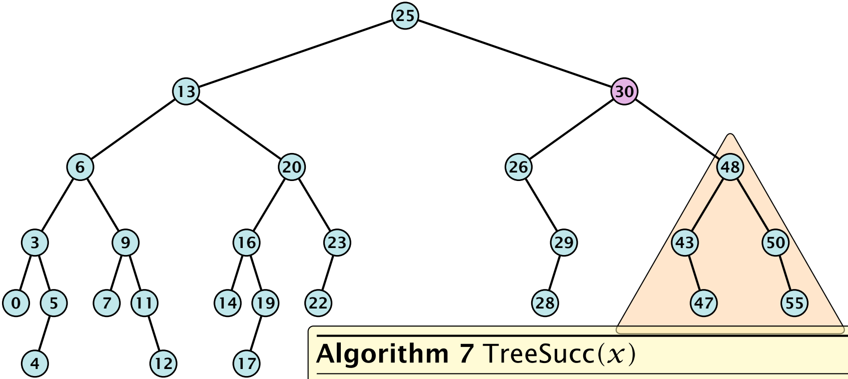
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x =$ right[$y$] do
    $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

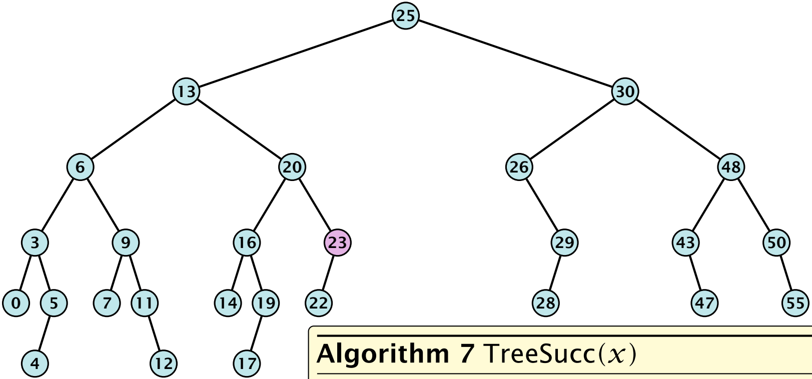
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x =$ right[$y$] do
        $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

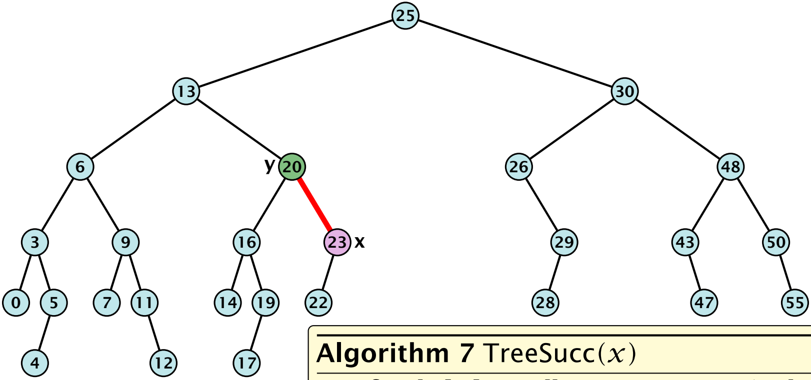
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x =$ right[$y$] do
    $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

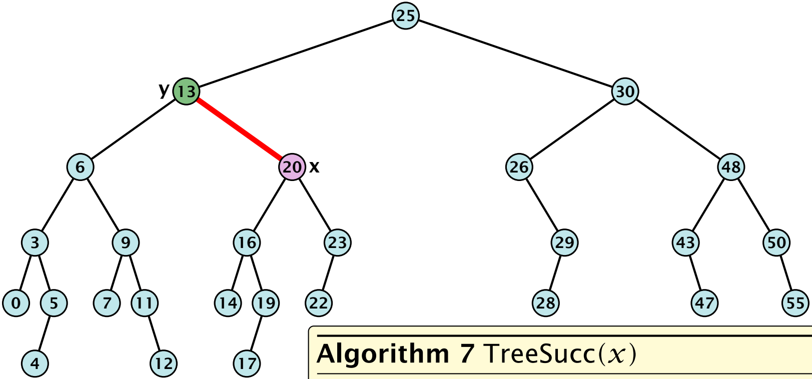
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x =$ right[$y$] do
        $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

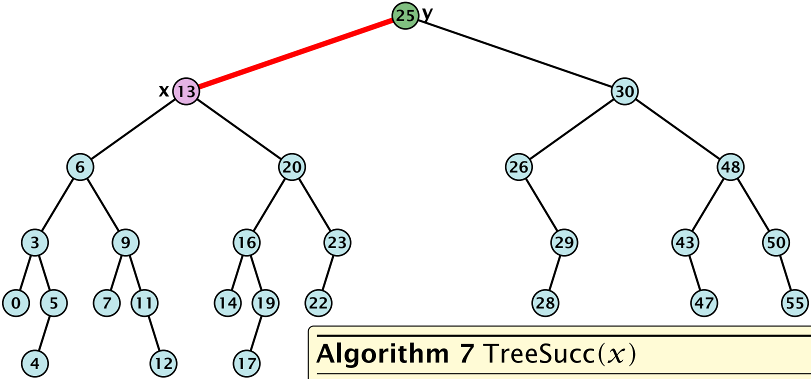
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x =$ right[$y$] do
    $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

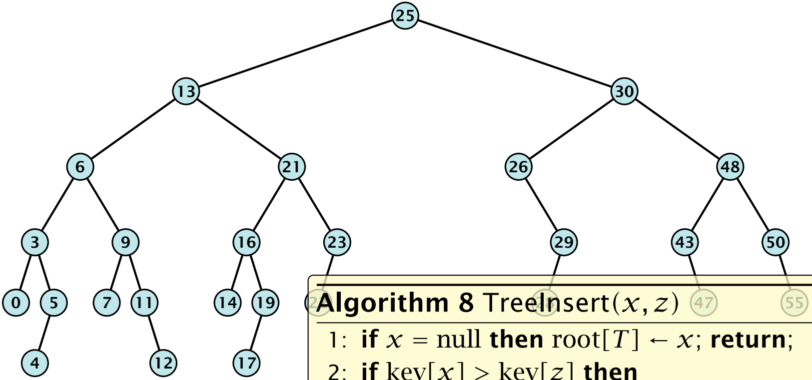
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[$x$] $\neq$ null return TreeMin(right[$x$])
$y \leftarrow$ parent[$x$]
while $y \neq$ null and $x$ = right[$y$] do
    $x \leftarrow y$; $y \leftarrow$ parent[$x$]
return $y$;
```

Binary Search Trees: Insert

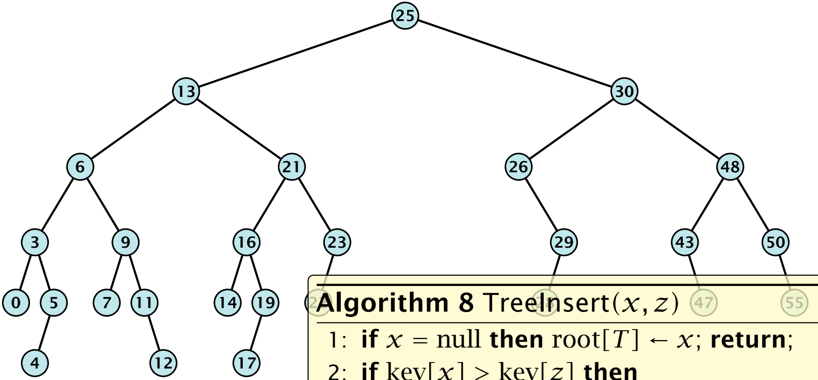


Algorithm 8 TreeInsert(x, z)

```
if $x$ = null then root[$T$] ← $x$; return;
if key[$x$] > key[$z$] then
    if left[$x$] = null then left[$x$] ← $z$;
    else TreeInsert(left[$x$], $z$);
else
    if right[$x$] = null then right[$x$] ← $z$;
    else TreeInsert(right[$x$], $z$);
return
```

Binary Search Trees: Insert

Insert element **not** in the tree.

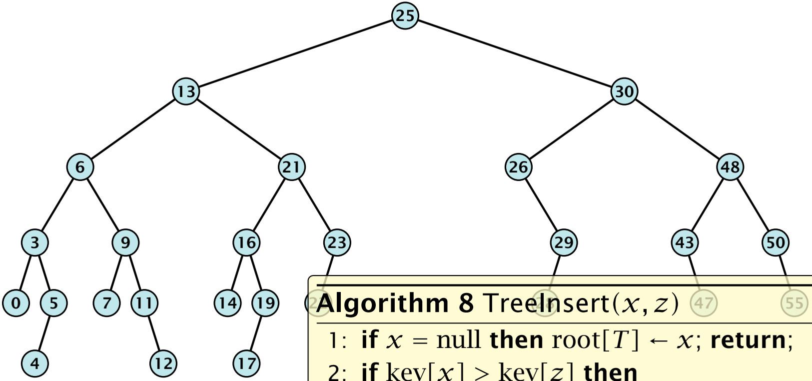


Algorithm 8 TreeInsert(x, z)

```
if $x$ = null then root[$T$] ← $x$; return;
if key[$x$] > key[$z$] then
    if left[$x$] = null then left[$x$] ← $z$;
    else TreeInsert(left[$x$], $z$);
else
    if right[$x$] = null then right[$x$] ← $z$;
    else TreeInsert(right[$x$], $z$);
return
```

Binary Search Trees: Insert

Insert element **not** in the tree.



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

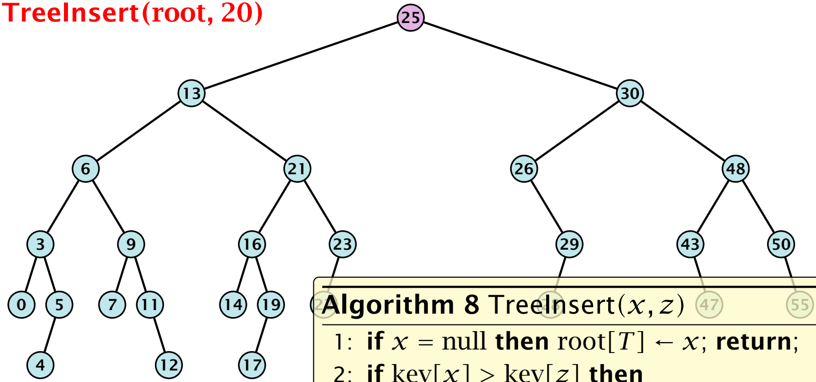
Algorithm 8 TreeInsert(x, z)

```
if $x$ = null then root$[T] \leftarrow x$; return;
if key$[x]$ > key$[z]$ then
    if left$[x]$ = null then left$[x] \leftarrow z$;
    else TreeInsert(left$[x]$, $z$);
else
    if right$[x]$ = null then right$[x] \leftarrow z$;
    else TreeInsert(right$[x]$, $z$);
return
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

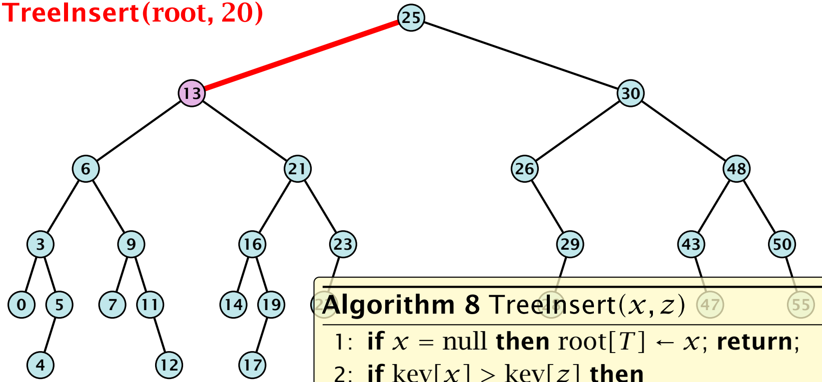
```
if $x$ = null then root$[T] \leftarrow x$; return;
if key$[x]$ > key$[z]$ then
    if left$[x]$ = null then left$[x] \leftarrow z$;
    else TreeInsert(left$[x], z$);
else
    if right$[x]$ = null then right$[x] \leftarrow z$;
    else TreeInsert(right$[x], z$);
return
```

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

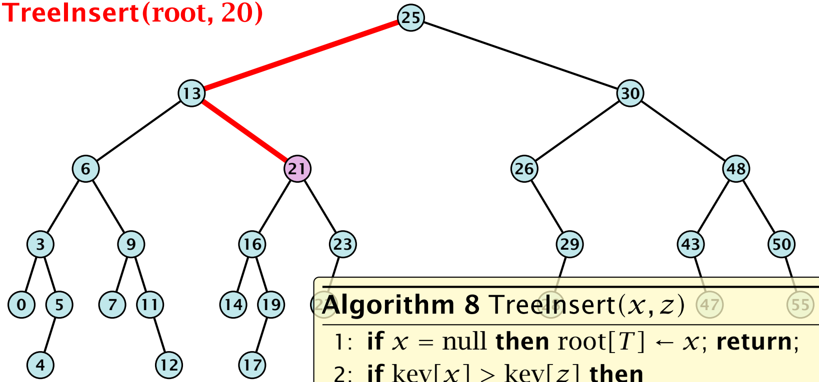
```
if $x$ = null then root$[T] \leftarrow x$; return;
if key$[x]$ > key$[z]$ then
    if left$[x]$ = null then left$[x] \leftarrow z$;
    else TreeInsert(left$[x]$, $z$);
else
    if right$[x]$ = null then right$[x] \leftarrow z$;
    else TreeInsert(right$[x]$, $z$);
return
```

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

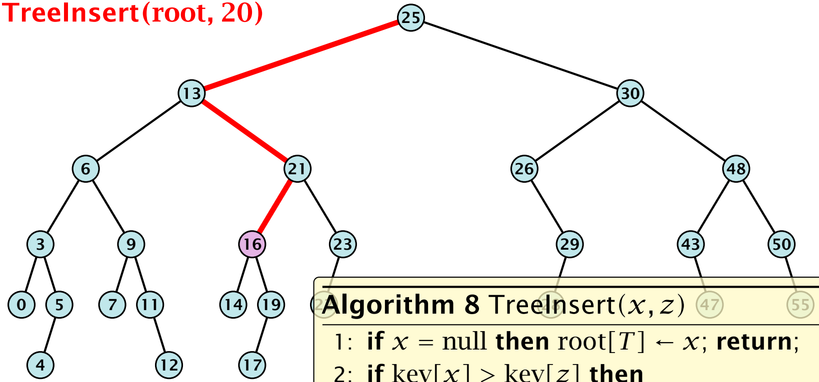
```
if $x$ = null then root$[T] \leftarrow x$; return;
if key$[x]$ > key$[z]$ then
    if left$[x]$ = null then left$[x] \leftarrow z$;
    else TreeInsert(left$[x], z$);
else
    if right$[x]$ = null then right$[x] \leftarrow z$;
    else TreeInsert(right$[x], z$);
return
```

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

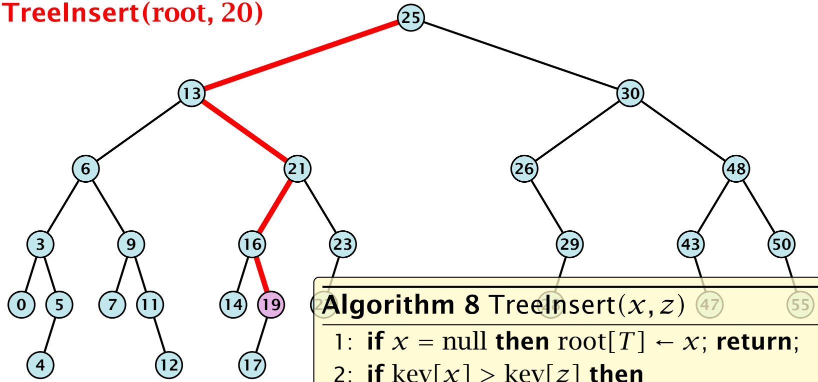
Algorithm 8 TreeInsert(x, z)

```
if $x$ = null then root[$T$] ← $x$; return;
if key[$x$] > key[$z$] then
    if left[$x$] = null then left[$x$] ← $z$;
    else TreeInsert(left[$x$], $z$);
else
    if right[$x$] = null then right[$x$] ← $z$;
    else TreeInsert(right[$x$], $z$);
return
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

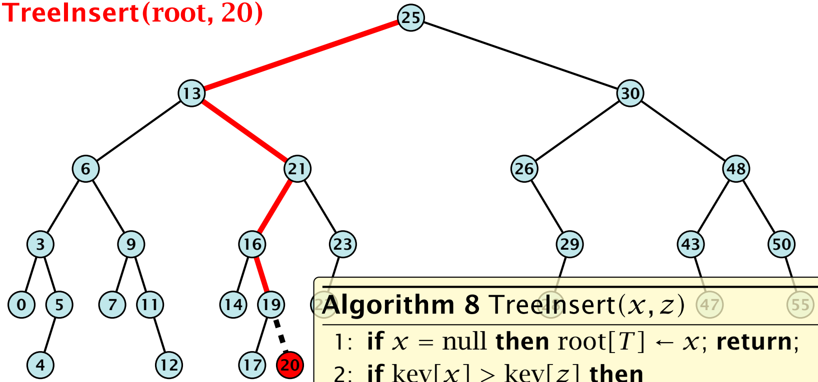
```
if $x$ = null then root[$T$] ← $x$; return;
if key[$x$] > key[$z$] then
    if left[$x$] = null then left[$x$] ← $z$;
    else TreeInsert(left[$x$], $z$);
else
    if right[$x$] = null then right[$x$] ← $z$;
    else TreeInsert(right[$x$], $z$);
return
```

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

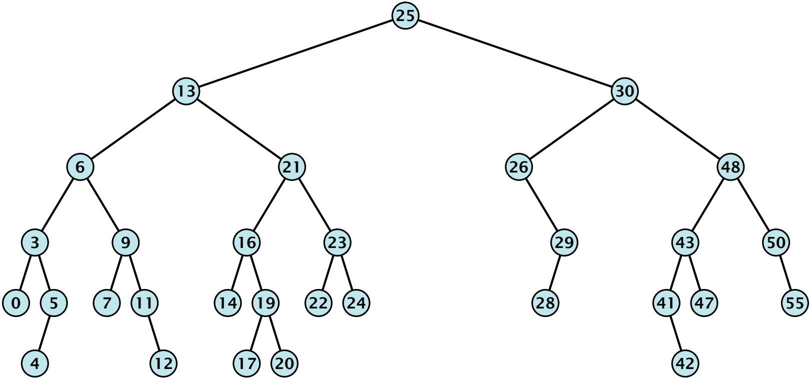


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

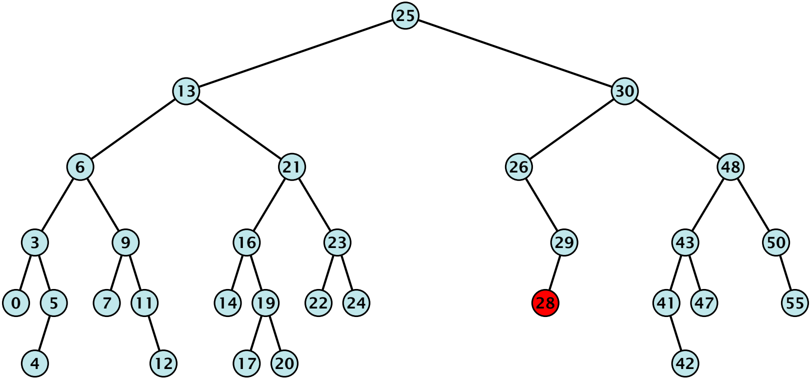
Algorithm 8 TreeInsert(x, z)

```
if $x$ = null then root$[T] \leftarrow x$; return;
if key$[x]$ > key$[z]$ then
    if left$[x]$ = null then left$[x] \leftarrow z$;
    else TreeInsert(left$[x], z$);
else
    if right$[x]$ = null then right$[x] \leftarrow z$;
    else TreeInsert(right$[x], z$);
return
```

Binary Search Trees: Delete



Binary Search Trees: Delete

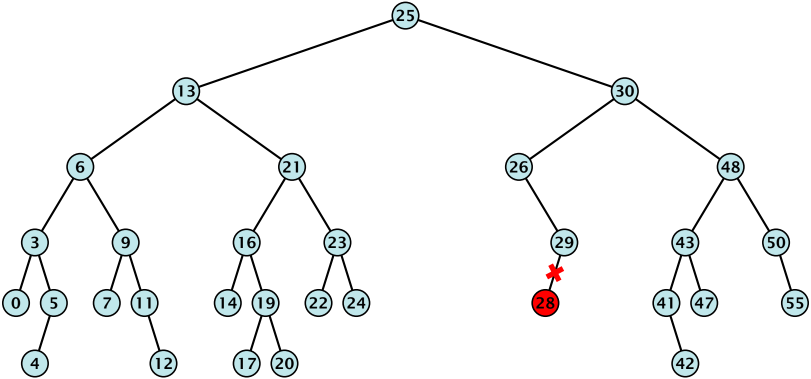


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

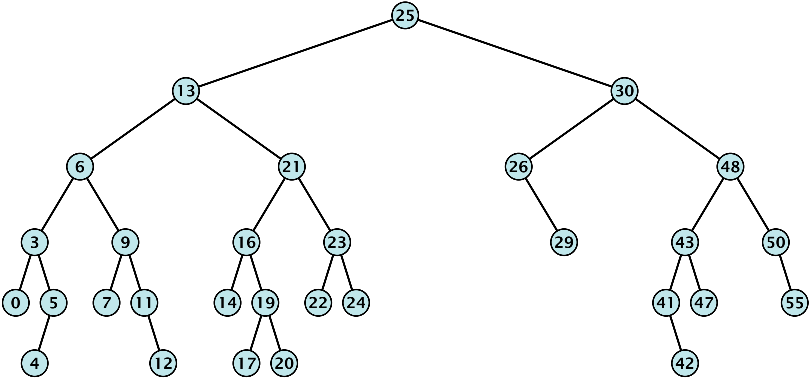


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

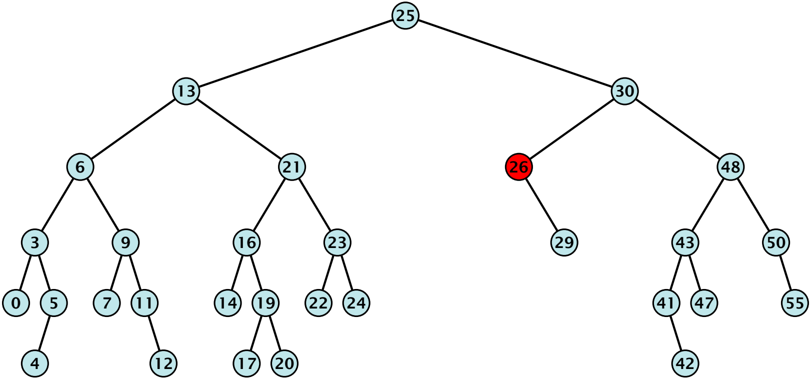


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

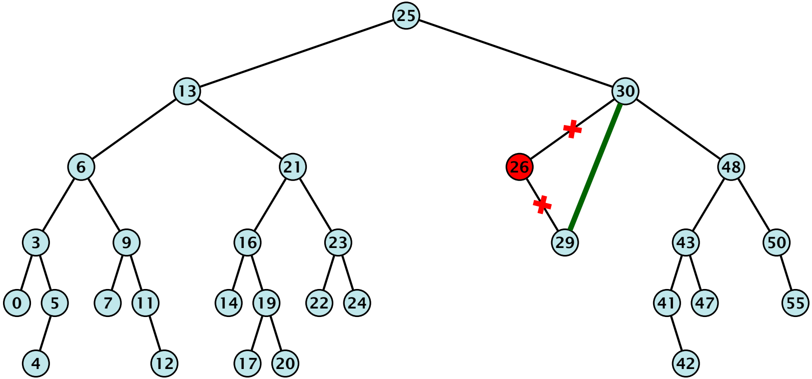


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

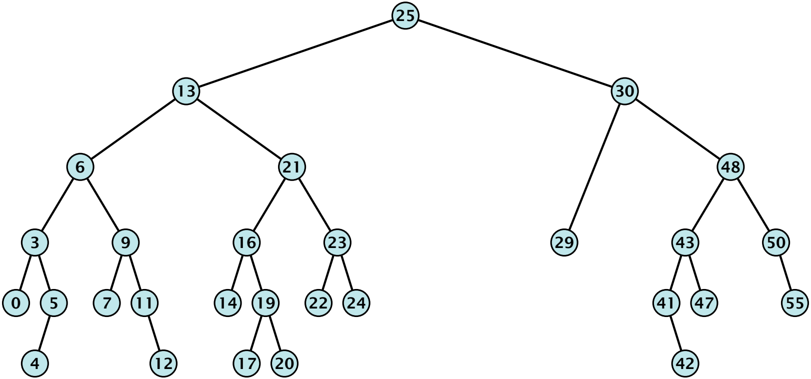


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

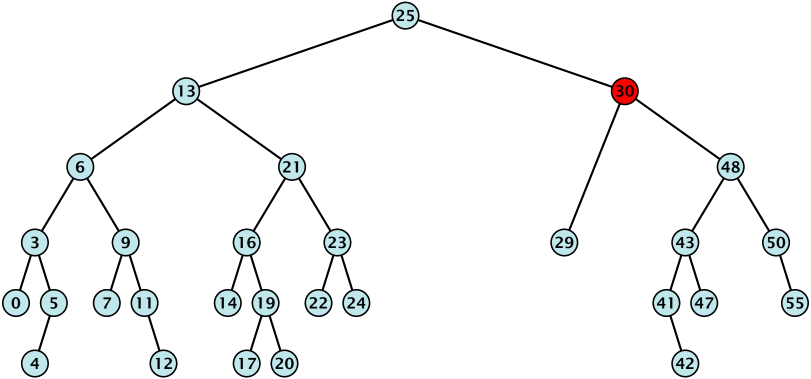


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

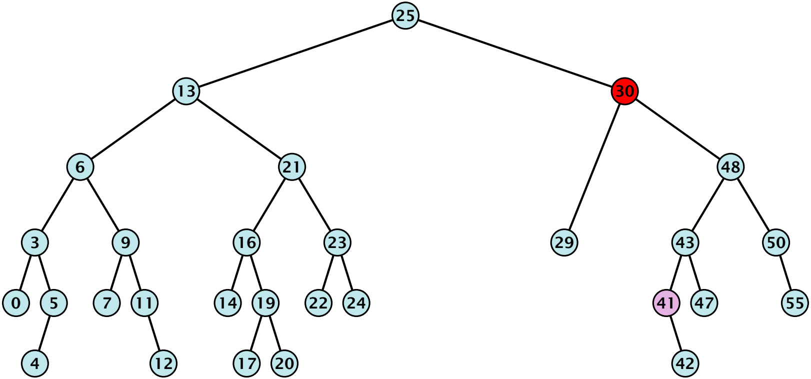


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

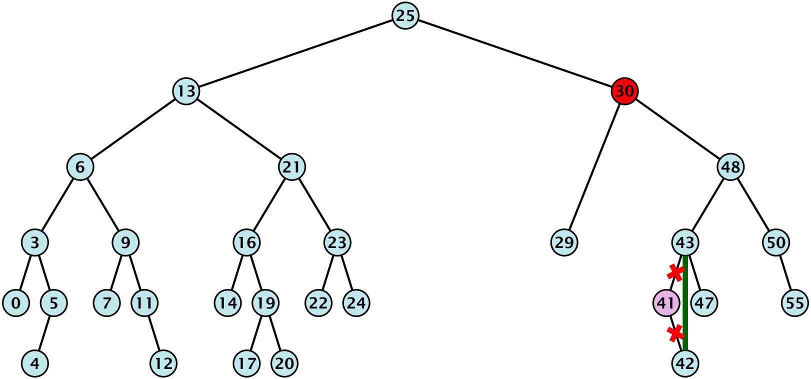


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

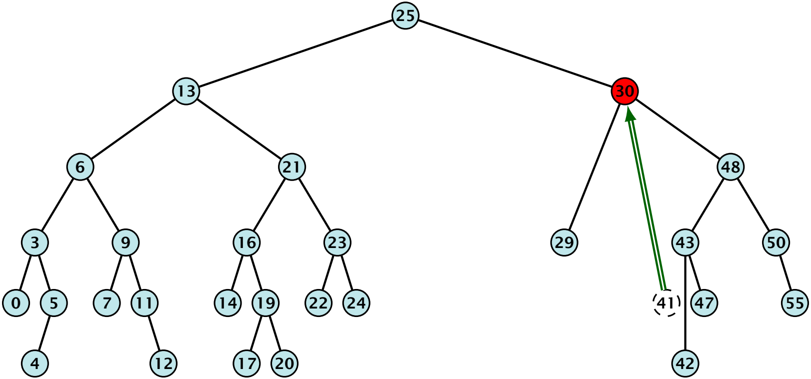


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

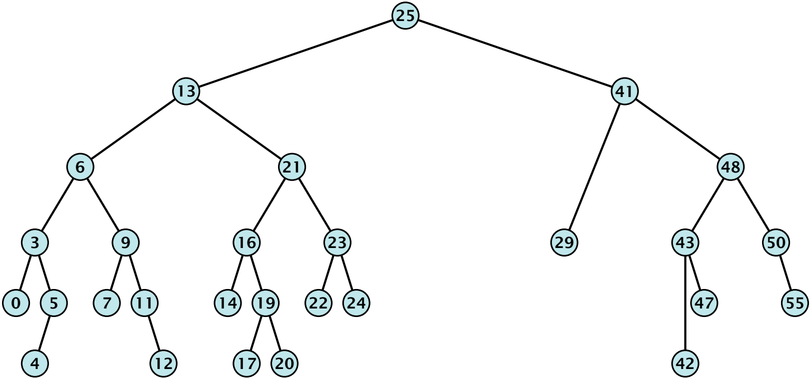


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

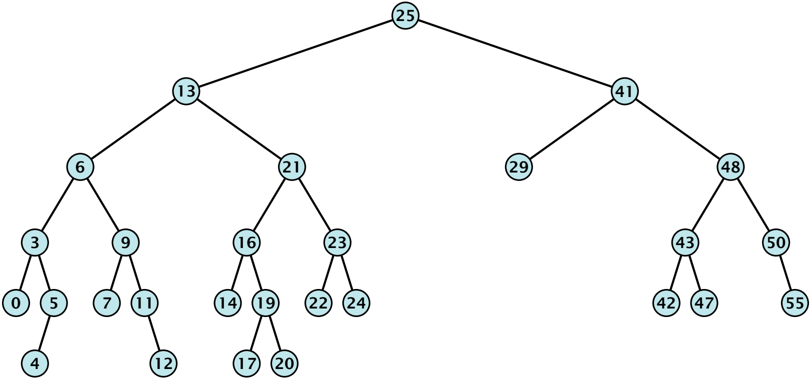


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete



Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

Algorithm 9 TreeDelete(z)

```
if $\text{left}[z] = \text{null}$ or $\text{right}[z] = \text{null}$
    then $y \leftarrow z$ else $y \leftarrow \text{TreeSucc}(z)$;        select $y$ to splice out
if $\text{left}[y] \neq \text{null}$
    then $x \leftarrow \text{left}[y]$ else $x \leftarrow \text{right}[y]$;   $x$ is child of $y$ (or null)
if $x \neq \text{null}$ then $\text{parent}[x] \leftarrow \text{parent}[y]$;    $\text{parent}[x]$ is correct
if $\text{parent}[y] = \text{null}$ then
    $\text{root}[T] \leftarrow x$
else
    if $y = \text{left}[\text{parent}[x]]$ then                    fix pointer to $x$
        $\text{left}[\text{parent}[y]] \leftarrow x$
    else
        $\text{right}[\text{parent}[y]] \leftarrow x$
if $y \neq z$ then copy $y$-data to $z$
```

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees

With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees

With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

7.2 Red Black Trees

Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

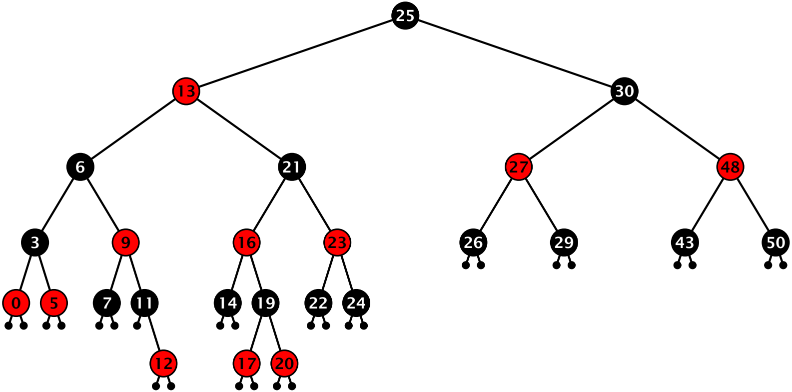
Definition 11

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a colour, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

Red Black Trees: Example



7.2 Red Black Trees

Lemma 12

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 13

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 14

A sub-tree of black height $\mathrm{bh}(v)$ *in a red black tree contains at least* $2^{\mathrm{bh}(v)} - 1$ *internal vertices.*

7.2 Red Black Trees

Lemma 12

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 13

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 14

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Lemma 12

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 13

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 14

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\text{height}(v) = 0)$

- If height(v) (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case (height$(v) = 0$)

- If height(v) (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case ($\mathrm{height}(v) = 0$)

- If $\mathrm{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case ($\text{height}(v) = 0$)

- If $\text{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\mathrm{height}(v) = 0)$

- If $\mathrm{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

□

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\mathrm{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\mathrm{bh}(c_i) = \mathrm{bh}(v)$ or $\mathrm{bh}(c_i) = \mathrm{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\mathrm{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\mathrm{bh}(v)-1} - 1) + 1 \geq 2^{\mathrm{bh}(v)} - 1$ vertices.

$\square$

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

$\square$

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\mathrm{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\mathrm{bh}(c_i) = \mathrm{bh}(v)$ or $\mathrm{bh}(c_i) = \mathrm{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\mathrm{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\mathrm{bh}(v)-1} - 1) + 1 \geq 2^{\mathrm{bh}(v)} - 1$ vertices.

□

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

☐

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\mathrm{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\mathrm{bh}(c_i) = \mathrm{bh}(v)$ or $\mathrm{bh}(c_i) = \mathrm{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\mathrm{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\mathrm{bh}(v)-1} - 1) + 1 \geq 2^{\mathrm{bh}(v)} - 1$ vertices.

☐

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

Hence, $h \leq 2 \log n + 1 = \mathcal{O}(\log n)$. ☐

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

Hence, $h \leq 2\log n + 1 = \mathcal{O}(\log n)$. ☐

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

Hence, $h \leq 2 \log n + 1 = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

Hence, $h \leq 2\log n + 1 = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \geq n$.

Hence, $h \leq 2\log n + 1 = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 12.

Let h denote the height of the red-black tree, and let p denote a path from the root to the furthest leaf.

At least half of the node on p must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2}-1$ internal vertices. Hence, $2^{h/2}-1 \geq n$.

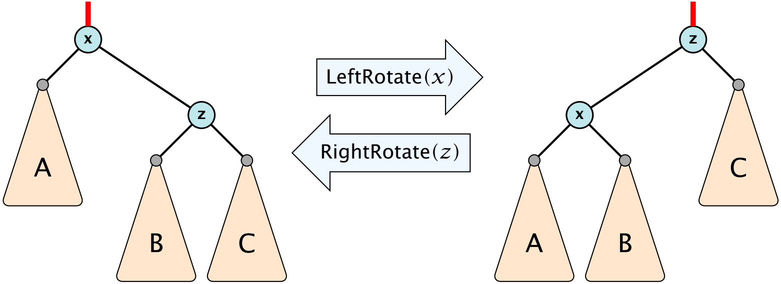
Hence, $h \leq 2\log n + 1 = \mathcal{O}(\log n)$. ☐

7.2 Red Black Trees

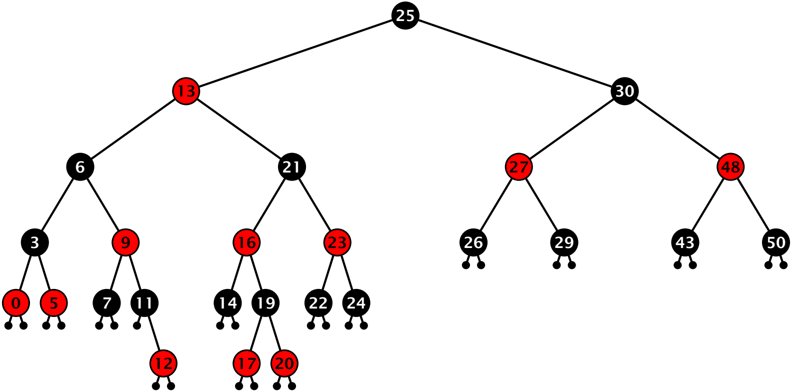
We need to adapt the insert and delete operations so that the red black properties are maintained.

Rotations

The properties will be maintained through rotations:



Red Black Trees: Insert

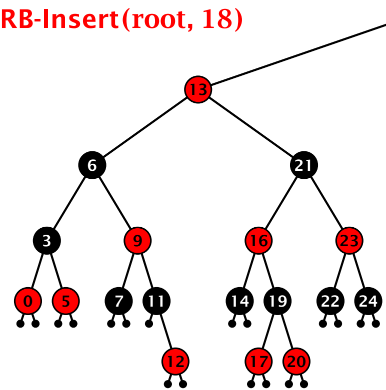


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

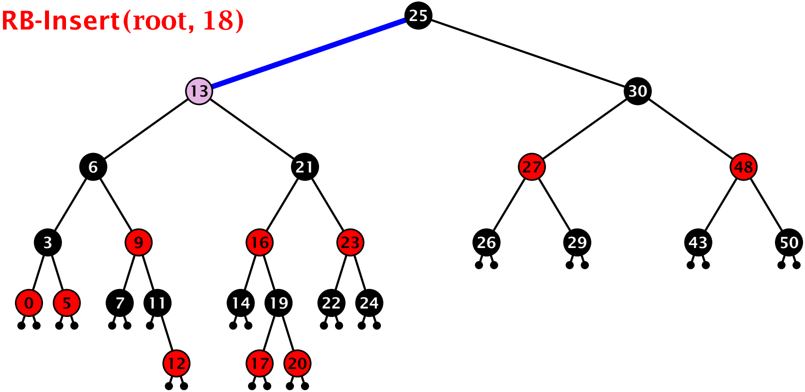
RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

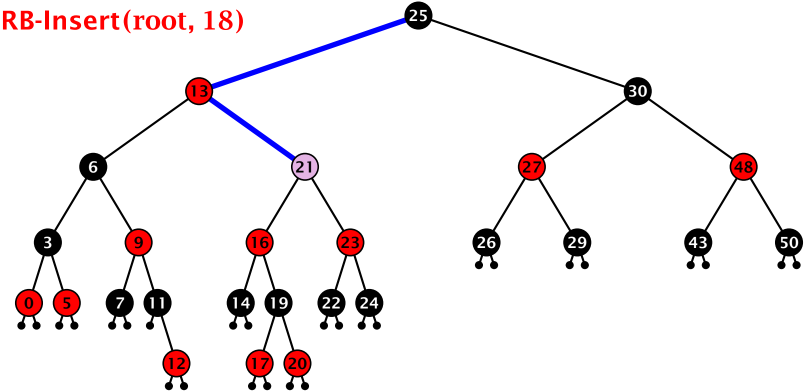
Red Black Trees: Insert



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

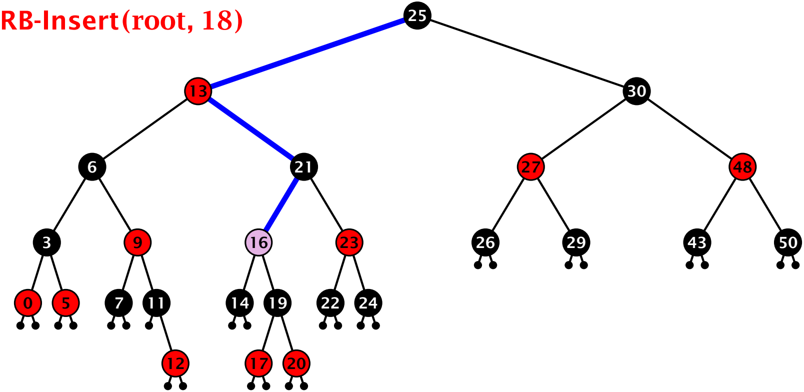


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

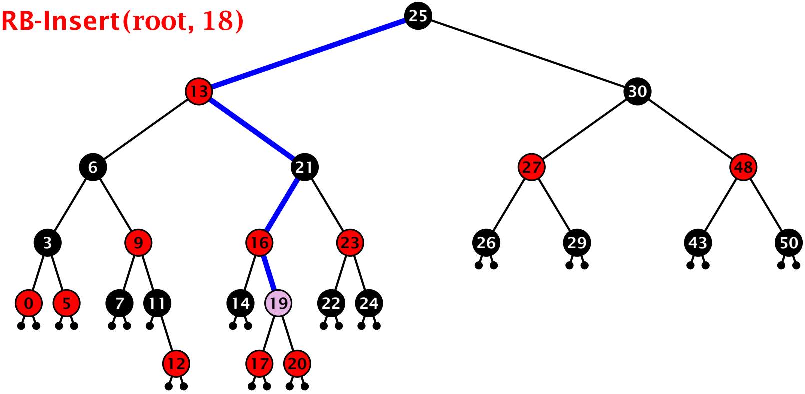
RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

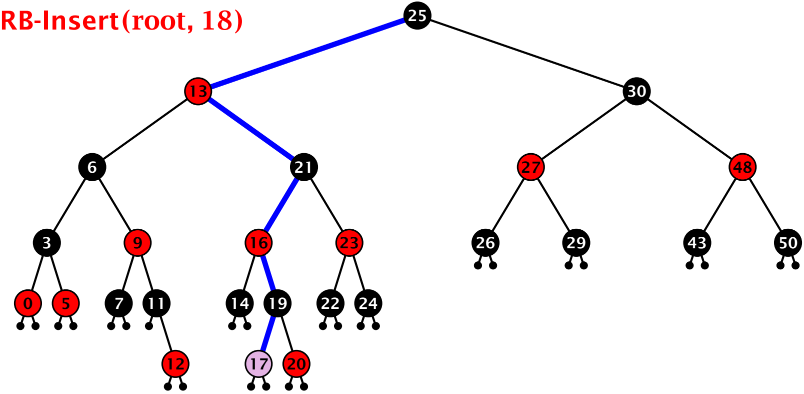


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)

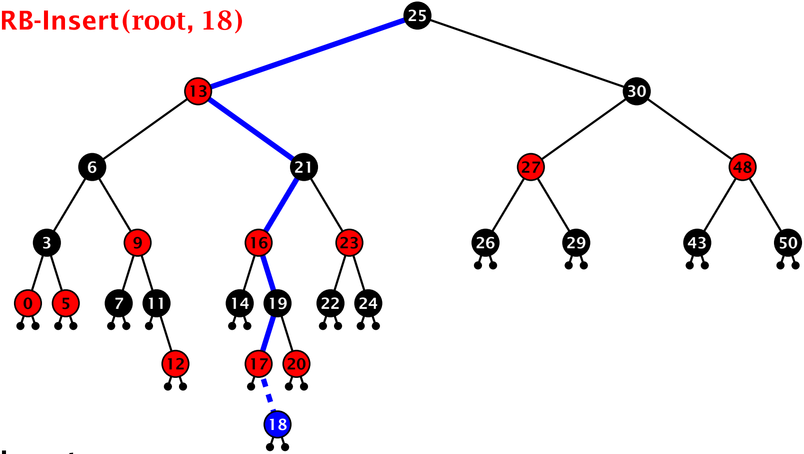


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)

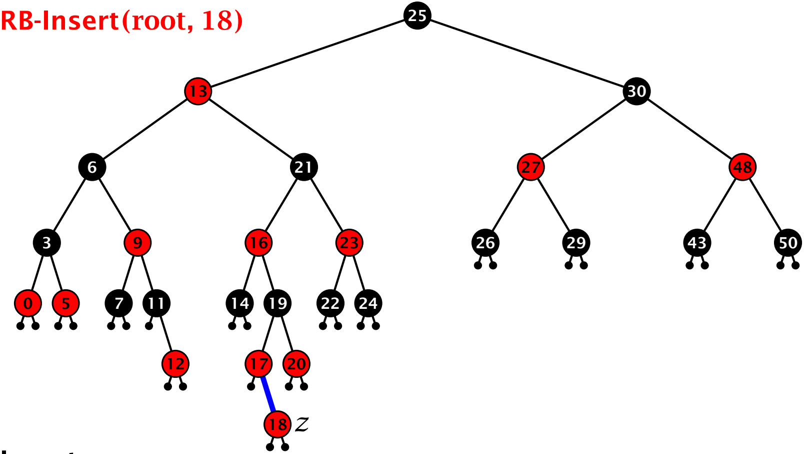


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
    - either both of them are red
      (most important case)
    - or the parent does not exist
      (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
    - either both of them are red (most important case)
    - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[$z$] $\neq$ null and col[parent[$z$]] = red do
    if parent[$z$] = left[gp[z]] then
        $uncle \leftarrow$ right[grandparent[$z$]]
        if col[$uncle$] = red then
            col[p[$z$]] $\leftarrow$ black; col[$u$] $\leftarrow$ black;
            col[gp[$z$]] $\leftarrow$ red; $z \leftarrow$ grandparent[$z$];
        else
            if $z$ = right[parent[$z$]] then
                $z \leftarrow$ p[$z$]; LeftRotate($z$);
            col[p[$z$]] $\leftarrow$ black; col[gp[$z$]] $\leftarrow$ red;
            RightRotate(gp[$z$]);
    else same as then-clause but right and left exchanged
col(root[$T$]) $\leftarrow$ black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[$z$] $\neq$ null and col[parent[$z$]] = red do
    if parent[$z$] = left[gp[z]] then    $z$ in left subtree of grandparent
        $uncle \leftarrow$ right[grandparent[$z$]]
        if col[$uncle$] = red then
            col[p[$z$]] $\leftarrow$ black; col[$u$] $\leftarrow$ black;
            col[gp[$z$]] $\leftarrow$ red; $z \leftarrow$ grandparent[$z$];
        else
            if $z$ = right[parent[$z$]] then
                $z \leftarrow$ p[$z$]; LeftRotate($z$);
            col[p[$z$]] $\leftarrow$ black; col[gp[$z$]] $\leftarrow$ red;
            RightRotate(gp[$z$]);
    else same as then-clause but right and left exchanged
col(root[$T$]) $\leftarrow$ black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then                              Case 1: uncle red
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[$z$] $\neq$ null and col[parent[$z$]] = red do
    if parent[$z$] = left[gp[z]] then
        $uncle \leftarrow$ right[grandparent[$z$]]
        if col[$uncle$] = red then
            col[p[$z$]] $\leftarrow$ black; col[$u$] $\leftarrow$ black;
            col[gp[$z$]] $\leftarrow$ red; $z \leftarrow$ grandparent[$z$];
        else                                                    Case 2: uncle black
            if $z$ = right[parent[$z$]] then
                $z \leftarrow$ p[$z$]; LeftRotate($z$);
            col[p[$z$]] $\leftarrow$ black; col[gp[$z$]] $\leftarrow$ red;
            RightRotate(gp[$z$]);
    else same as then-clause but right and left exchanged
col(root[$T$]) $\leftarrow$ black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

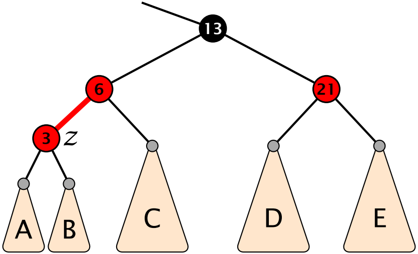
```
while parent[$z$] $\neq$ null and col[parent[$z$]] = red do
    if parent[$z$] = left[gp[z]] then
        $uncle \leftarrow$ right[grandparent[$z$]]
        if col[$uncle$] = red then
            col[p[$z$]] $\leftarrow$ black; col[$u$] $\leftarrow$ black;
            col[gp[$z$]] $\leftarrow$ red; $z \leftarrow$ grandparent[$z$];
        else
            if $z$ = right[parent[$z$]] then                2a: $z$ right child
                $z \leftarrow$ p[$z$]; LeftRotate($z$);
            col[p[$z$]] $\leftarrow$ black; col[gp[$z$]] $\leftarrow$ red;
            RightRotate(gp[$z$]);
    else same as then-clause but right and left exchanged
col(root[$T$]) $\leftarrow$ black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

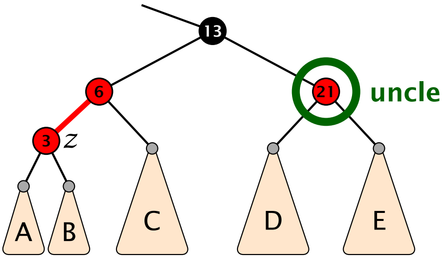
```
while parent[$z$] $\neq$ null and col[parent[$z$]] = red do
    if parent[$z$] = left[gp[z]] then
        $uncle \leftarrow$ right[grandparent[$z$]]
        if col[$uncle$] = red then
            col[p[$z$]] $\leftarrow$ black; col[$u$] $\leftarrow$ black;
            col[gp[$z$]] $\leftarrow$ red; $z \leftarrow$ grandparent[$z$];
        else
            if $z$ = right[parent[$z$]] then
                $z \leftarrow$ p[$z$]; LeftRotate($z$);
            col[p[$z$]] $\leftarrow$ black; col[gp[$z$]] $\leftarrow$ red;   2b: $z$ left child
            RightRotate(gp[$z$]);
    else same as then-clause but right and left exchanged
col(root[$T$]) $\leftarrow$ black;
```

Case 1: Red Uncle



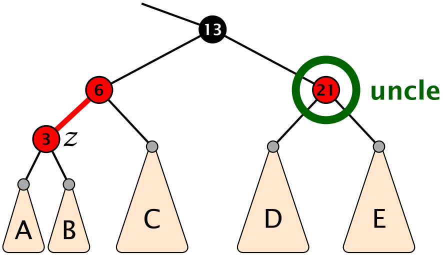
1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

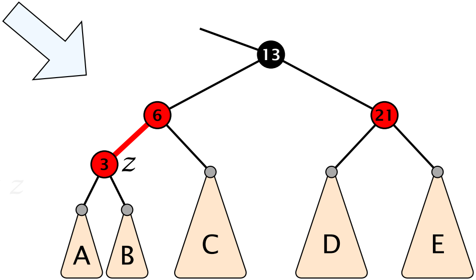
Case 1: Red Uncle



1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

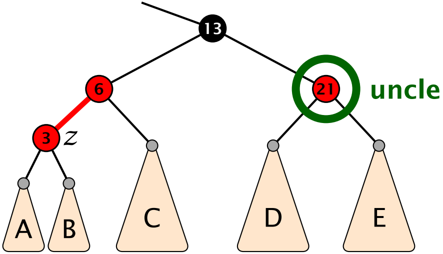
Case 1: Red Uncle

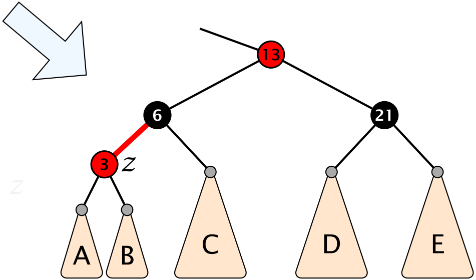




1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

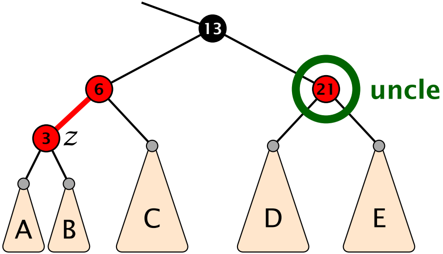
Case 1: Red Uncle

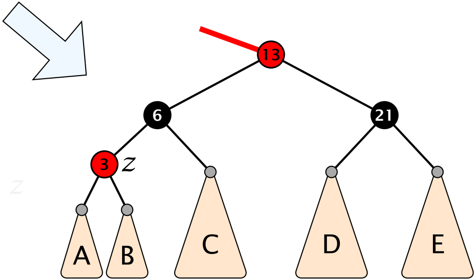




1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

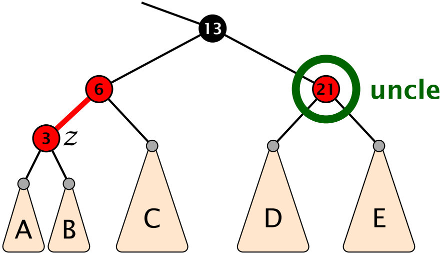
Case 1: Red Uncle

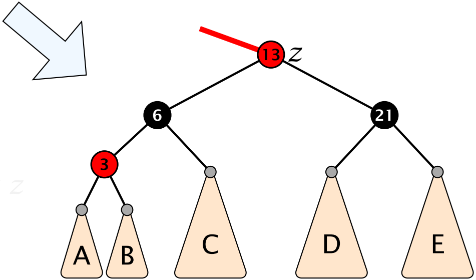




1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

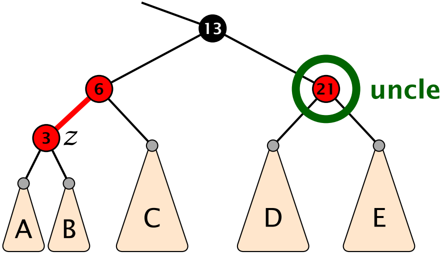
Case 1: Red Uncle

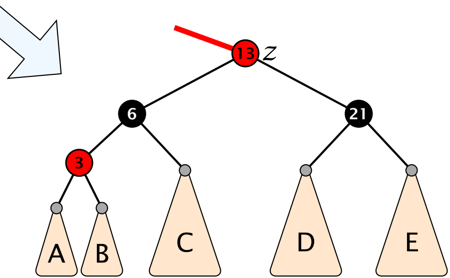




1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

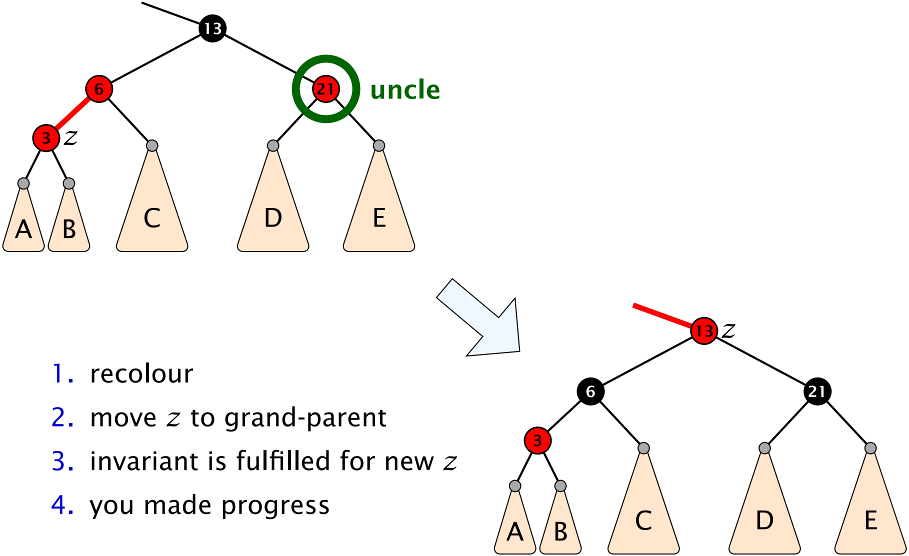
Case 1: Red Uncle





1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

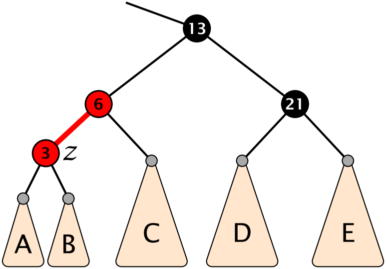
Case 1: Red Uncle



1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

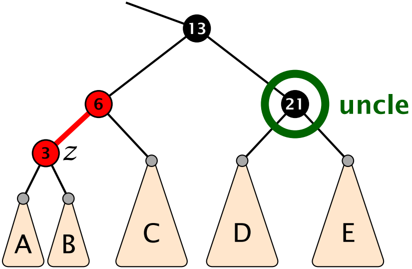
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



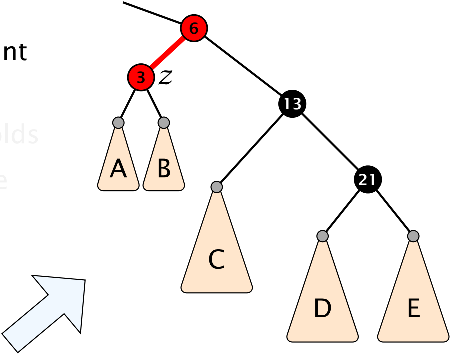
Case 2b: Black uncle and z is left child

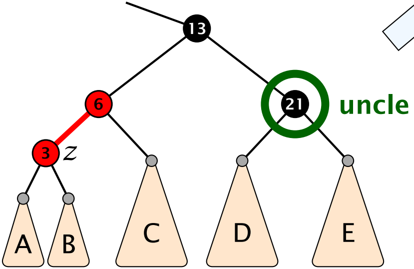
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



Case 2b: Black uncle and z is left child

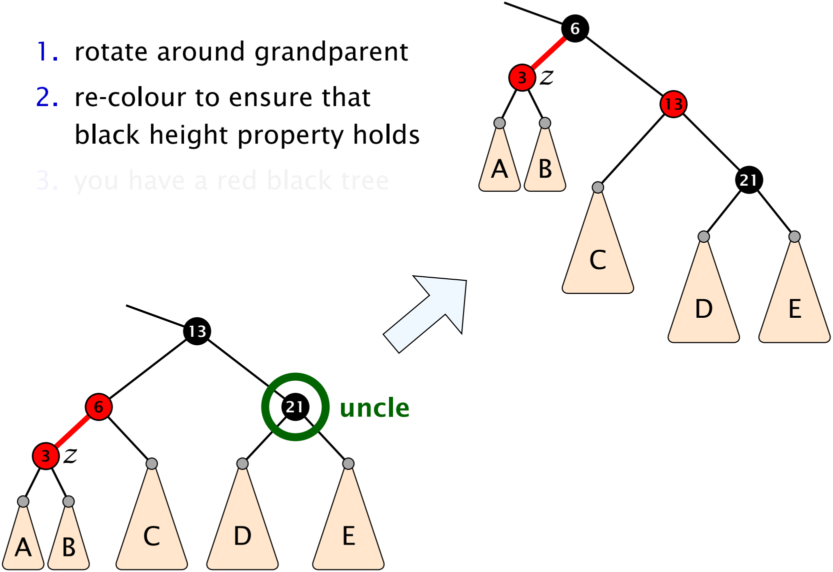
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree





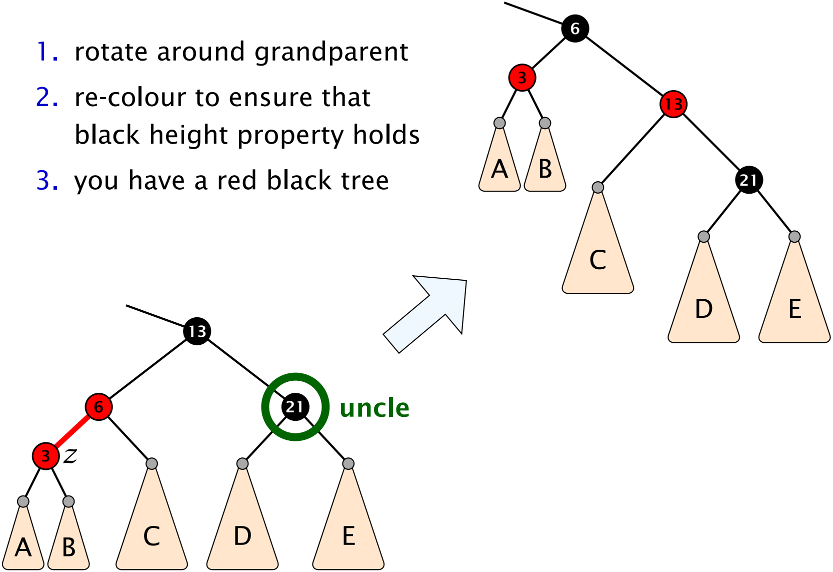
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



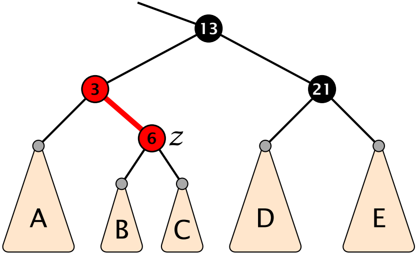
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



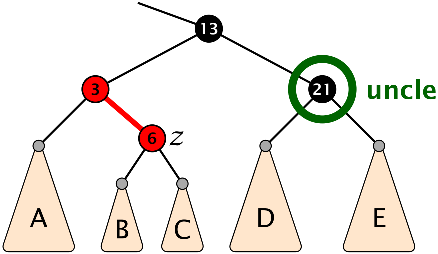
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



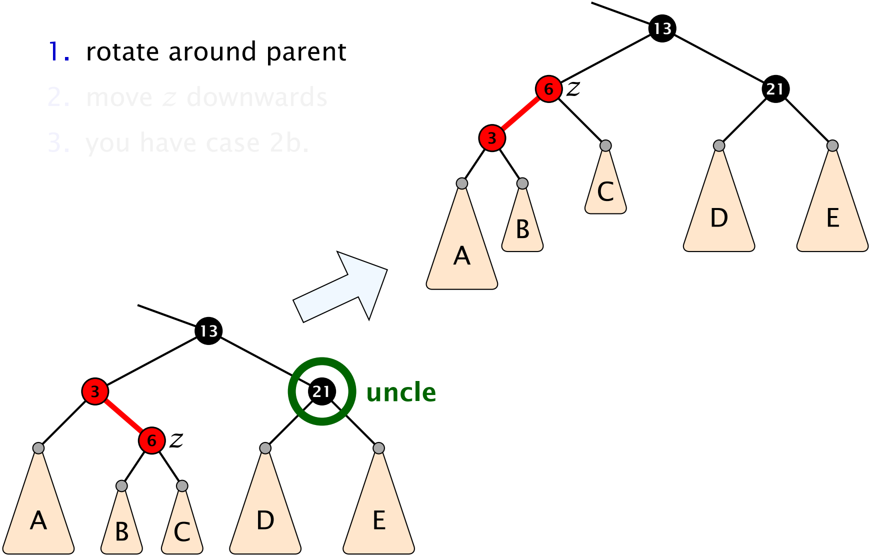
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



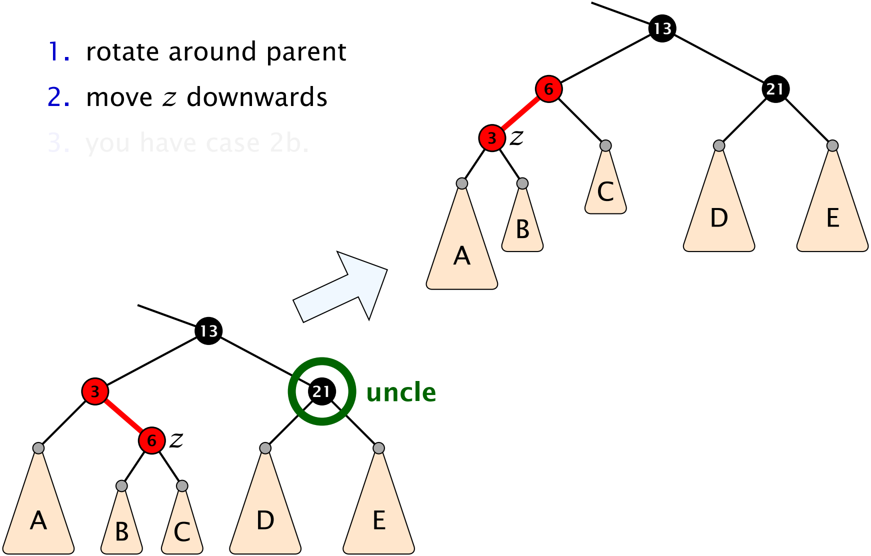
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



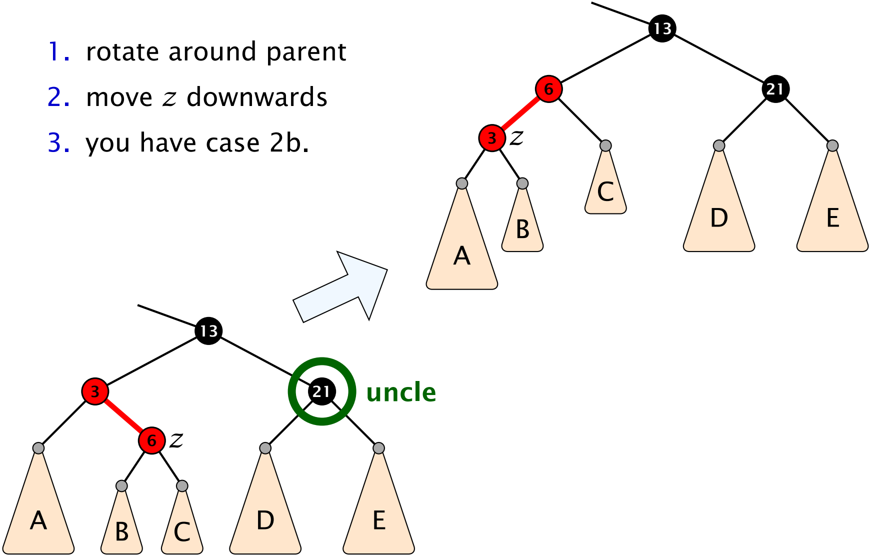
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have case 2b.



Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing step one $\mathcal{O}(\log n)$ times and every other step at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colourings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing step one $\mathcal{O}(\log n)$ times and every other step at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colourings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing step one $\mathcal{O}(\log n)$ times and every other step at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colourings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing step one $\mathcal{O}(\log n)$ times and every other step at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colourings and at most 2 rotations.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

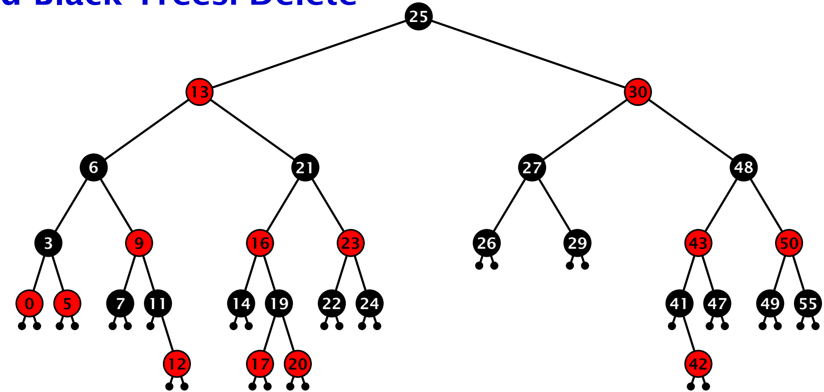
First do a standard delete.

If the spliced out node x was red everyhting is fine.

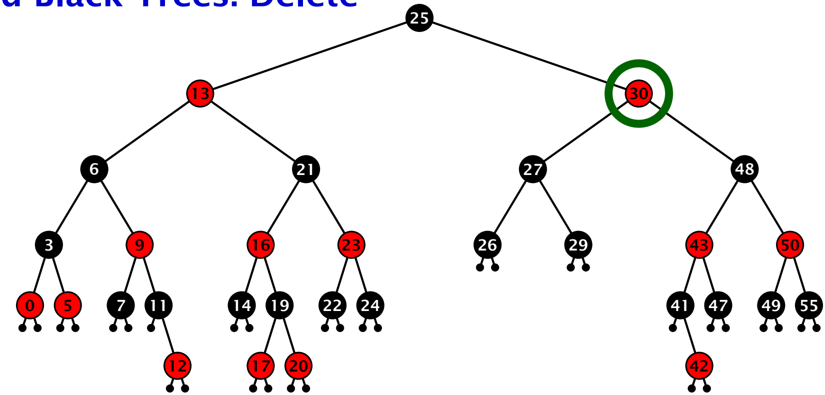
If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete



Red Black Trees: Delete

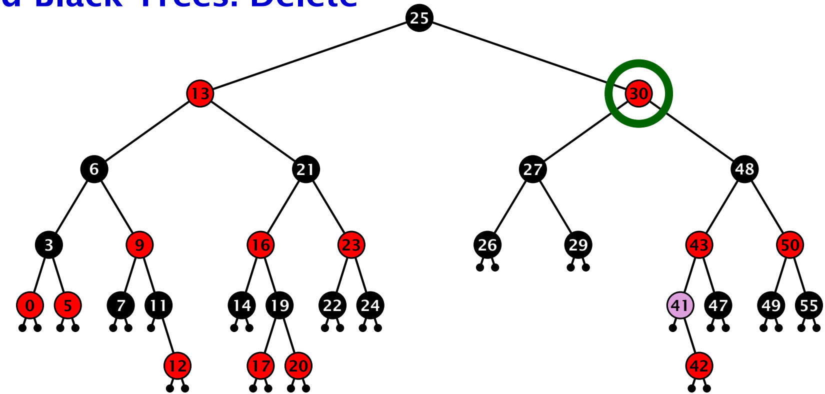


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

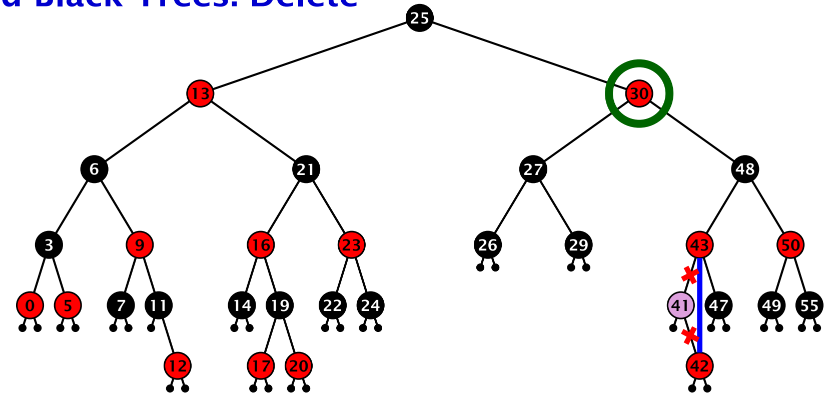


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

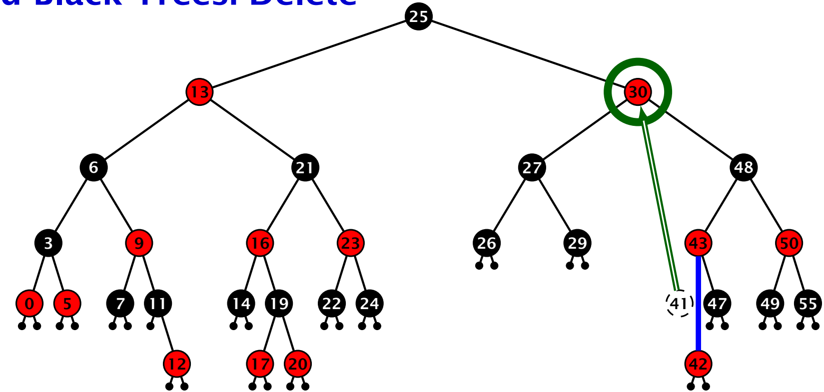


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

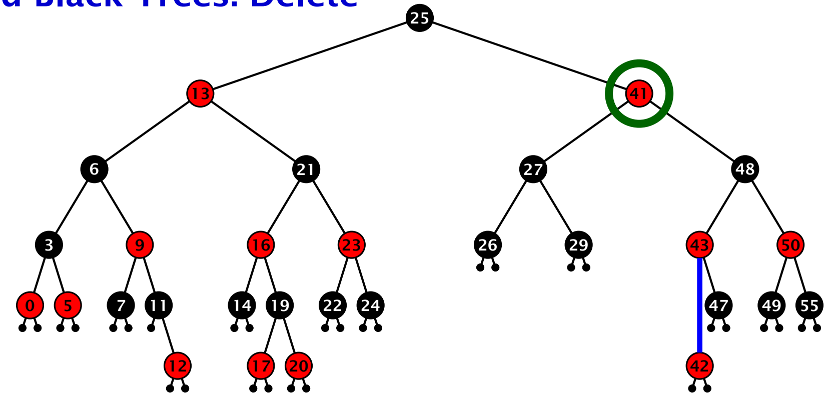


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

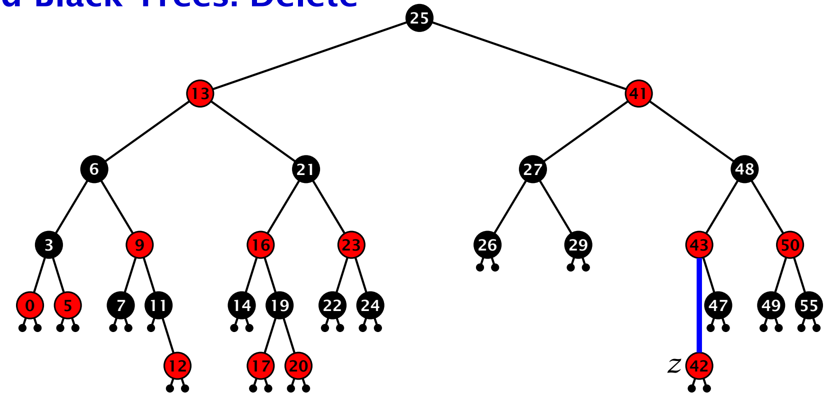


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

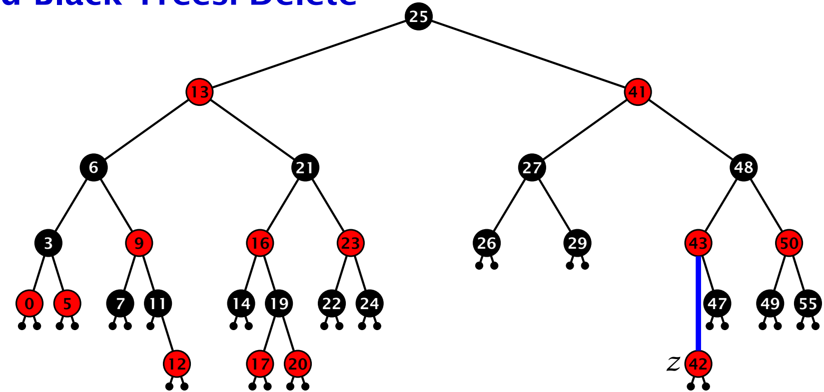
Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

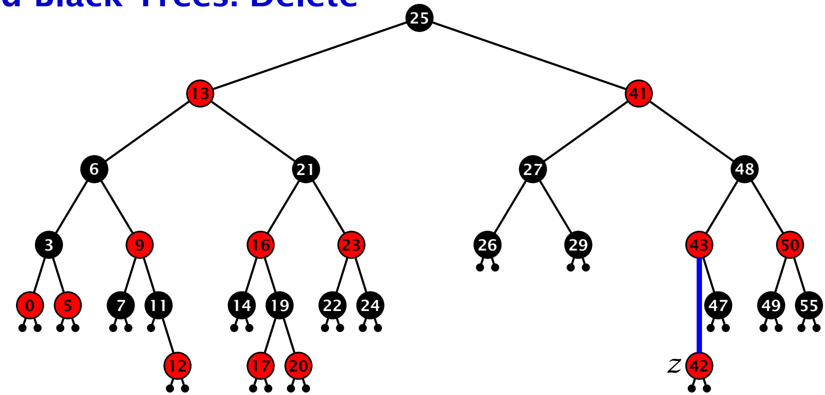
Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

Red Black Trees: Delete

Invariant of the fix-up algorihtm

- the node z is black
- if we “assign” a fake black unit to the edge from z to its parent then the black-height property is fulfilled

Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

Red Black Trees: Delete

Invariant of the fix-up algorihtm

- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

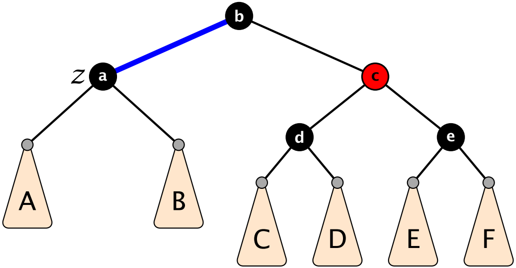
Red Black Trees: Delete

Invariant of the fix-up algorihtm

- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

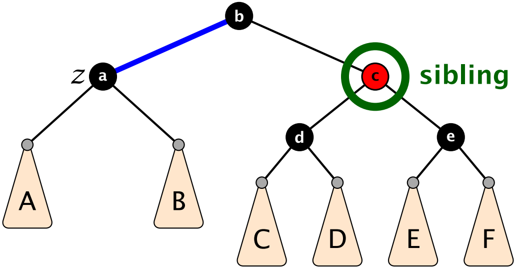
Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

Case 1: Sibling of z is red



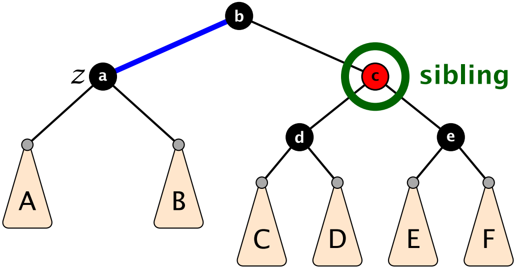
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

Case 1: Sibling of z is red

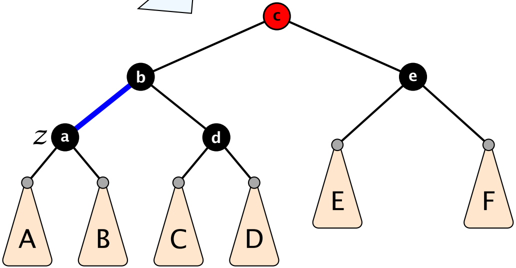


1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

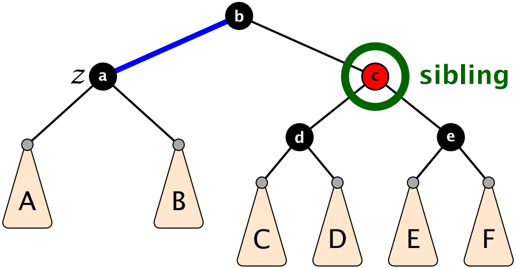
Case 1: Sibling of z is red



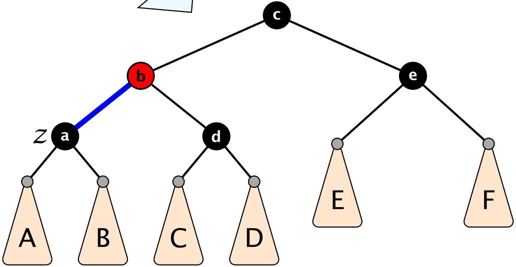
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4



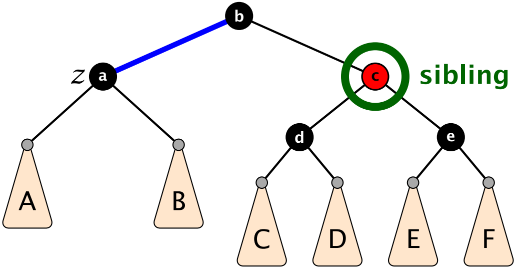
Case 1: Sibling of z is red



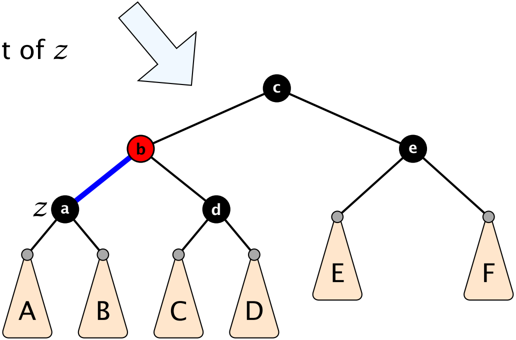
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4



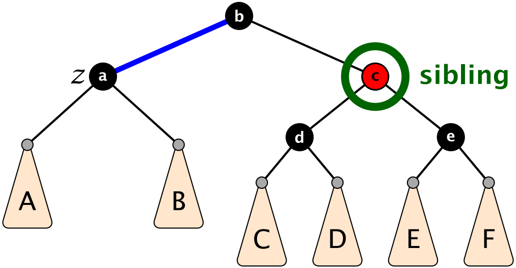
Case 1: Sibling of z is red



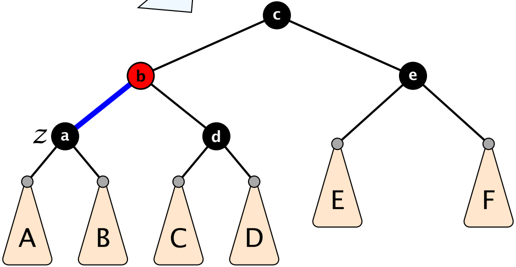
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4



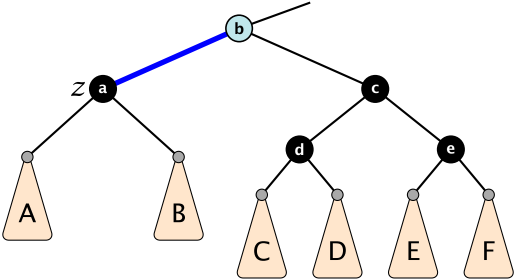
Case 1: Sibling of z is red



1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4

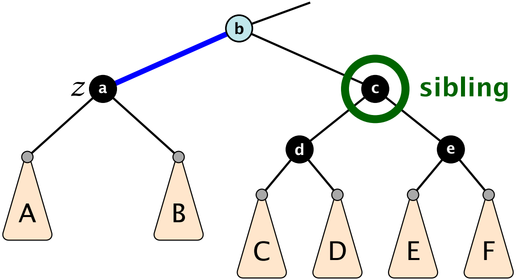


Case 2: Sibling is black with two black children



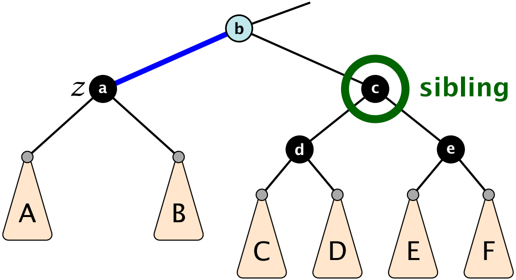
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

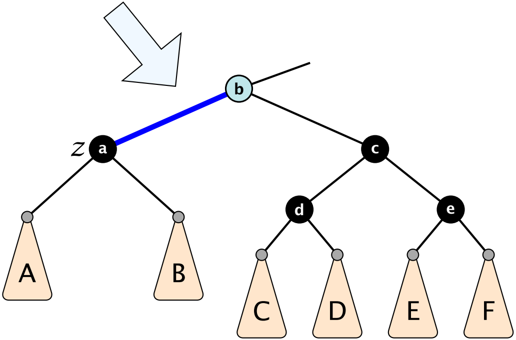
Case 2: Sibling is black with two black children



1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

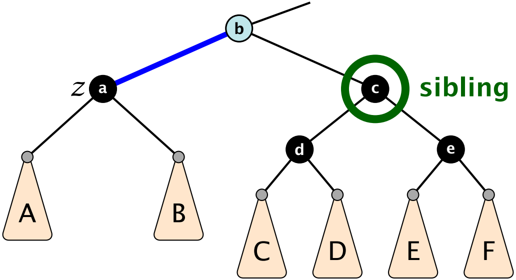
Case 2: Sibling is black with two black children



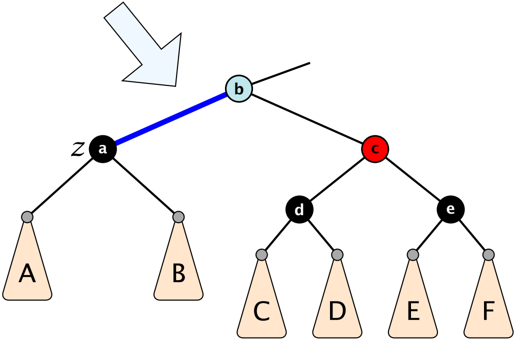


1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

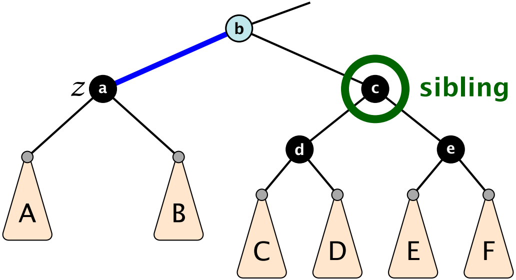
Case 2: Sibling is black with two black children



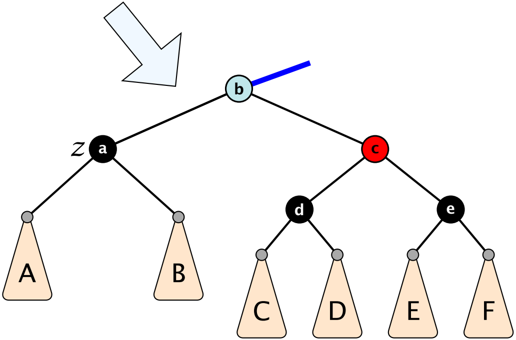
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



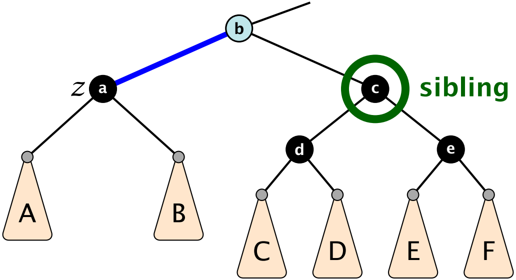
Case 2: Sibling is black with two black children



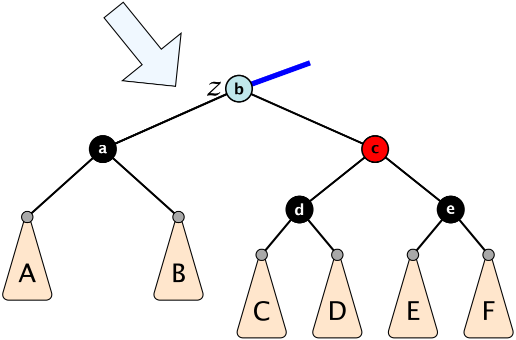
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



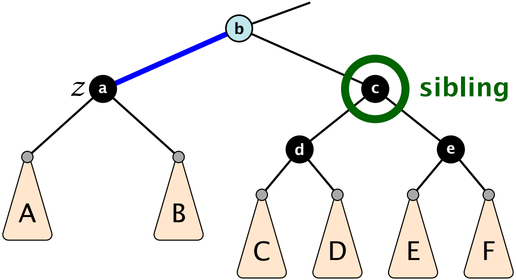
Case 2: Sibling is black with two black children



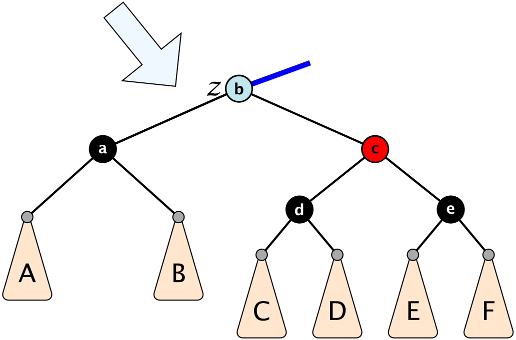
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



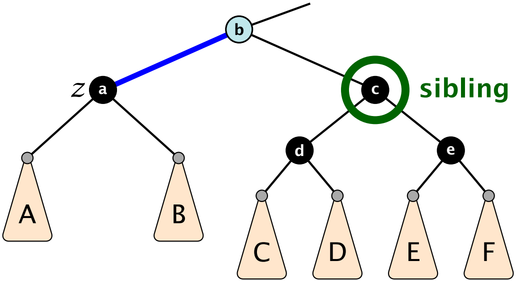
Case 2: Sibling is black with two black children

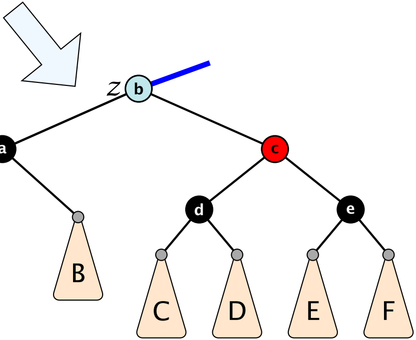


1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



Case 2: Sibling is black with two black children

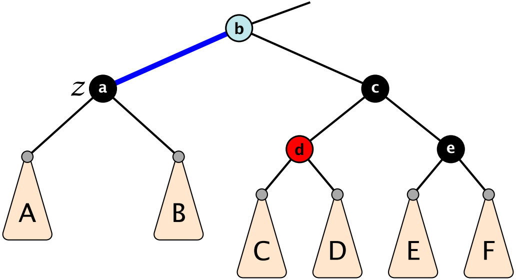




1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

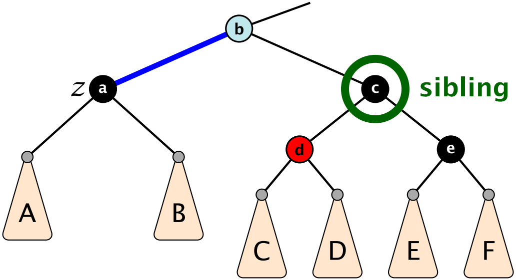
Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



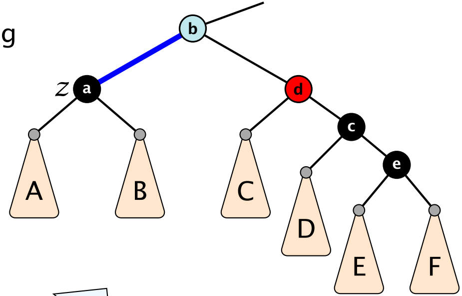
Case 3: Sibling black with one black child to the right

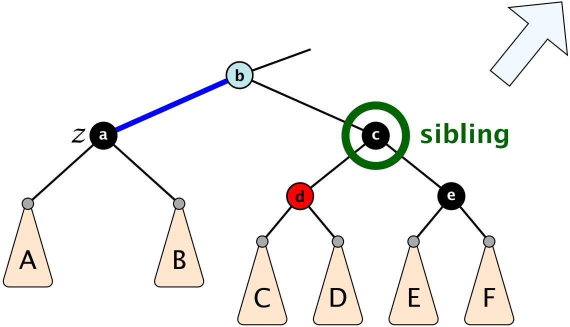
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



Case 3: Sibling black with one black child to the right

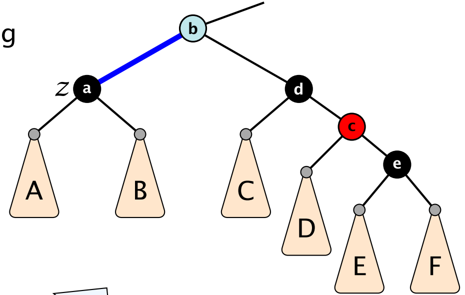
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)

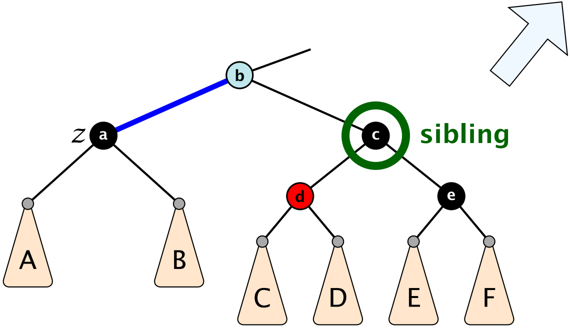




Case 3: Sibling black with one black child to the right

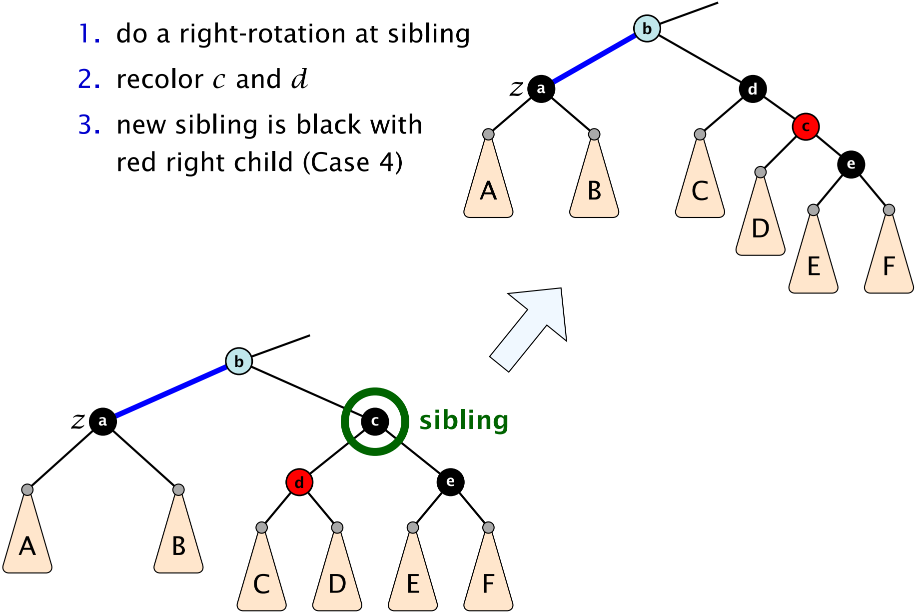
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



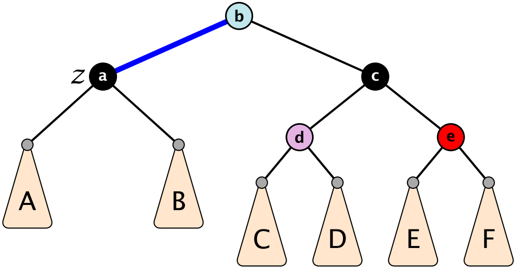


Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)

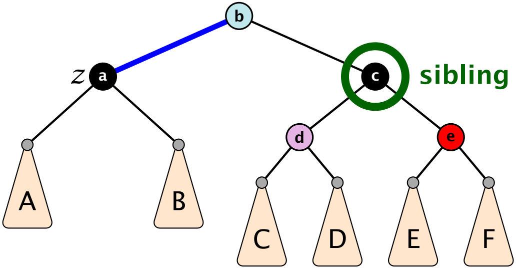


Case 4: Sibling is black with red right child



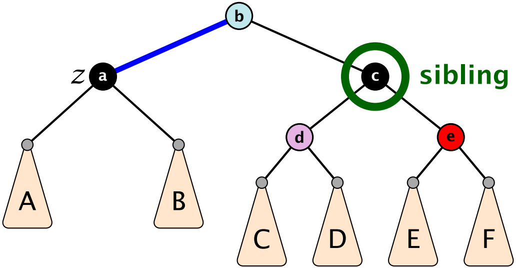
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree

Case 4: Sibling is black with red right child

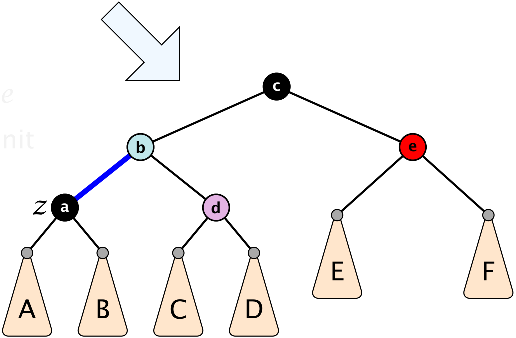


1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree

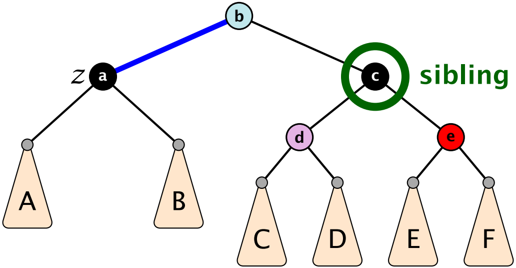
Case 4: Sibling is black with red right child



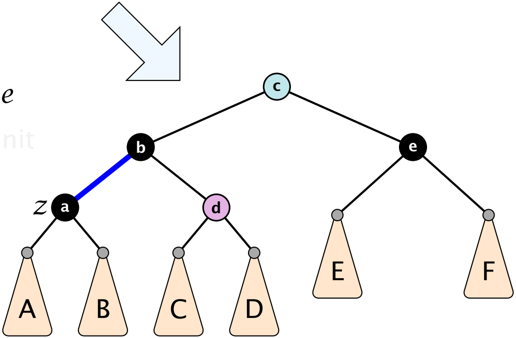
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree



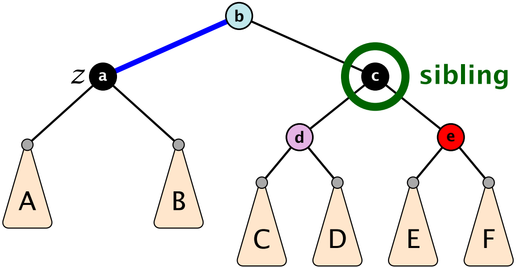
Case 4: Sibling is black with red right child



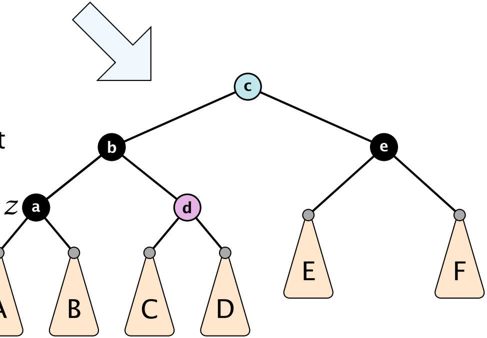
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree



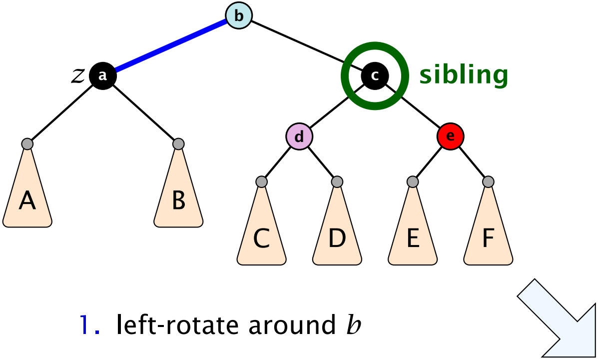
Case 4: Sibling is black with red right child



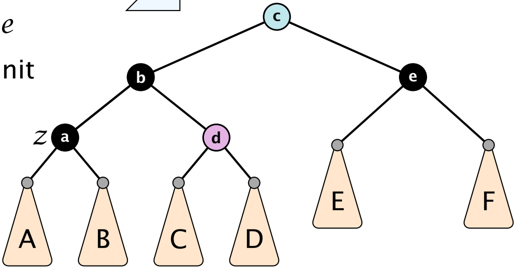
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree



Case 4: Sibling is black with red right child



1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree



Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $O(\log n)$ re-colourings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $O(\log n)$ re-colourings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $O(\log n)$ re-colourings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colourings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $O(\log n)$ re-colourings and at most 3 rotations.

7.3 AVL-Trees

Definition 15

AVL-trees are binary search trees that fulfill the following balance condition. For every node v

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \;.$$

Lemma 16

An AVL-tree of height h contains at least $F_{h+2} - 1$ and at most $2^h - 1$ internal nodes, where F_n is the n-th Fibonacci number ($F_0 = 0$, $F_1 = 1$), and the height is the maximal number of edges from the root to an (empty) dummy leaf.

7.3 AVL-Trees

Definition 15

AVL-trees are binary search trees that fulfill the following balance condition. For every node v

$$|\mathrm{height}(\text{left sub-tree}(v)) - \mathrm{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

Lemma 16

An AVL-tree of height h contains at least $F_{h+2} - 1$ and at most $2^h - 1$ internal nodes, where F_n is the n-th Fibonacci number ($F_0 = 0$, $F_1 = 1$), and the height is the maximal number of edges from the root to an (empty) dummy leaf.

Proof.

The upper bound is clear, as a binary tree of height h can only contain

$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

internal nodes.

Proof (cont.)

Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

Proof (cont.)

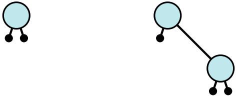
Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

Proof (cont.)

Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

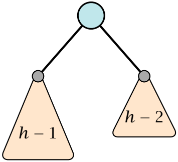


Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h - 1$ and $h - 2$, respectively. Both, sub-trees have minmal node number.

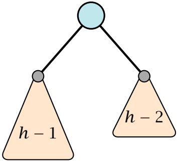
Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.

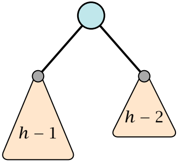


Let

$$f_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



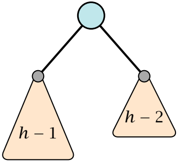
Let

$$f_h := 1 + \text{minimal size of AVL-tree of height } h\ .$$

Then

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

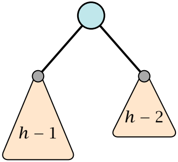
$$f_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

$$f_1 = 2 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad = F_3$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

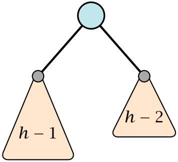
$$f_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

$$\begin{aligned} f_1 &= 2 && = F_3 \\ f_2 &= 3 && = F_4 \end{aligned}$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

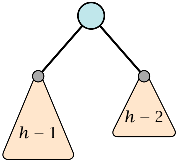
$$f_h := 1 + \text{minimal size of AVL-tree of height } h \;.$$

Then

$$\begin{aligned} f_1 &= 2 && = F_3 \\ f_2 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{hence} \end{aligned}$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

$$f_h := 1 + \text{minimal size of AVL-tree of height } h\ .$$

Then

$$\begin{aligned} f_1 &= 2 && = F_3 \\ f_2 &= 3 && = F_4 \\ f_h - 1 &= 1 + f_{h-1} - 1 + f_{h-2} - 1\,, && \text{hence} \\ f_h &= f_{h-1} + f_{h-2} && = F_{h+2} \end{aligned}$$

7.3 AVL-Trees

Since

$$F(k) \approx \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^k \quad ,$$

an AVL-tree with n internal nodes has height $\Theta(\log n)$.

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \enspace ,$$

where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

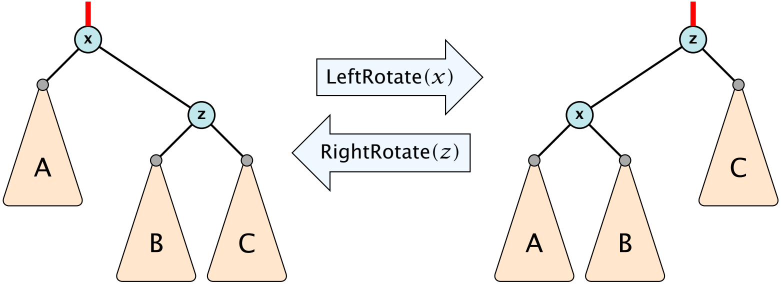
For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

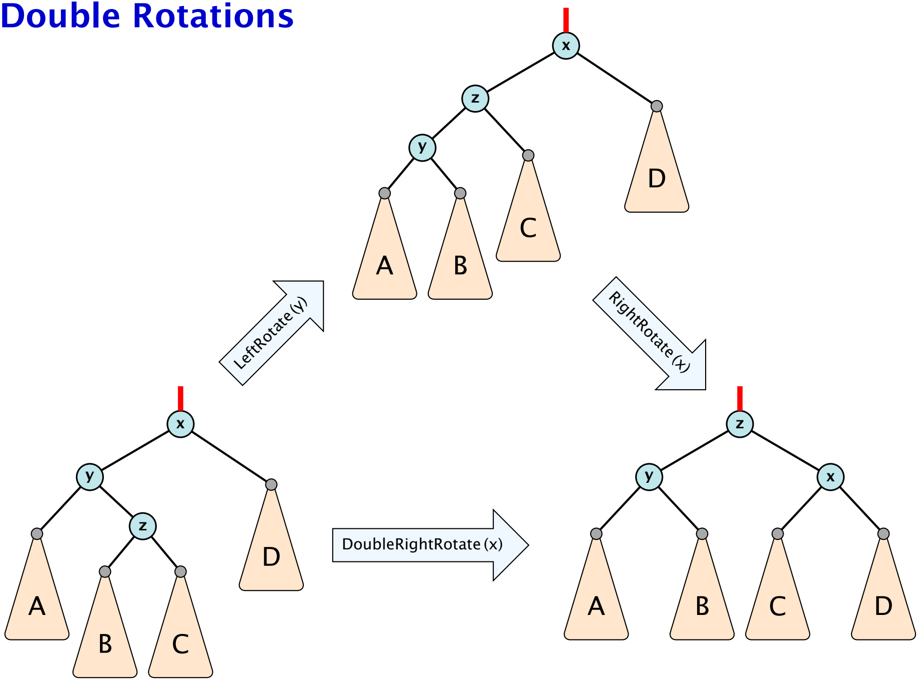
where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

Rotations

The properties will be maintained through rotations:



Double Rotations



AVL-trees: Insert

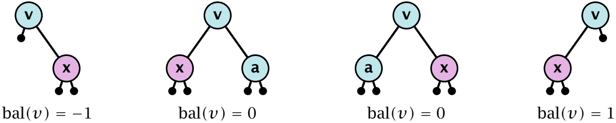
- Insert like in a binary search tree.

AVL-trees: Insert

- Insert like in a binary search tree.
- Let v denote the parent of the newly inserted node x.

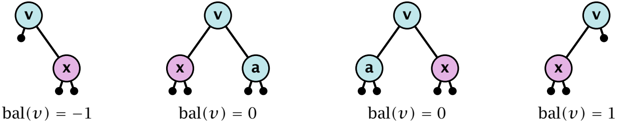
AVL-trees: Insert

- Insert like in a binary search tree.
- Let v denote the parent of the newly inserted node x.
- One of the following cases holds:



AVL-trees: Insert

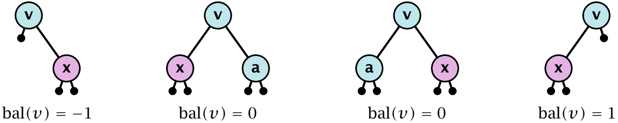
- Insert like in a binary search tree.
- Let v denote the parent of the newly inserted node x.
- One of the following cases holds:



- If $\text{bal}[v] \neq 0$, T_v has changed height; the balance-constraint may be violated at ancestors of v.

AVL-trees: Insert

- Insert like in a binary search tree.
- Let v denote the parent of the newly inserted node x.
- One of the following cases holds:



- If $\text{bal}[v] \neq 0$, T_v has changed height; the balance-constraint may be violated at ancestors of v.
- Call fix-up$(\text{parent}[v])$ to restore the balance-condition.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\mathrm{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at the node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Insert

Algorithm 11 AVL-fix-up-insert(v)

```
if balance[$v$] $\in \{-2, 2\}$ then DoRotationInsert($v$);
if balance[$v$] $\in \{0\}$ return;
AVL-fix-up-insert(parent[$v$]);
```

We will show that the above procedure is correct, and that it will do at most one rotation.

AVL-trees: Insert

Algorithm 12 DoRotationInsert(v)

```
if balance[$v$] = $-2$ then
    if balance[right[$v$]] = $-1$ then
        LeftRotate($v$);
    else
        DoubleLeftRotate($v$);
else
    if balance[left[$v$]] = $1$ then
        RightRotate($v$);
    else
        DoubleRightRotate($v$);
```

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

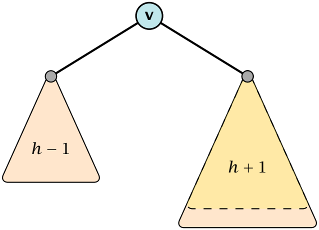
We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

We have the following situation:

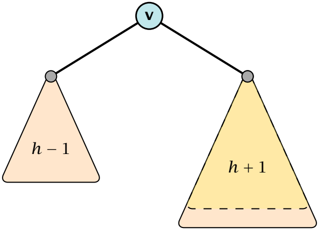


The right sub-tree of v has increased its height which results in a balance of -2 at v.

Before the insertion the height of T_v was $h+1$.

AVL-trees: Insert

We have the following situation:



The right sub-tree of v has increased its height which results in a balance of -2 at v.

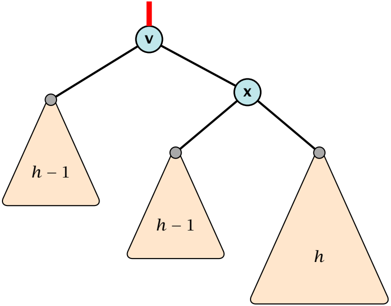
Before the insertion the height of T_v was $h+1$.

Case 1: balance[right[v]] $= -1$

We do a left rotation at v

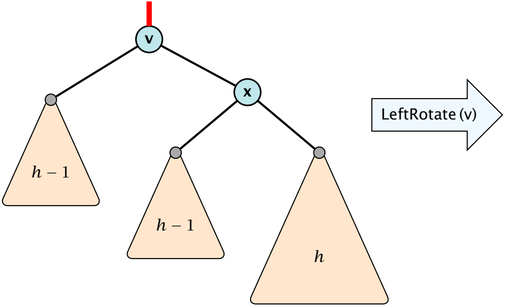
Case 1: balance[right[v]] $= -1$

We do a left rotation at v



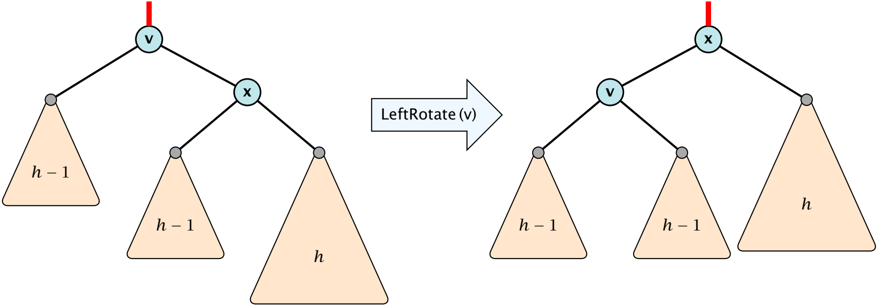
Case 1: balance[right[v]] = −1

We do a left rotation at v



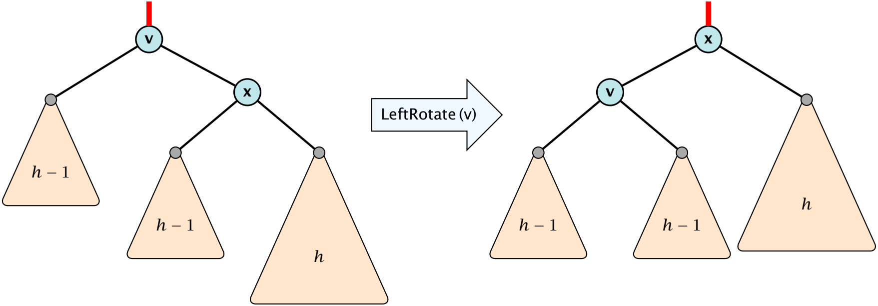
Case 1: balance[right[v]] $= -1$

We do a left rotation at v



Case 1: balance[right[v]] = −1

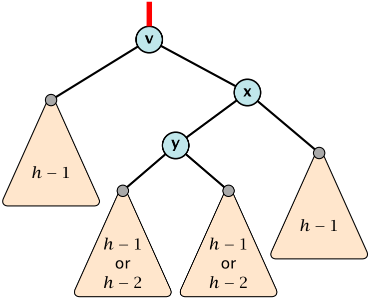
We do a left rotation at v

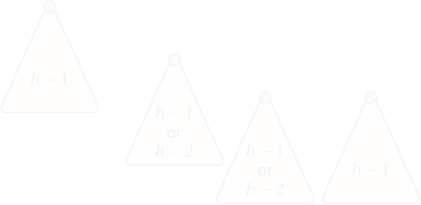


Now, T_v has height $h+1$ as before the insertion. Hence, we do not need to continue.

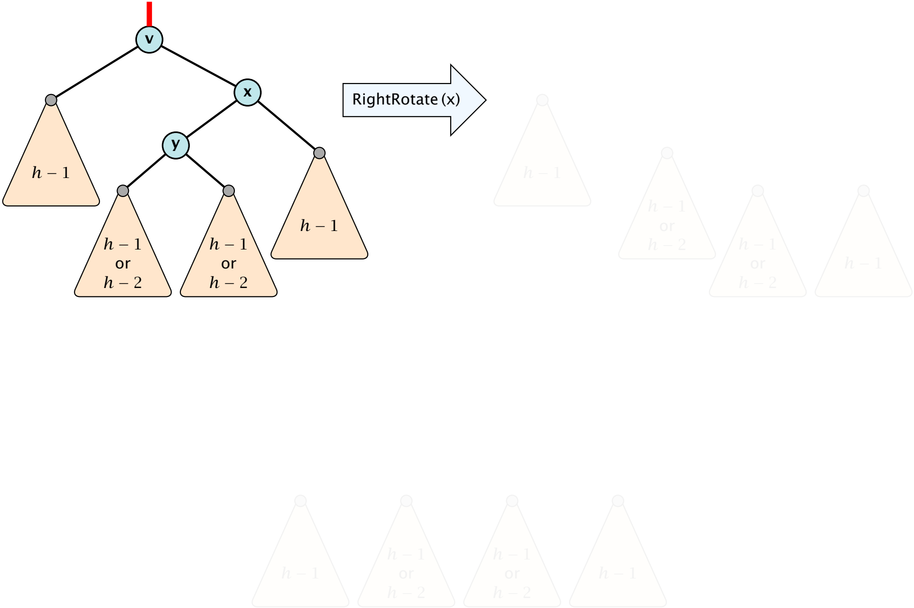
Case 2: balance[right[v]] = 1

Case 2: balance[right[v]] = 1

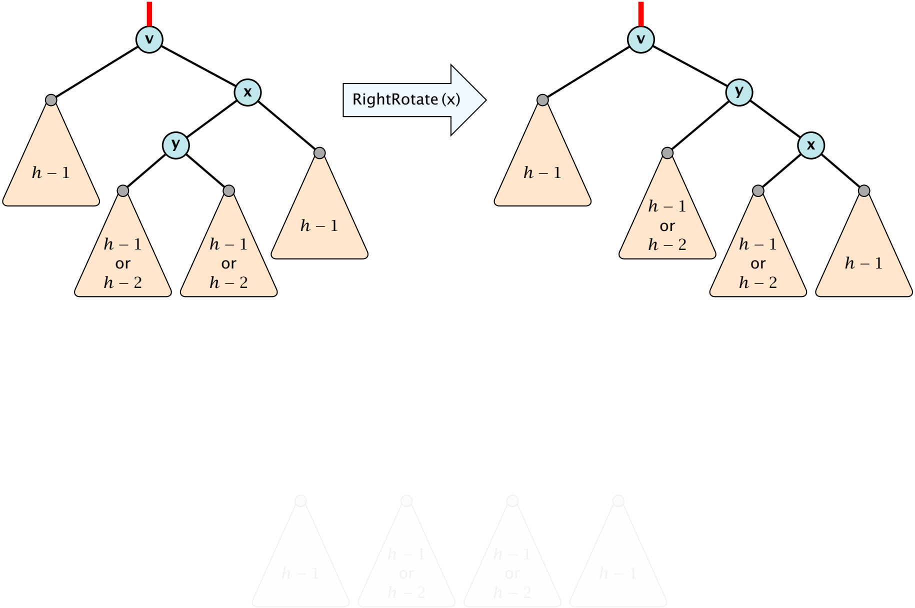




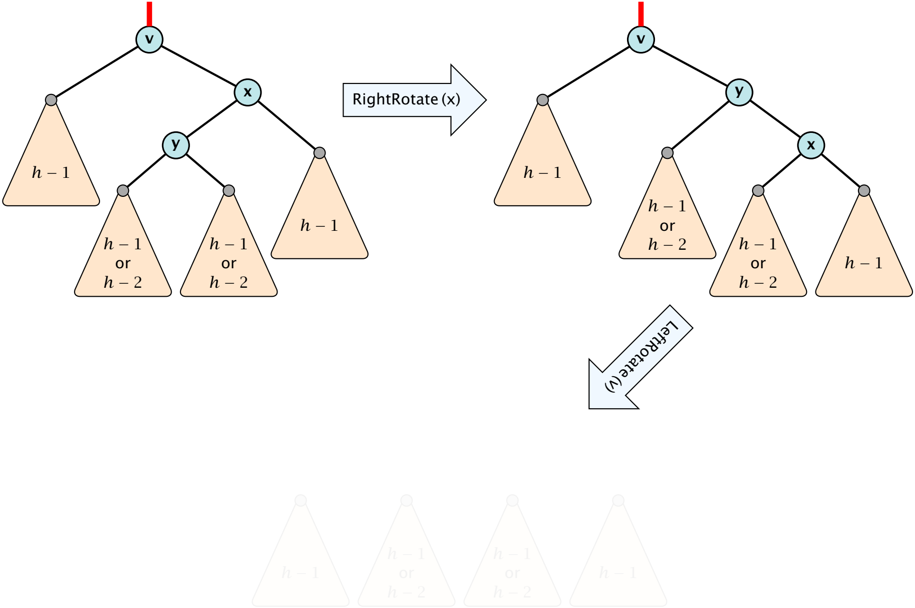
Case 2: balance[right[v]] = 1



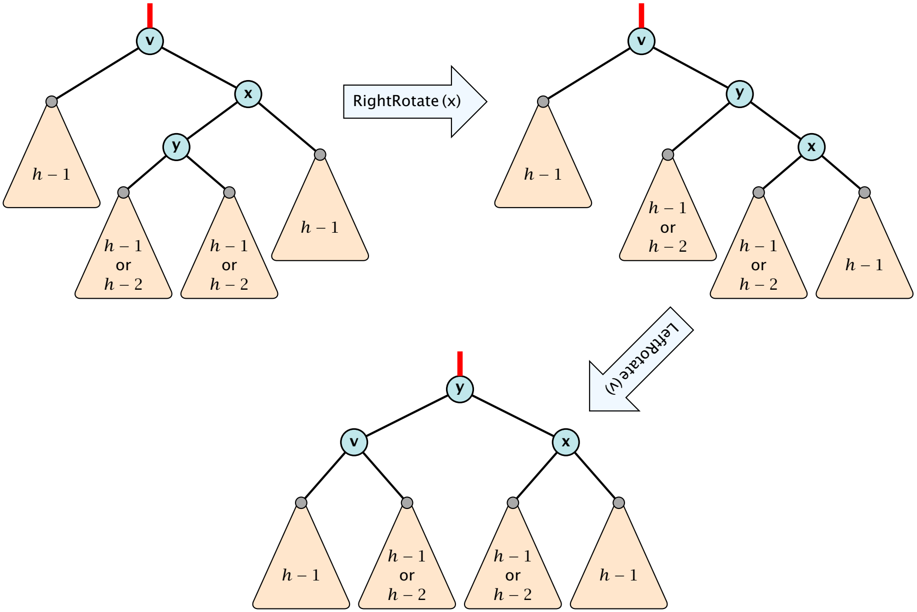
Case 2: balance[right[v]] = 1



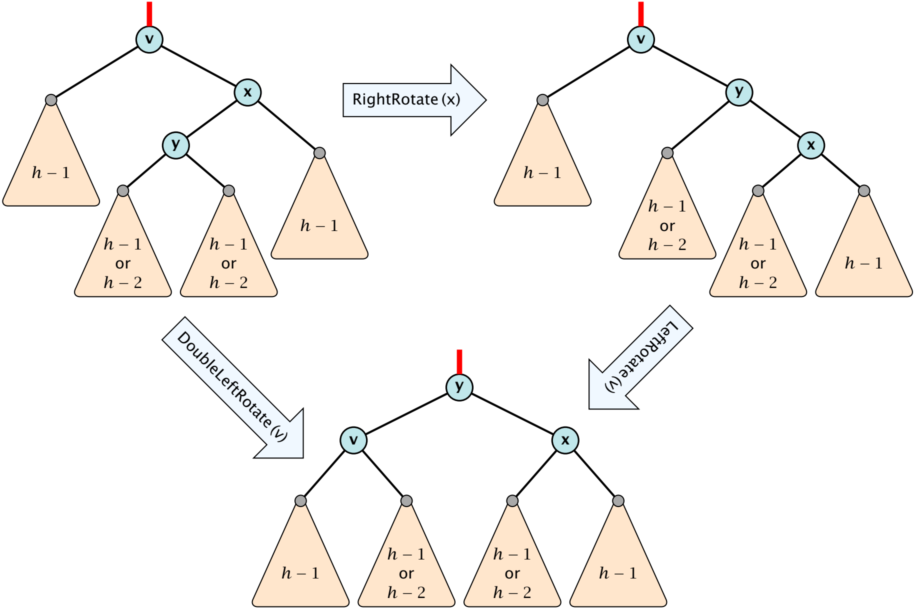
Case 2: balance[right[v]] = 1



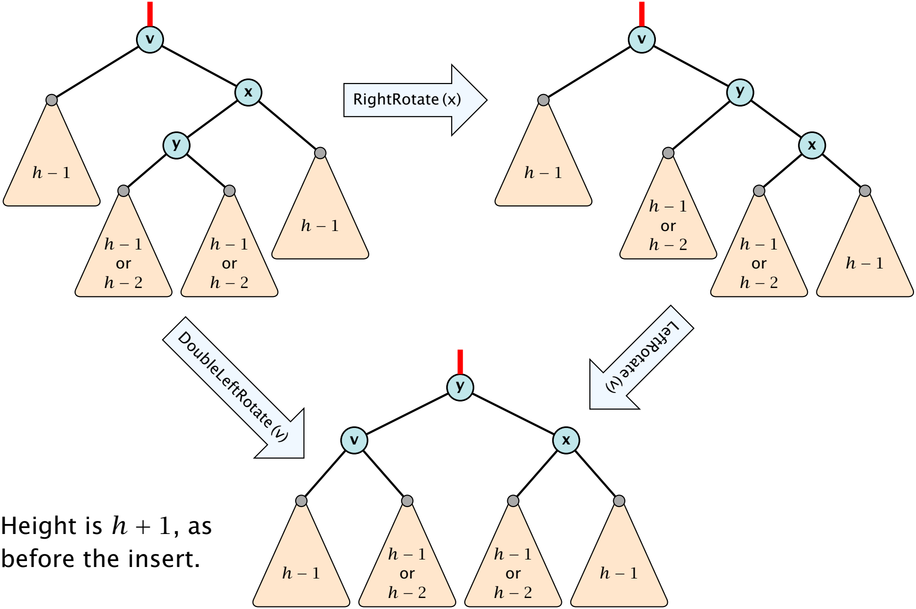
Case 2: balance[right[v]] = 1



Case 2: balance[right[v]] = 1



Case 2: balance[right[v]] = 1



Height is $h + 1$, as before the insert.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.

Case 1 Case 2

In both cases $\mathrm{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.

Case 1 Case 2

In both cases $\text{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.

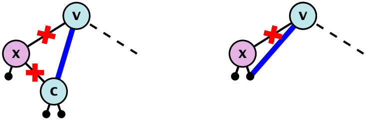
Case 1 Case 2

In both cases $\mathrm{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.



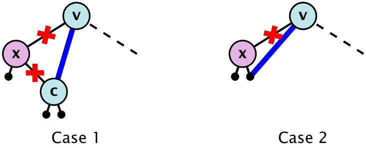
Case 1 Case 2

In both cases $\text{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed— is either a dummy leaf or a node with two dummy leafs as children.



In both cases $\text{bal}[c] = 0$.

- Call fix-up(v) to restore the balance-condition.

AVL-trees: Delete

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has either decreased its height by one or it has stayed the same (note that this is clear right after the deletion but we have to make sure that it also holds after the rotations done within T_c in previous iterations).
4. The balance at the node c fulfills $\text{balance}[c] = \{0\}$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has either decreased its height by one or it has stayed the same (note that this is clear right after the deletion but we have to make sure that it also holds after the rotations done within T_c in previous iterations).
4. The balance at the node c fulfills balance$[c] = \{0\}$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has either decreased its height by one or it has stayed the same (note that this is clear right after the deletion but we have to make sure that it also holds after the rotations done within T_c in previous iterations).
4. The balance at the node c fulfills $\text{balance}[c] = \{0\}$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning fix-up(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has either decreased its height by one or it has stayed the same (note that this is clear right after the deletion but we have to make sure that it also holds after the rotations done within T_c in previous iterations).
4. The balance at the node c fulfills $\text{balance}[c] = \{0\}$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure will have been aborted in the previous step.

AVL-trees: Delete

Algorithm 13 AVL-fix-up-delete(v)

```
if $\text{balance}[v] \in \{-2, 2\}$ then DoRotationDelete($v$);
if $\text{balance}[v] \in \{-1, 1\}$ return;
AVL-fix-up-delete($\text{parent}[v]$);
```

We will show that the above procedure is correct. However, for the case of a delete there may be a logarithmic number of rotations.

AVL-trees: Delete

Algorithm 13 AVL-fix-up-delete(v)

```
if balance[$v$] $\in \{-2, 2\}$ then DoRotationDelete($v$);
if balance[$v$] $\in \{-1, 1\}$ return;
AVL-fix-up-delete(parent[$v$]);
```

We will show that the above procedure is correct. However, for the case of a delete there may be a logarithmic number of rotations.

AVL-trees: Delete

Algorithm 14 DoRotationDelete(v)

```
if $\text{balance}[v] = -2$ then
    if $\text{balance}[\text{right}[v]] = -1$ then
        LeftRotate($v$);
    else
        DoubleLeftRotate($v$);
else
    if $\text{balance}[\text{left}[v]] = \{0, 1\}$ then
        RightRotate($v$);
    else
        DoubleRightRotate($v$);
```

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\mathrm{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

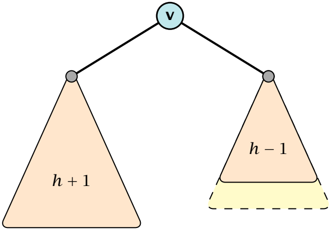
We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

We have the following situation:

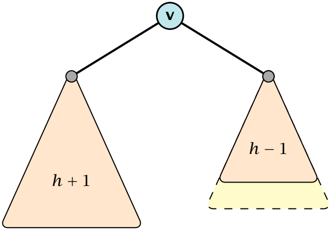


The right sub-tree of v has decreased its height which results in a balance of 2 at v.

Before the insertion the height of T_v was $h + 2$.

AVL-trees: Delete

We have the following situation:



The right sub-tree of v has decreased its height which results in a balance of 2 at v.

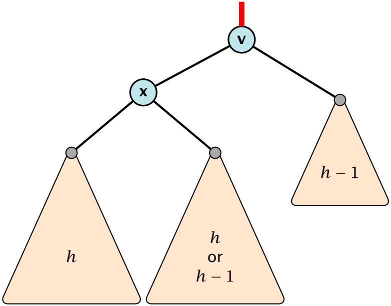
Before the insertion the height of T_v was $h + 2$.

Case 1: balance[left[v]] $\in \{0, 1\}$

If the middle subtree has height h the whole tree has height $h+2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h-1$ the whole tree has decreased its height from $h+2$ to $h+1$. We do continue the fix-up procedure as the balance at the root is zero.

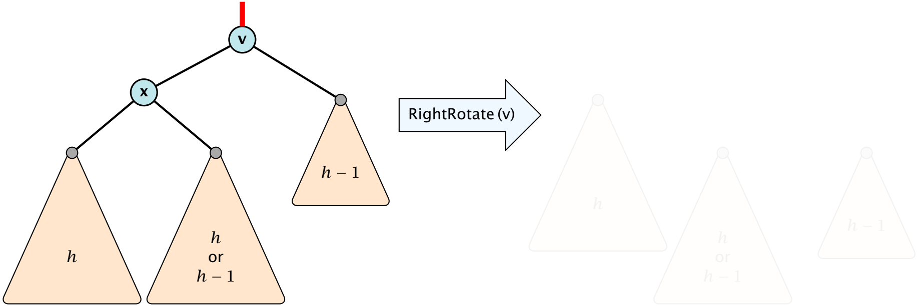
Case 1: balance[left[v]] $\in \{0, 1\}$



If the middle subtree has height h the whole tree has height $h+2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h-1$ the whole tree has decreased its height from $h+2$ to $h+1$. We do continue the fix-up procedure as the balance at the root is zero.

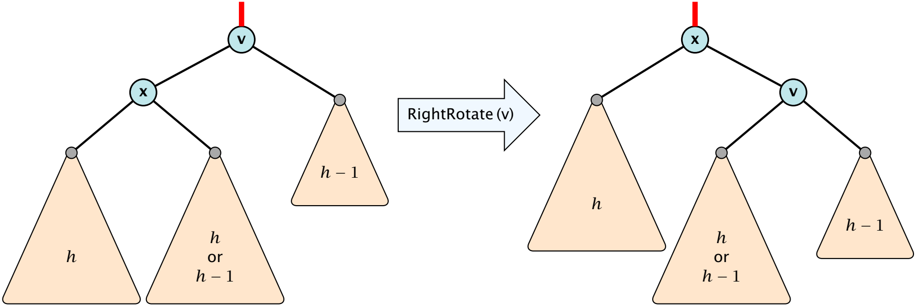
Case 1: balance[left[v]] $\in \{0, 1\}$



If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h - 1$ the whole tree has decreased its height from $h + 2$ to $h + 1$. We do continue the fix-up procedure as the balance at the root is zero.

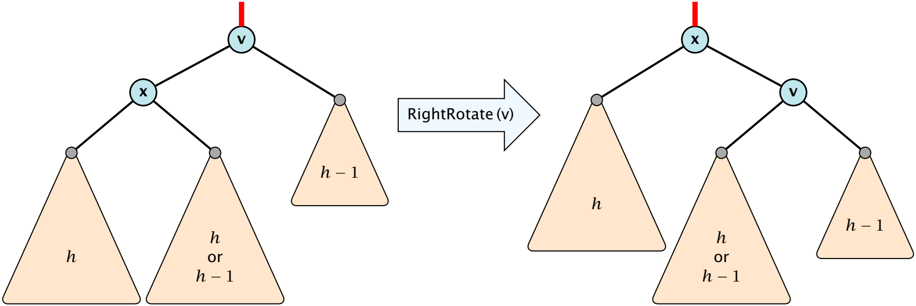
Case 1: balance[left[v]] $\in$ {0, 1}



If the middle subtree has height h the whole tree has height $h+2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h-1$ the whole tree has decreased its height from $h+2$ to $h+1$. We do continue the fix-up procedure as the balance at the root is zero.

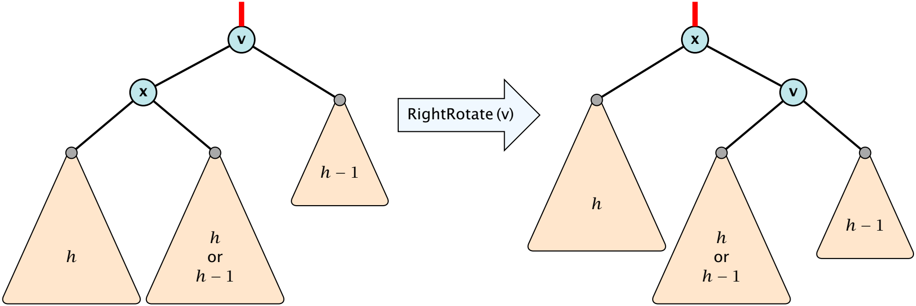
Case 1: balance[left[v]] $\in$ {0, 1}



If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

If the middle subtree has height $h - 1$ the whole tree has decreased its height from $h + 2$ to $h + 1$. We do continue the fix-up procedure as the balance at the root is zero.

Case 1: balance[left[v]] ∈ {0, 1}

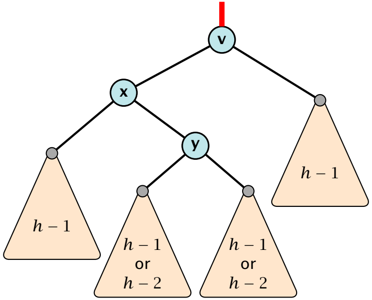


If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

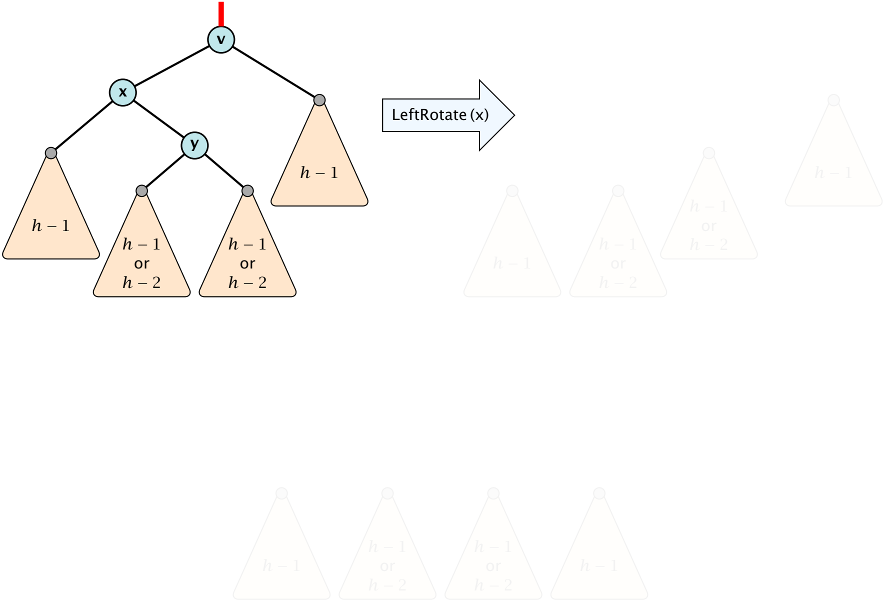
If the middle subtree has height $h - 1$ the whole tree has decreased its height from $h + 2$ to $h + 1$. We do continue the fix-up procedure as the balance at the root is zero.

Case 2: balance[left[v]] = −1

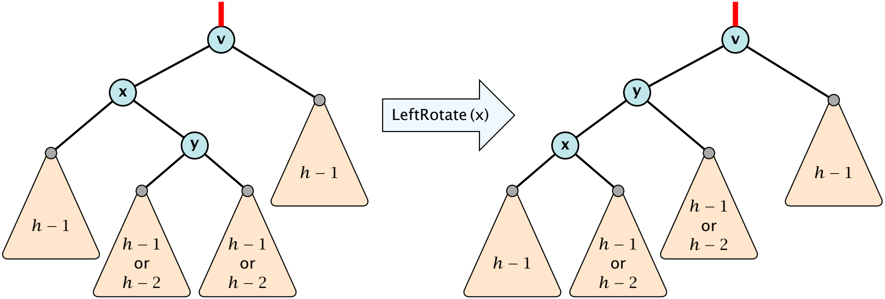
Case 2: balance[left[v]] = −1



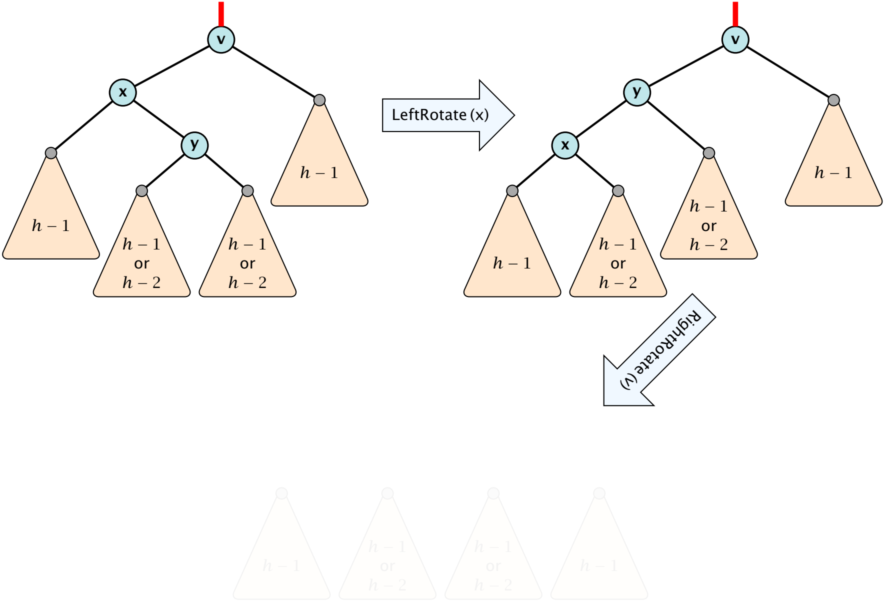
Case 2: balance[left[v]] = −1



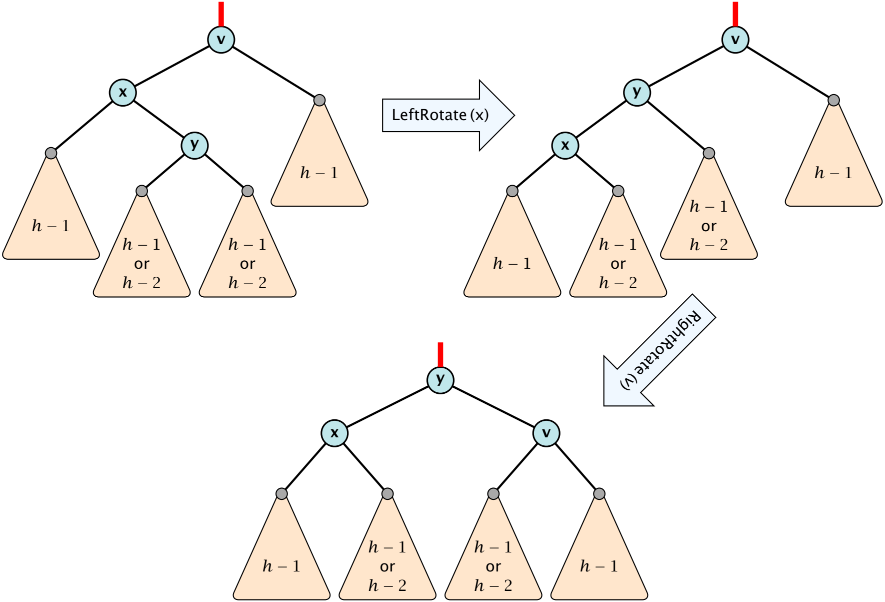
Case 2: balance[left[v]] = −1



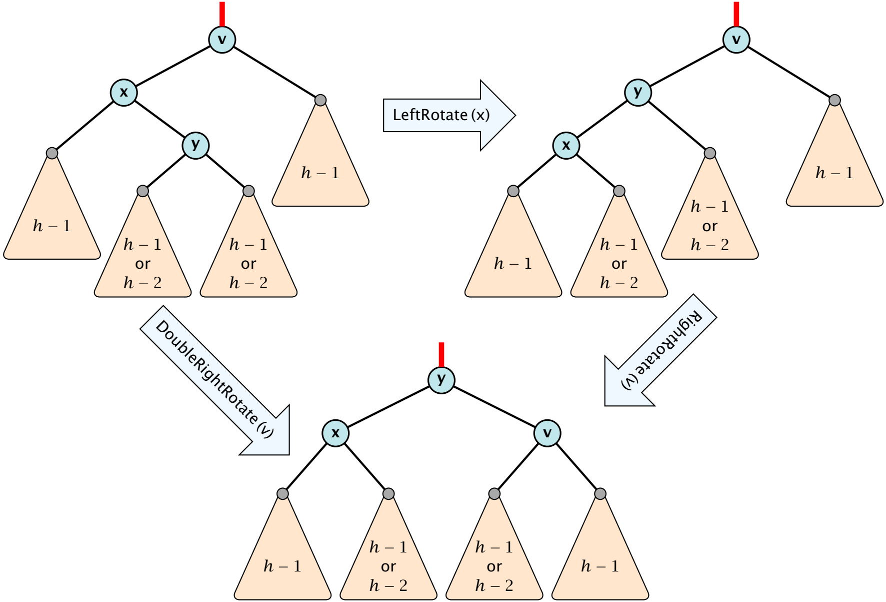
Case 2: balance[left[v]] = −1



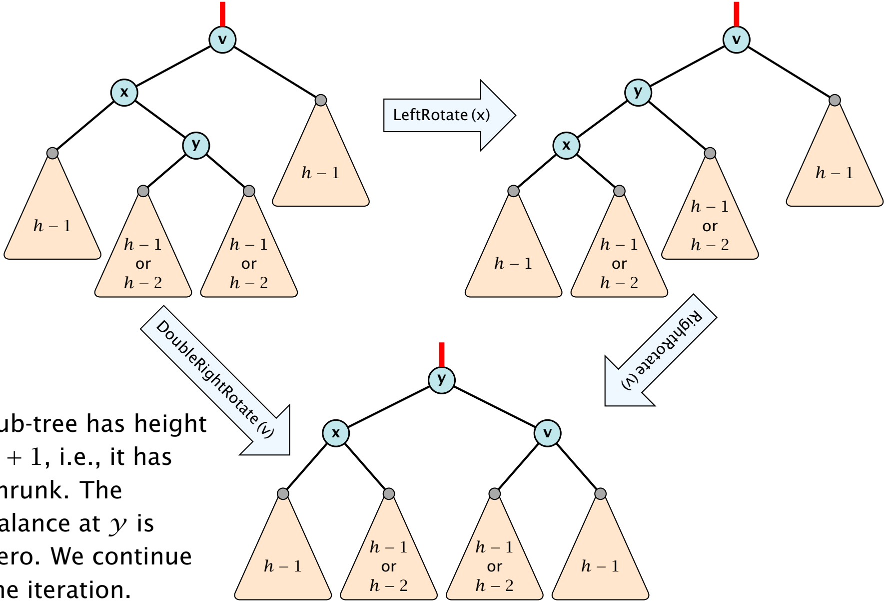
Case 2: balance[left[v]] = −1



Case 2: balance[left[v]] = −1



Case 2: balance[left[v]] = −1



Sub-tree has height $h+1$, i.e., it has shrunk. The balance at y is zero. We continue the iteration.

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a,b)-trees

Definition 17

For $b \geq 2a - 1$ an (a,b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

Definition 17

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.4 (a, b)-trees

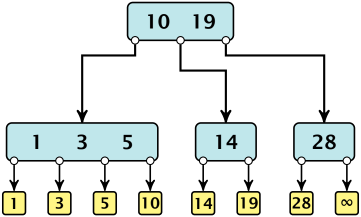
Each internal node v with $d(v)$ children stores $d - 1$ keys $k_1, \ldots, k_d - 1$. The i-th subtree of v fulfills

$$k_{i-1} < \text{ key in } i\text{-th sub-tree } \leq k_i \ ,$$

where we use $k_0 = -\infty$ and $k_d = \infty$.

7.4 (a, b)-trees

Example 18



7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

7.4 (a, b)-trees

Variants

- The dummy leaf element may not exist; this only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as only $1/2$-full (the requirement of a B-tree).

Lemma 19

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n+1 \leq b^h$
2. $\log_b(n+1) \leq h \leq \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

☐

Lemma 19

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n+1 \leq b^h$
2. $\log_b(n+1) \leq h \leq \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Lemma 19

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n+1 \leq b^h$
2. $\log_b(n+1) \leq h \leq \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Lemma 19

Let T be an (a, b)-tree for $n > 0$ elements (i.e., $n + 1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \le n + 1 \le b^h$
2. $\log_b(n + 1) \le h \le \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Lemma 19

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

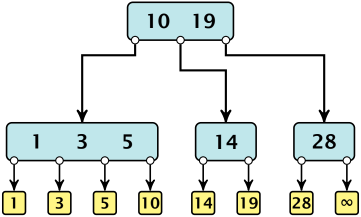
1. $2a^{h-1} \le n+1 \le b^h$
2. $\log_b(n+1) \le h \le \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

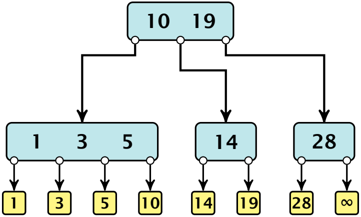
☐

Search



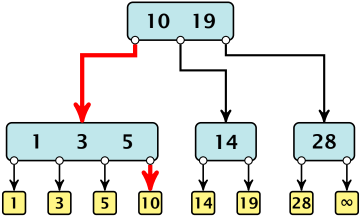
Search

Search(8)



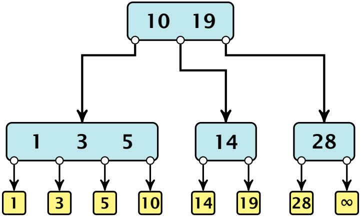
Search

Search(8)



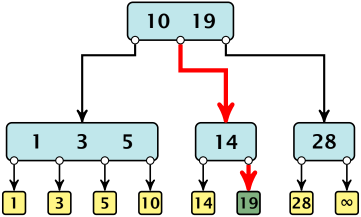
Search

Search(19)

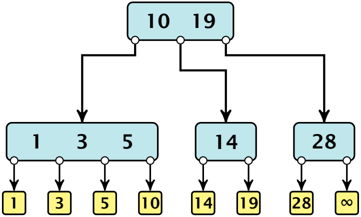


Search

Search(19)

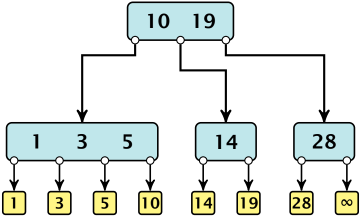


Search



The search is straightforward. It is only important that you need to go all the way to the leaf.

Search



The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$, if the individual nodes are organized as linear lists.

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\mathrm{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\mathrm{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\mathrm{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\mathrm{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

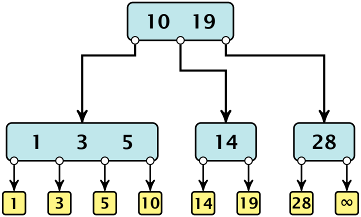
- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

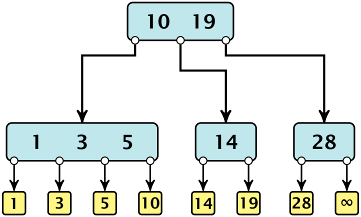
- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert



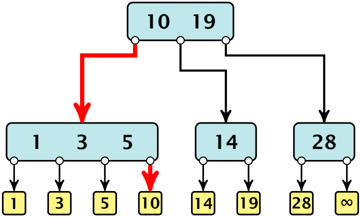
Insert

Insert(8)



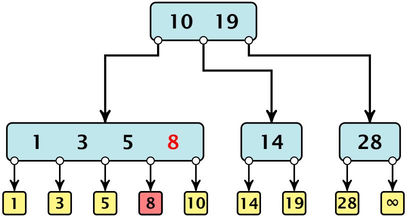
Insert

Insert(8)



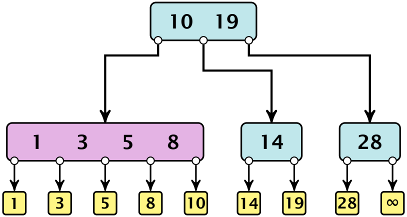
Insert

Insert(8)



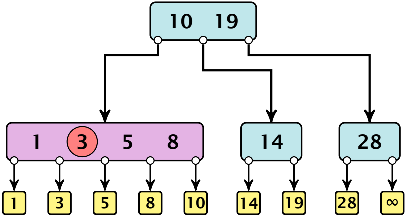
Insert

Insert(8)

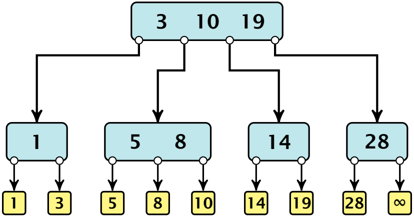


Insert

Insert(8)

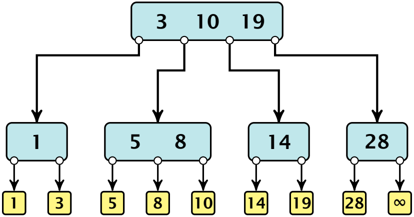


Insert



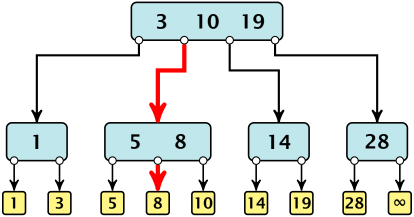
Insert

Insert(6)



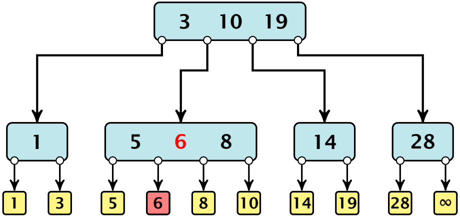
Insert

Insert(6)



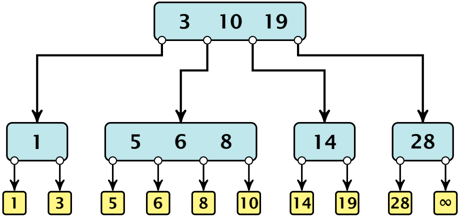
Insert

Insert(6)



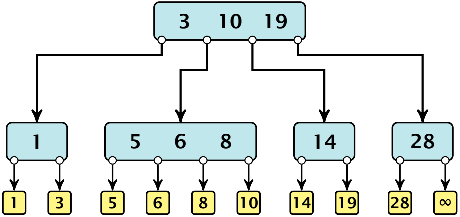
Insert

Insert(6)



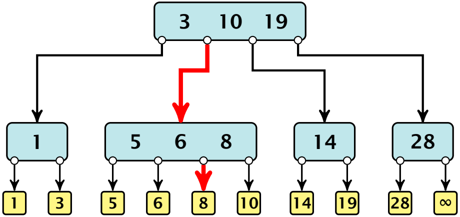
Insert

Insert(7)



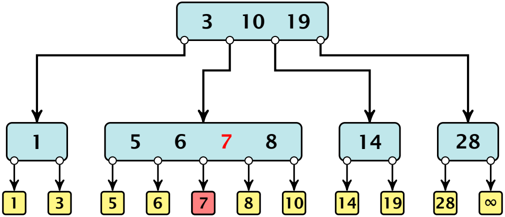
Insert

Insert(7)



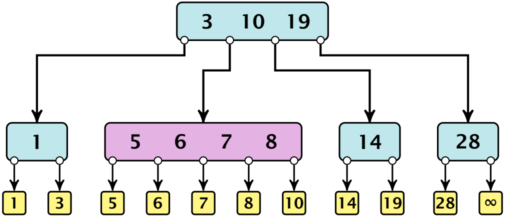
Insert

Insert(7)



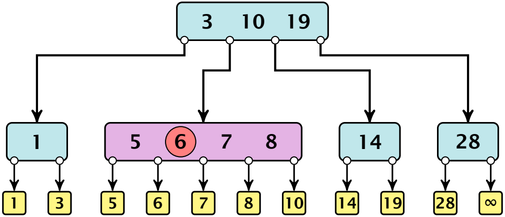
Insert

Insert(7)



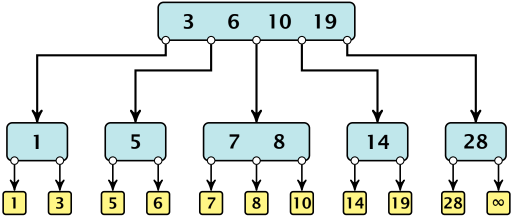
Insert

Insert(7)



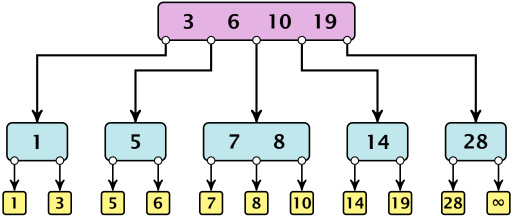
Insert

Insert(7)



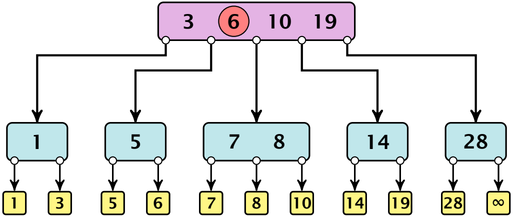
Insert

Insert(7)



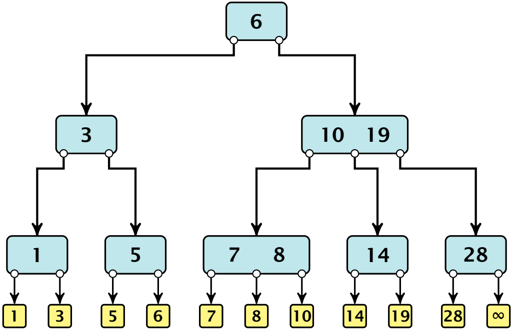
Insert

Insert(7)



Insert

Insert(7)



Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance'(v).

Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance'(v).

Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a-1$ perform Rebalance'(v).

Delete

Rebalance'(v):

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \leq b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance'(v):

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \leq b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance'(v):

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \leq b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance'(v):

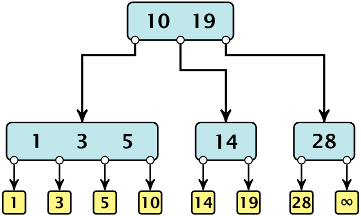
- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \leq b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance'(v):

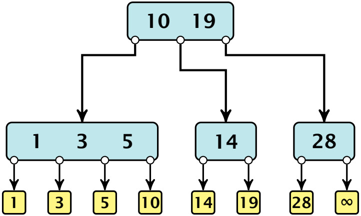
- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete



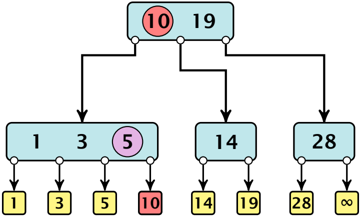
Delete

Delete(10)



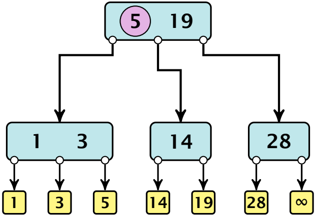
Delete

Delete(10)

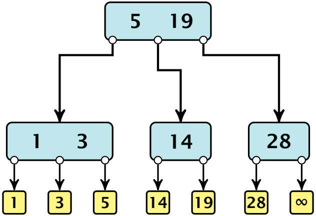


Delete

Delete(10)

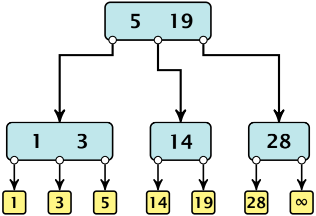


Delete



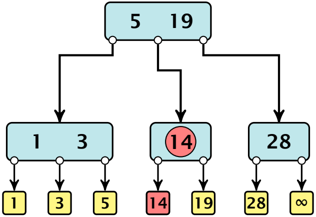
Delete

Delete(14)



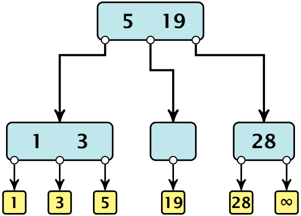
Delete

Delete(14)



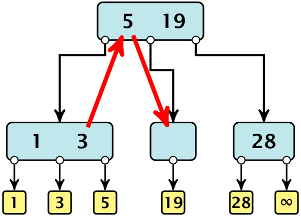
Delete

Delete(14)



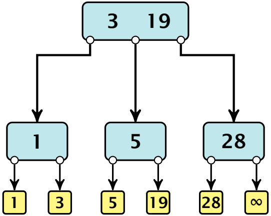
Delete

Delete(14)

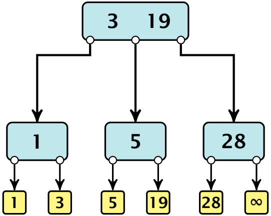


Delete

Delete(14)

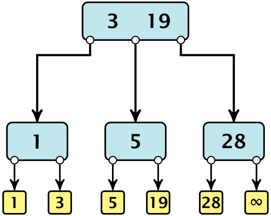


Delete



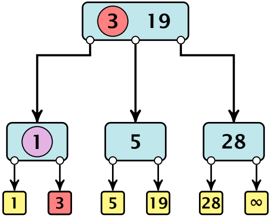
Delete

Delete(3)



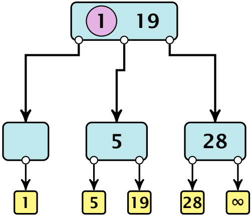
Delete

Delete(3)



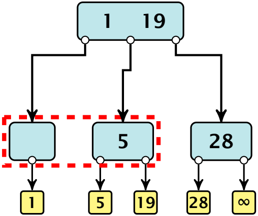
Delete

Delete(3)



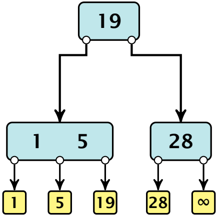
Delete

Delete(3)

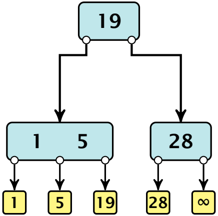


Delete

Delete(3)

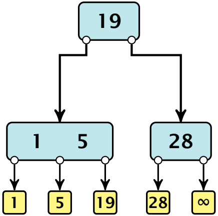


Delete



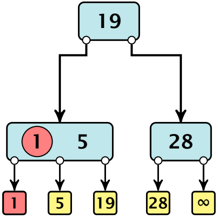
Delete

Delete(1)



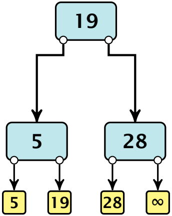
Delete

Delete(1)

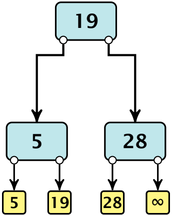


Delete

Delete(1)

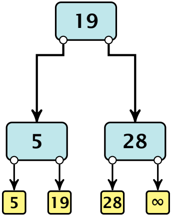


Delete



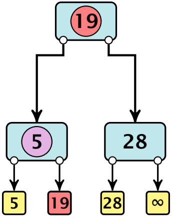
Delete

Delete(19)



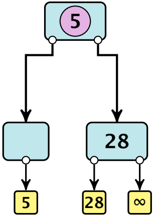
Delete

Delete(19)



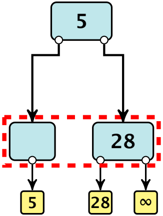
Delete

Delete(19)



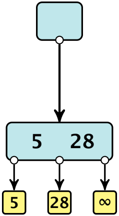
Delete

Delete(19)



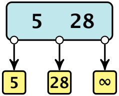
Delete

Delete(19)



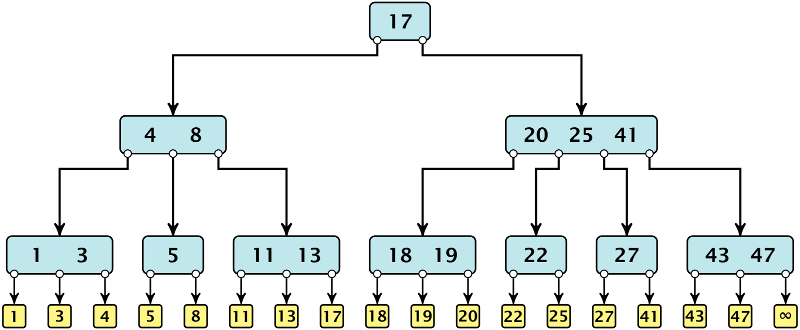
Delete

Delete(19)



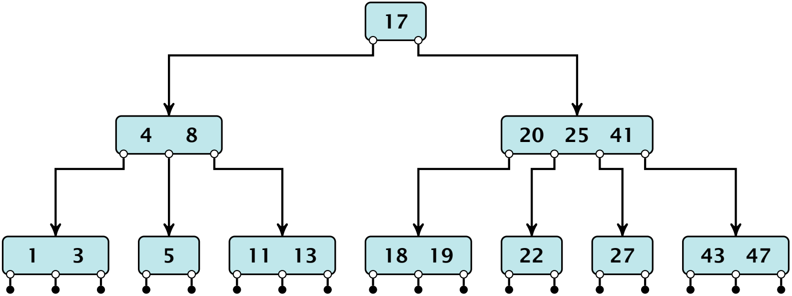
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



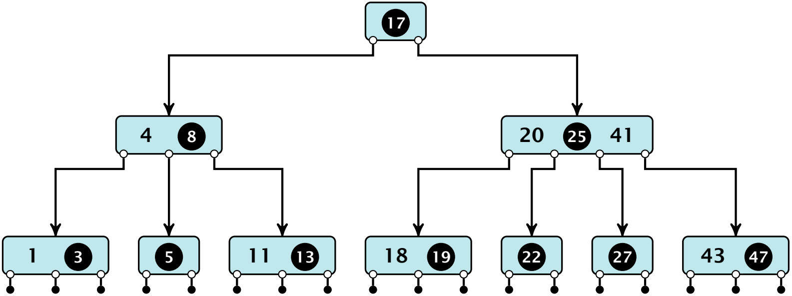
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



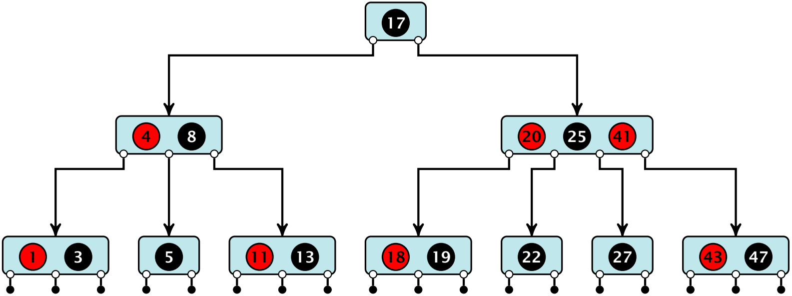
$(2, 4)$-trees and red black trees

There is a close relation between red-black trees and $(2, 4)$-trees:



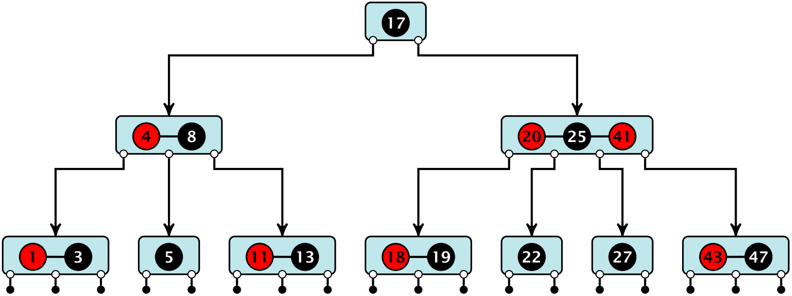
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



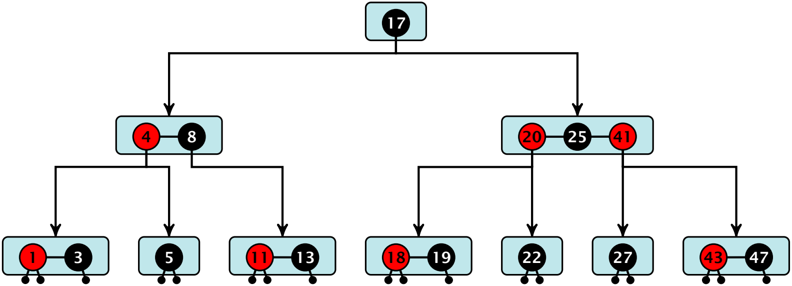
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



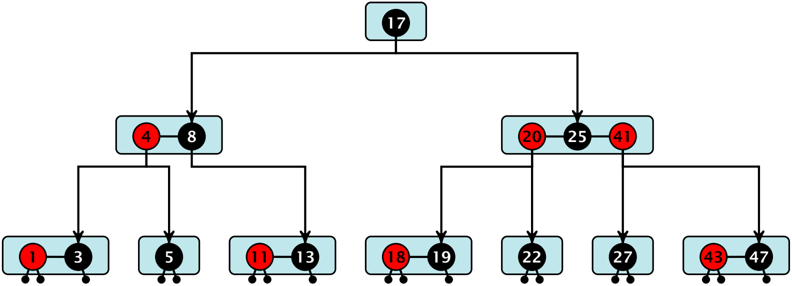
(2, 4)-trees and red black trees

There is a close relation between red-black trees and $(2, 4)$-trees:



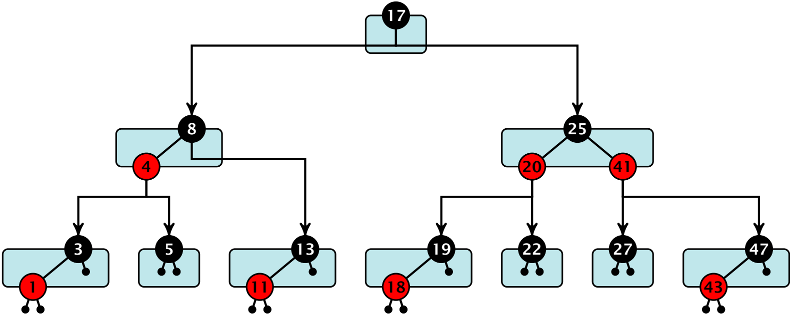
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



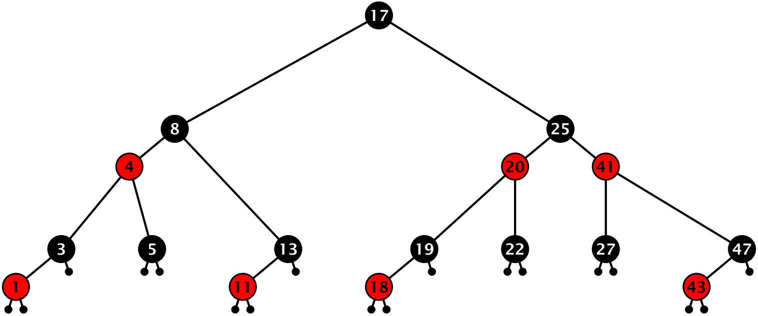
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



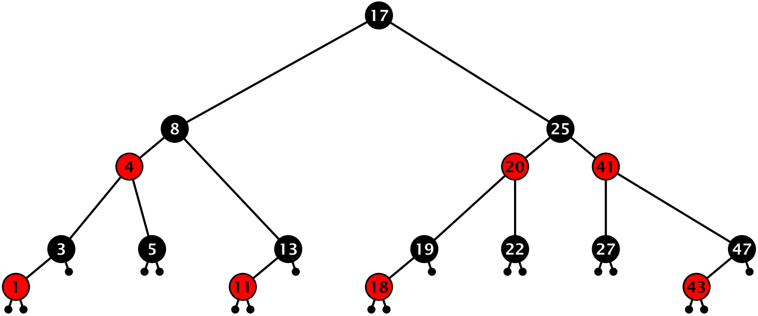
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same $(2,4)$-tree.

7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$

7.5 Skip Lists

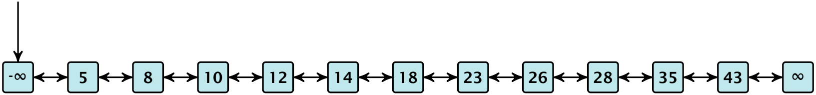
Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$

7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

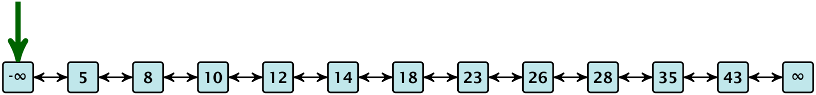
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

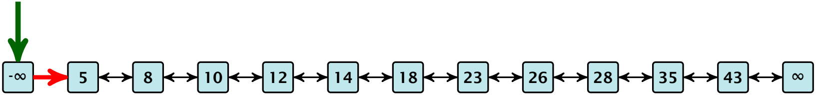
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

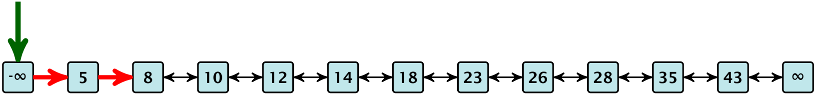
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

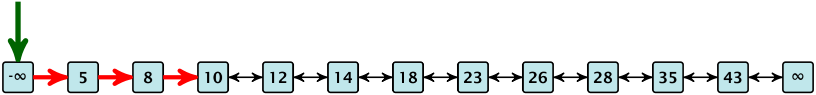
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

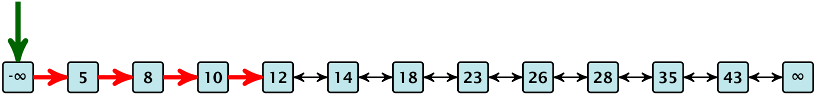
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

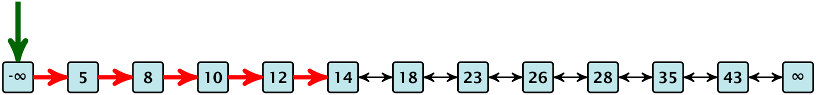
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

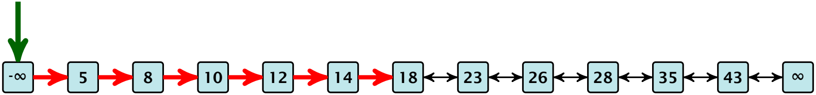
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

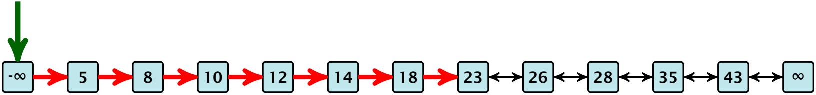
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

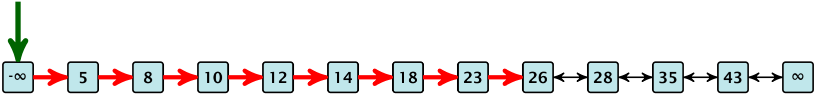
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(1)$



7.5 Skip Lists

How can we improve the search-operation?

7.5 Skip Lists

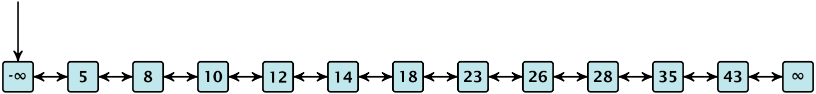
How can we improve the search-operation?

Add an express lane:

7.5 Skip Lists

How can we improve the search-operation?

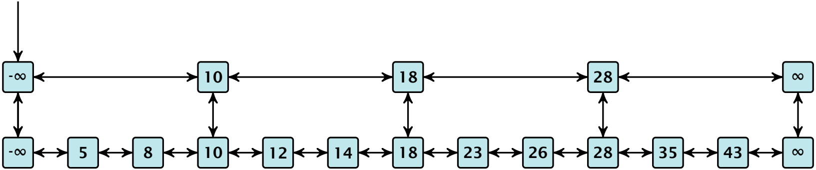
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

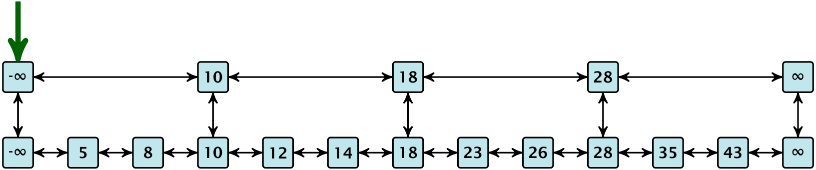
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

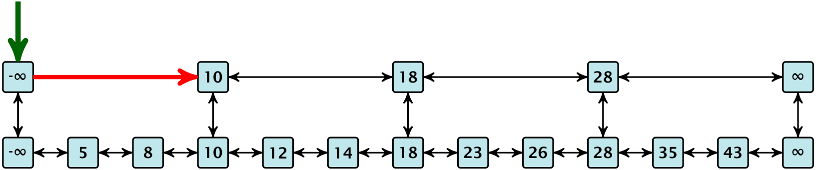
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

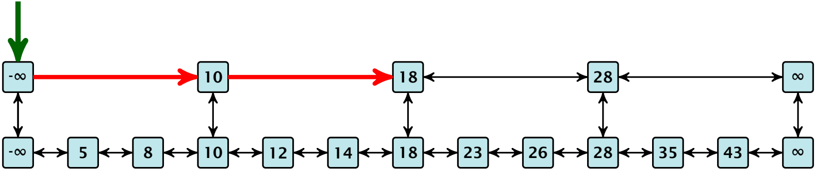
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

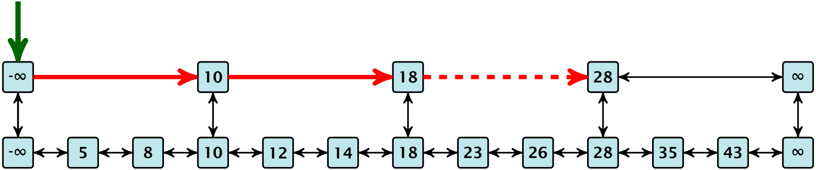
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

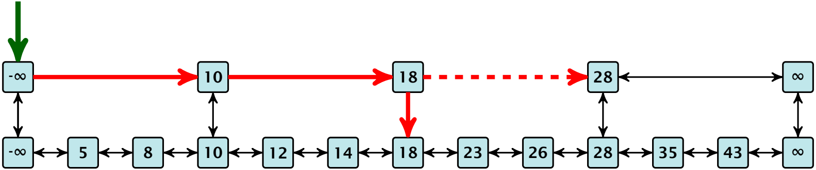
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

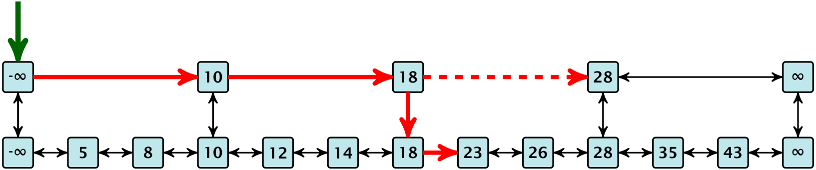
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

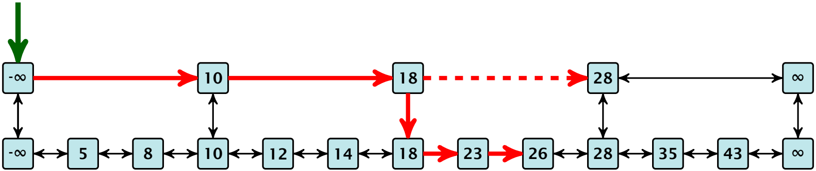
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

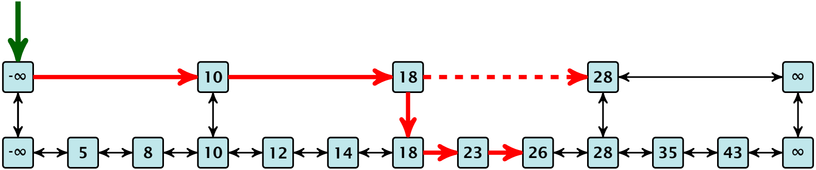
Add an express lane:



7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:

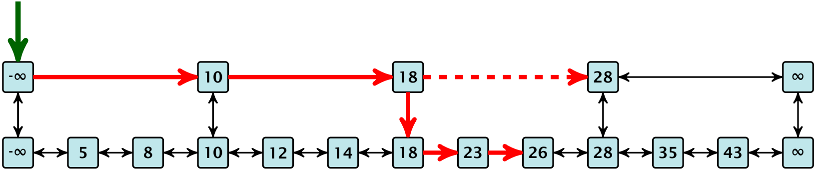


Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



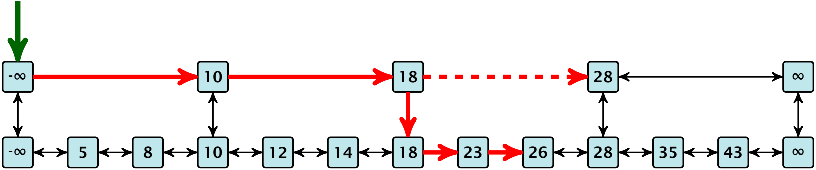
Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

7.5 Skip Lists

How can we improve the search-operation?

Add an express lane:



Let $|L_1|$ denote the number of elements in the “express lane”, and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

Choose $|L_1| = \sqrt{n}$. Then search time $\Theta(\sqrt{n})$.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \dots, L_k$)

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k + 1$ lists $L_0, \dots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \dots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \dots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- . . .

7.5 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \dots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- . . .
- At most $|L_k| + \sum_{i=1}^{k} \frac{L_{i-1}}{L_i} + 3(k+1)$ steps.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$. Choose

$$r = \sqrt[k+1]{n} \qquad \Longrightarrow \qquad \text{time: } \mathcal{O}(k \sqrt[k+1]{n})$$

7.5 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$. Choose

$$r = \sqrt[k+1]{n} \qquad \Longrightarrow \qquad \text{time: } \mathcal{O}(k\sqrt[k+1]{n})$$

Choosing $k = \Theta(\log k)$ gives a logarithmic running time.

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

7.5 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

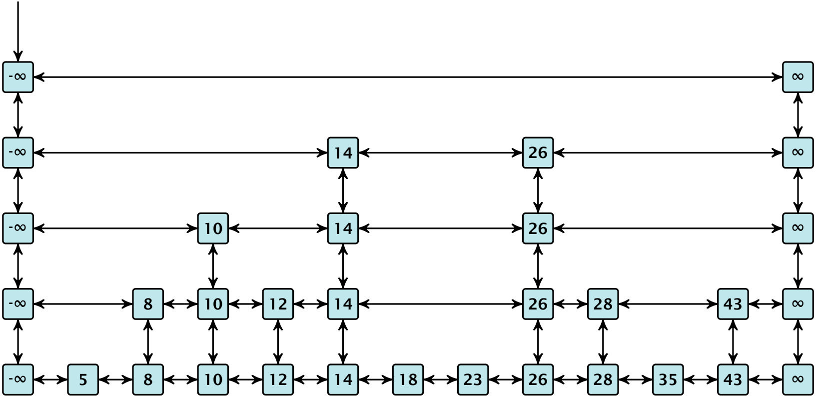
Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists in actually appears in.

The time for both operation is dominated by the search time.

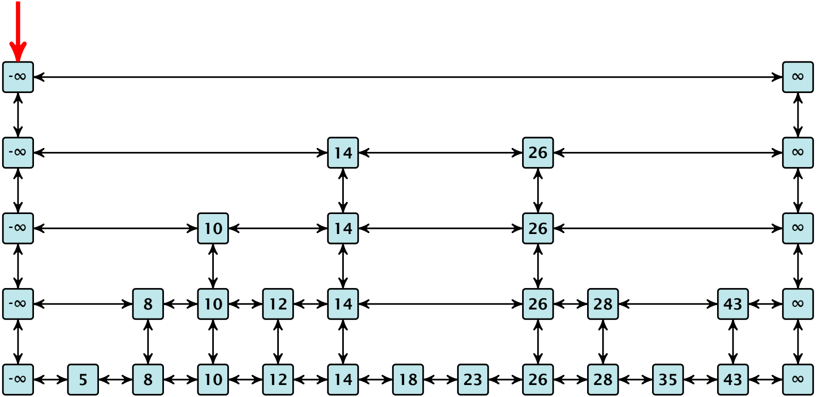
Skip Lists

Insert (35):



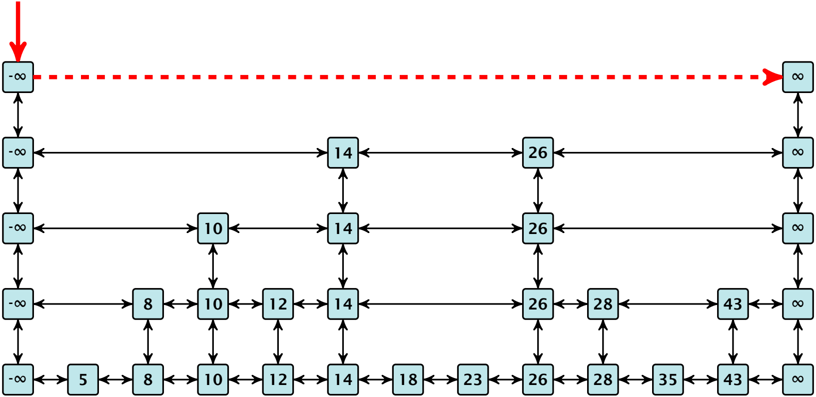
Skip Lists

Insert (35):



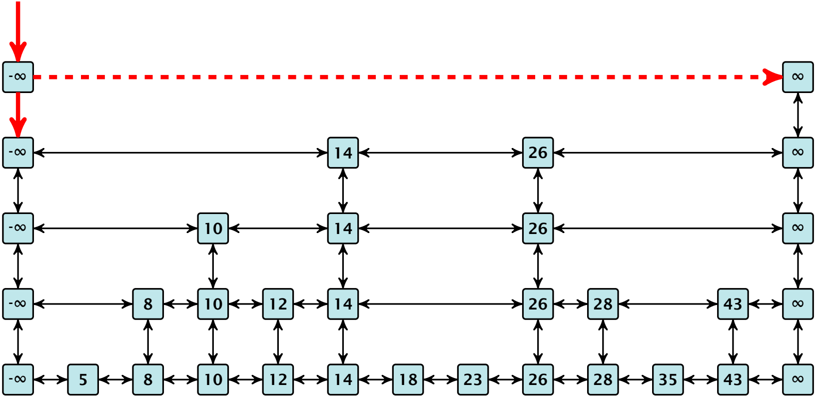
Skip Lists

Insert (35):



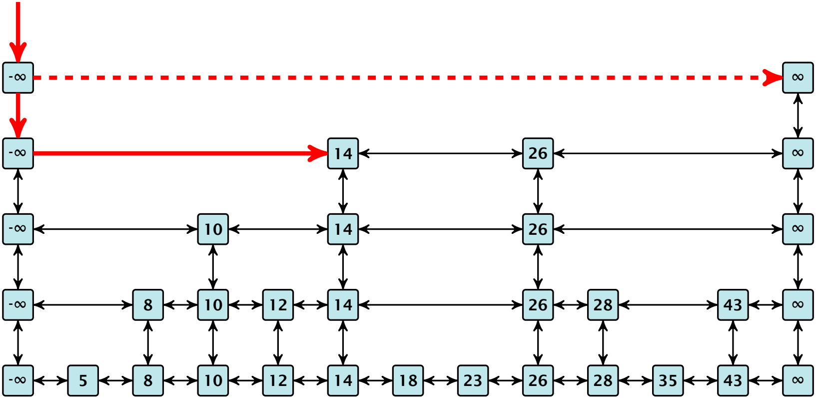
Skip Lists

Insert (35):



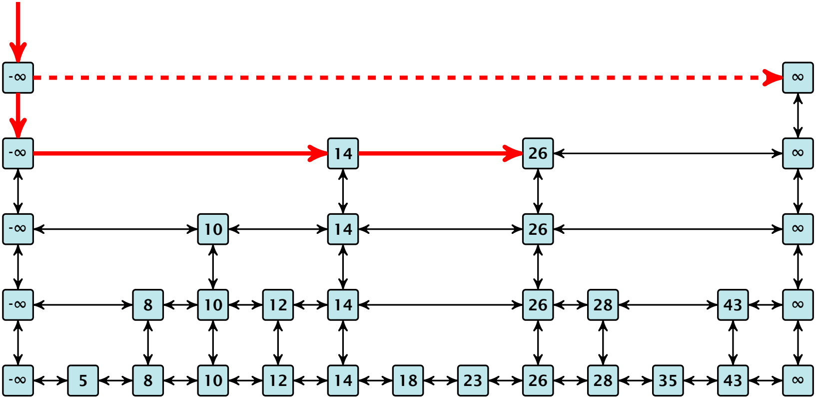
Skip Lists

Insert (35):



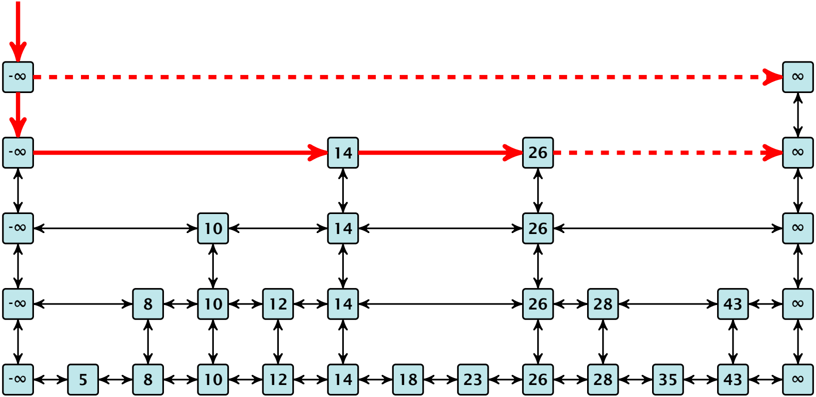
Skip Lists

Insert (35):



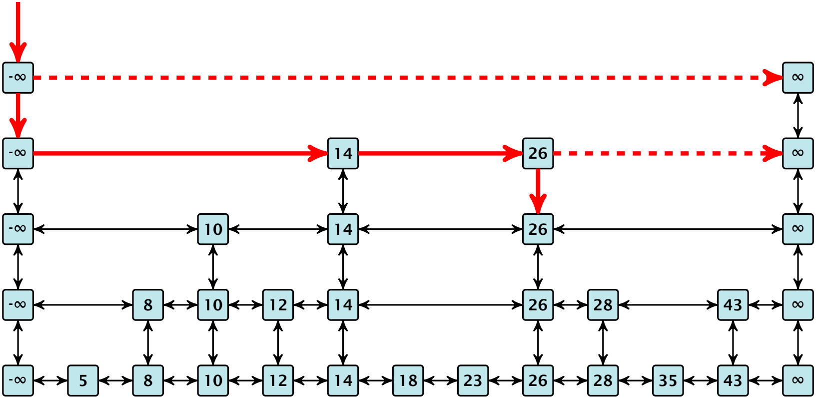
Skip Lists

Insert (35):



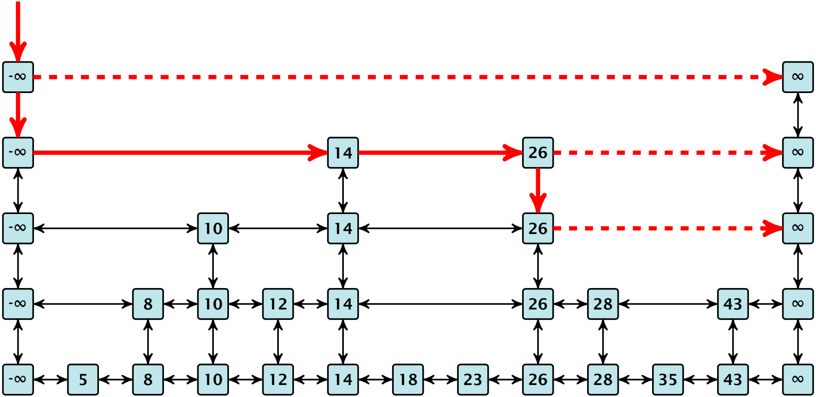
Skip Lists

Insert (35):



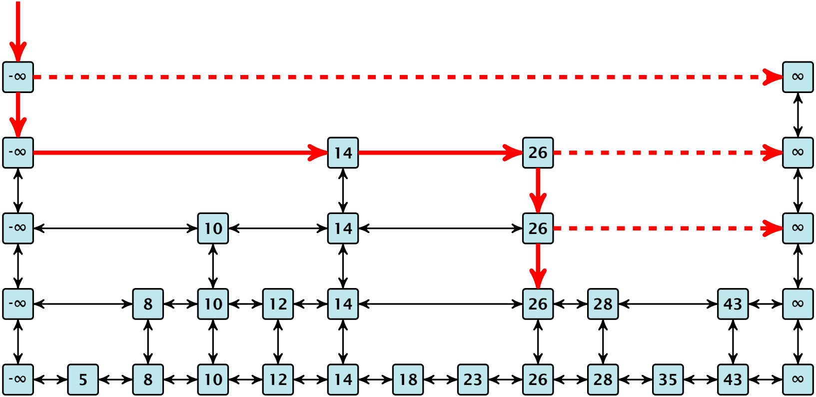
Skip Lists

Insert (35):



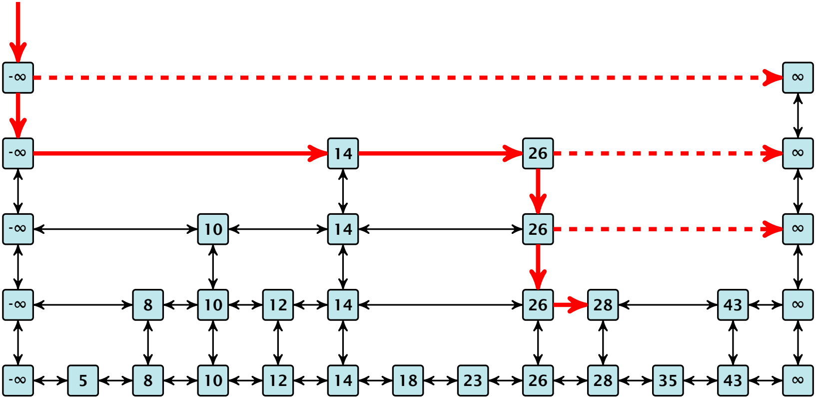
Skip Lists

Insert (35):



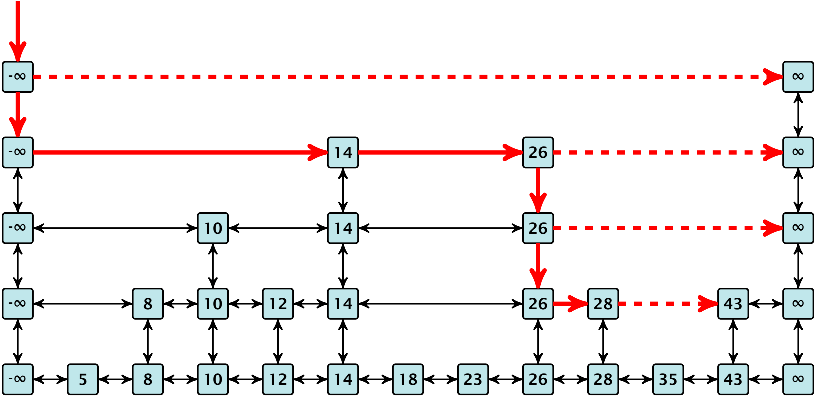
Skip Lists

Insert (35):



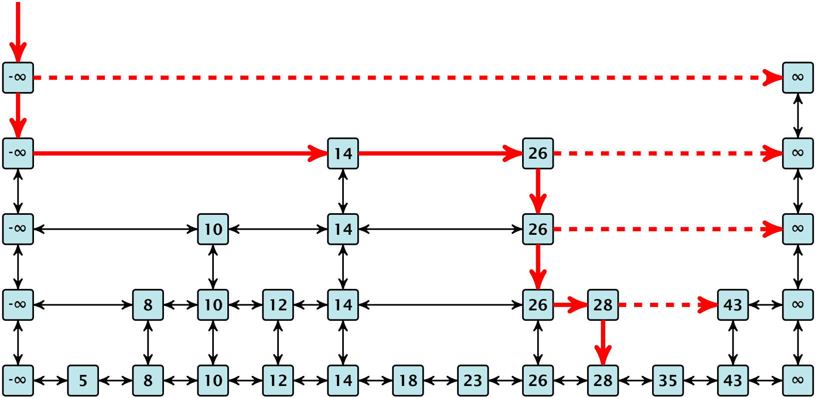
Skip Lists

Insert (35):



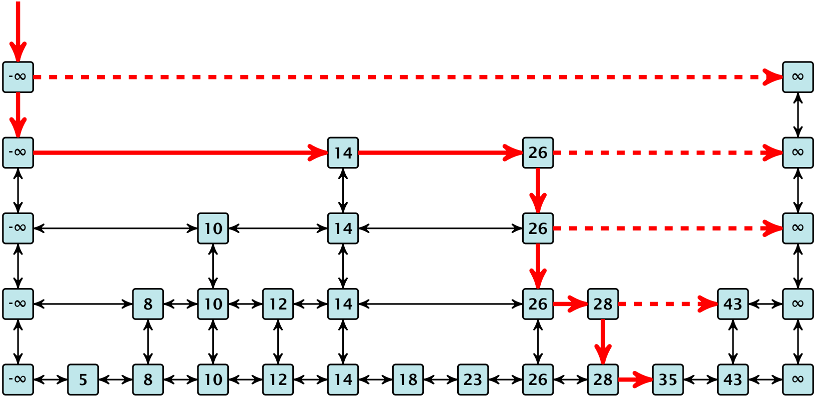
Skip Lists

Insert (35):



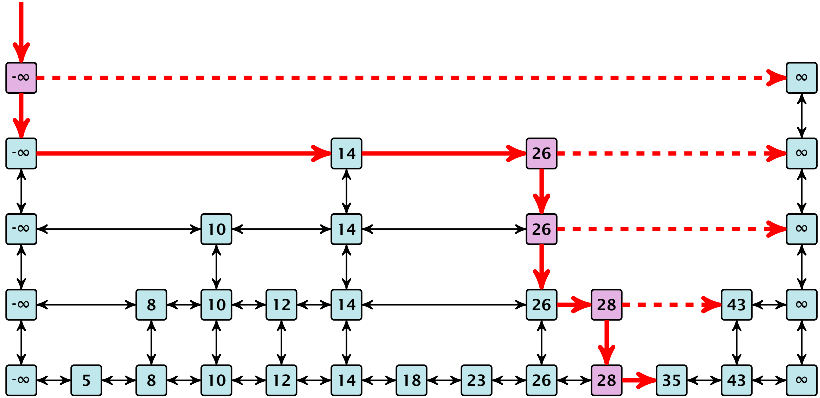
Skip Lists

Insert (35):



Skip Lists

Insert (35):



7.5 Skip Lists

Lemma 20

A search (and, hence, also insert and delete) in a skip list with n elements takes time $\mathcal{O}(logn)$ with high probability (w. h. p.).

This means for any constant α the search takes time $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Note that the constant in the $\mathcal{O}$-notation may depend on α.

7.5 Skip Lists

Lemma 20

A search (and, hence, also insert and delete) in a skip list with n *elements takes time* $\mathcal{O}(logn)$ *with high probability (w. h. p.).*

This means for any constant α *the search takes time* $\mathcal{O}(\log n)$ *with probability at least* $1 - \frac{1}{n^\alpha}$*.*

Note that the constant in the $\mathcal{O}$*-notation may depend on* α*.*

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probabilityx that all E_i hold is at least

$$\Pr[E_1 \wedge \dots \wedge E_\ell]$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probabilityx that all E_i hold is at least

$$\Pr[E_1 \wedge \cdots \wedge E_\ell] = 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell]$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probabilityx that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\le 1 - n^c \cdot n^{-\alpha}\end{aligned}$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probabilityx that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\leq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probabilityx that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\le 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

This means $\Pr[E_1 \wedge \cdots \wedge E_\ell]$ holds with high probability.

Skip Lists

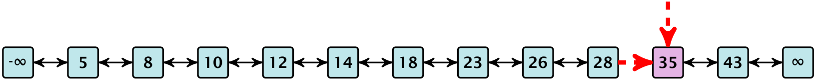
Backward analysis:

Skip Lists

Backward analysis:

Skip Lists

Backward analysis:

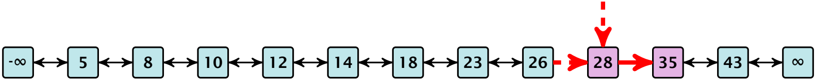


Skip Lists

Backward analysis:

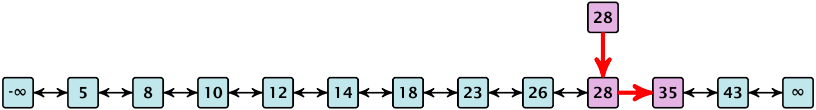
Skip Lists

Backward analysis:



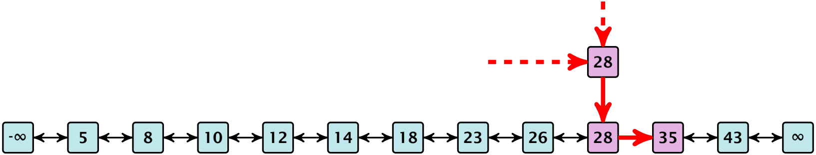
Skip Lists

Backward analysis:



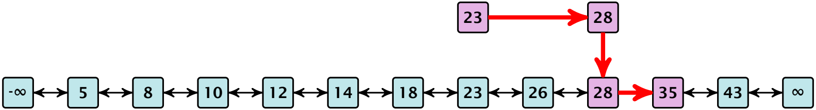
Skip Lists

Backward analysis:



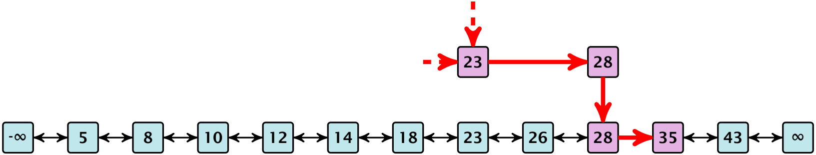
Skip Lists

Backward analysis:



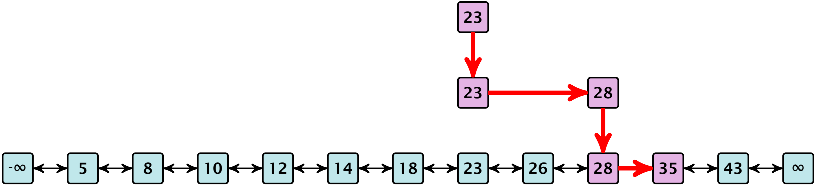
Skip Lists

Backward analysis:



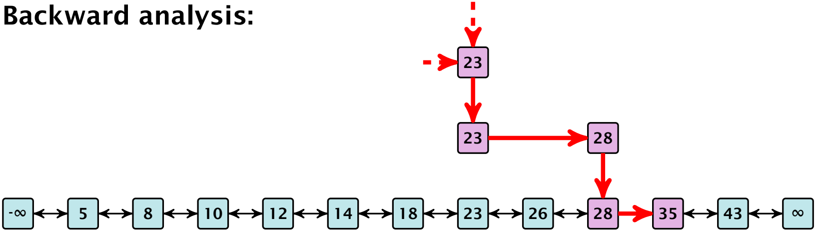
Skip Lists

Backward analysis:



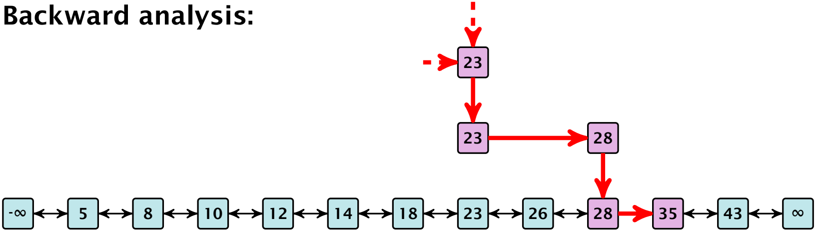
Skip Lists

Backward analysis:



Skip Lists

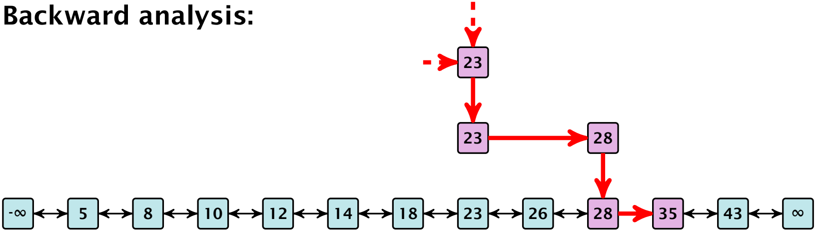
Backward analysis:



At each point the path goes up with probability $1/2$ and left with probability $1/2$.

Skip Lists

Backward analysis:



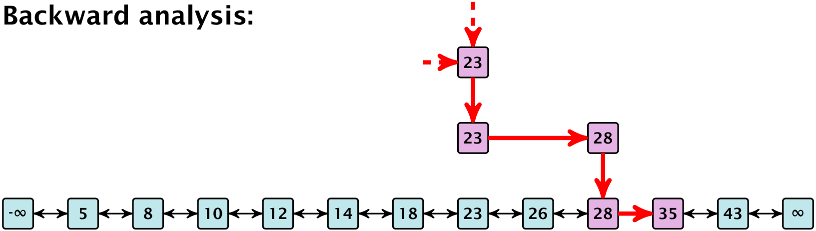
At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A “long” search path must also go very high.

Skip Lists

Backward analysis:



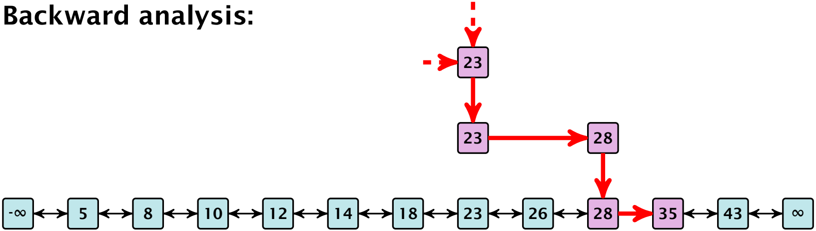
At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A “long” search path must also go very high.
- There are no elements in high lists.

Skip Lists

Backward analysis:



At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A "long" search path must also go very high.
- There are no elements in high lists.

From this it follows that w.h.p. there are no long paths.

7.5 Skip Lists

7.5 Skip Lists

Let $E_{z,k}$ denote the event that a search path is of length z (number of edges) but does not visit a list above L_k.

7.5 Skip Lists

Let $E_{z,k}$ denote the event that a search path is of length z (number of edges) but does not visit a list above L_k.

In particular, this means that during the construction in the backward analysis we see at most k heads (i.e., coin flips that tell you to go up) in z trials.

7.5 Skip Lists

$$\Pr[E_{z,k}]$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

7.5 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\le \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \leq \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\leq \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha} \leq n^{-\alpha}$$

7.5 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k (2^{-\beta})^k \cdot n^{-\alpha} \leq \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\leq \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha} \leq n^{-\alpha}$$

for $\alpha \geq 1$.

7.5 Skip Lists

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \quad .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\Pr[\text{search requires } z \text{ steps}]$$

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \enspace .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\Pr[\text{search requires } z \text{ steps}] \leq \Pr[E_{z,k}] + \Pr[A_{k+1}]$$

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\begin{aligned}\Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)}\end{aligned}$$

7.5 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\begin{aligned}\Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)}\end{aligned}$$

This means, the search requires at most z steps, w. h. p.

7.6 Augmenting Data Structures

Suppose you want to develop a data structure with:

- **Insert(x):** insert element x.
- **Search(k):** search for element with key k.
- **Delete(x):** delete element referenced by pointer x.
- **find-by-rank(ℓ):** return the k-th element; return “error” if the data-structure contains less than k elements.

Augment an existing data-structure instead of developing a new one.

7.6 Augmenting Data Structures

Suppose you want to develop a data structure with:

- **Insert(x):** insert element x.
- **Search(k):** search for element with key k.
- **Delete(x):** delete element referenced by pointer x.
- **find-by-rank(ℓ):** return the k-th element; return “error” if the data-structure contains less than k elements.

Augment an existing data-structure instead of developing a new one.

7.6 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.6 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.6 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.6 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.6 Augmenting Data Structures

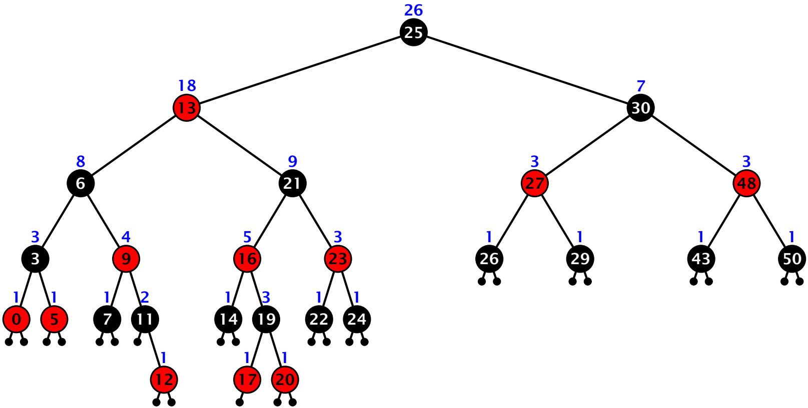
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

4. How does find-by-rank work?
   Find-by-rank(k) ≔ Select(root, k) with

Algorithm 15 Select(x, i)

```
if $x = \text{null}$ then return error
if $\text{left}[x] \neq \text{null}$ then $r \leftarrow \text{left}[x].\text{size} + 1$ else $r \leftarrow 1$
if $i = r$ then return $x$
if $i < r$ then
    return Select$(\text{left}[x], i)$
else
    return Select$(\text{right}[x], i - r)$
```

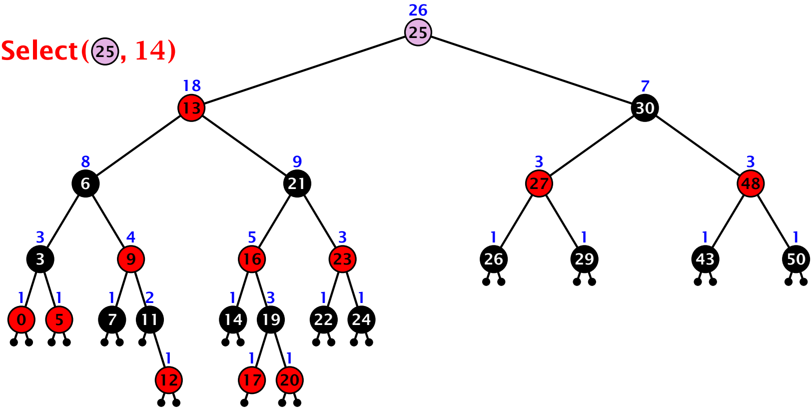
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

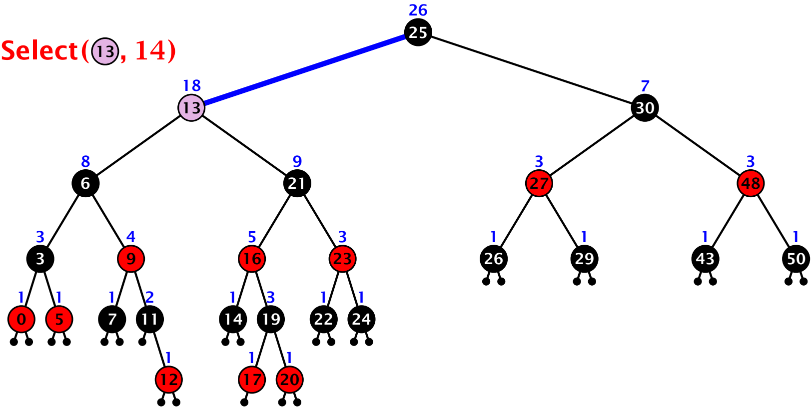
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

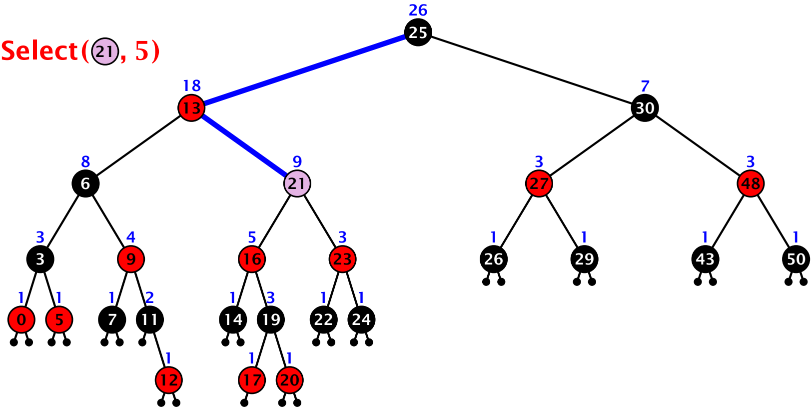
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

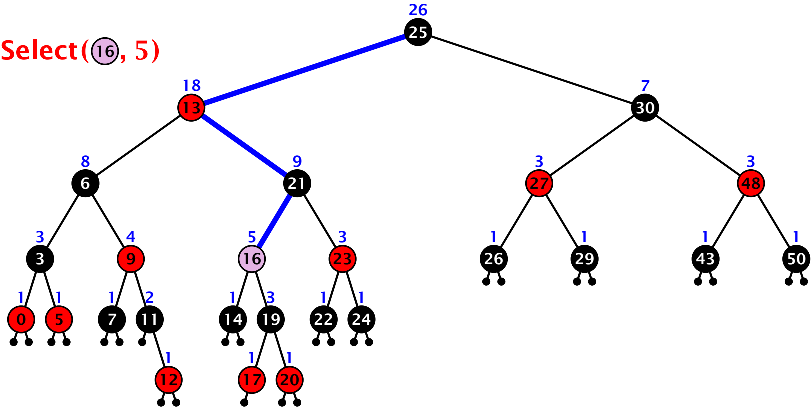
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

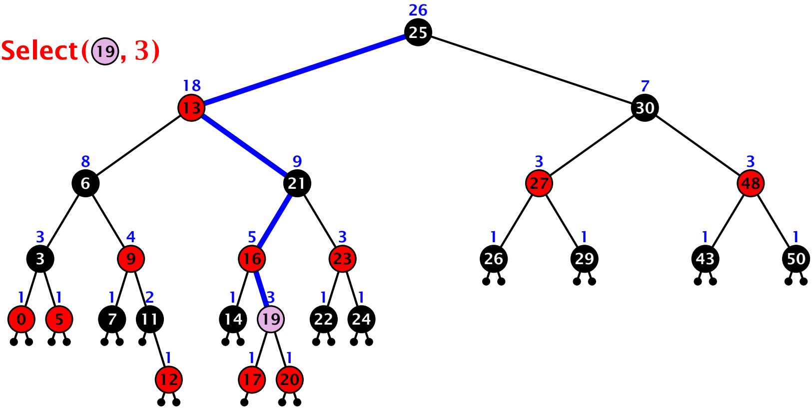
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

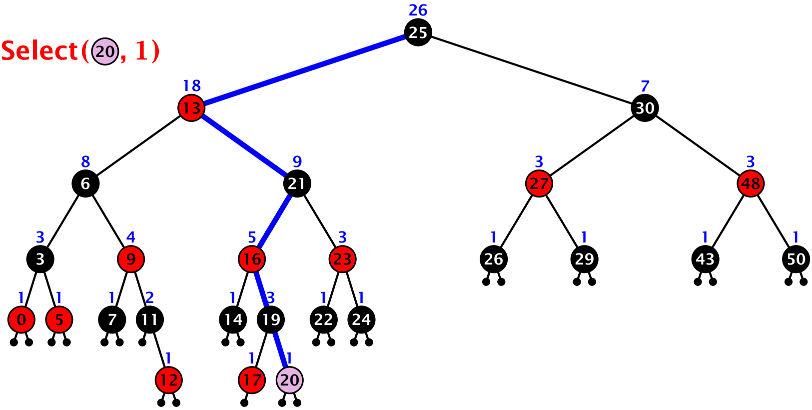
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.6 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

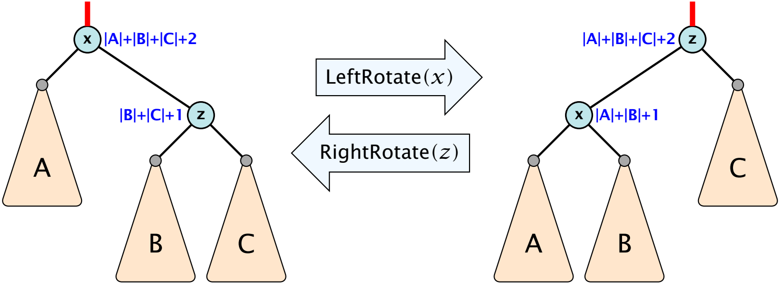
Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes x and z are the only nodes changing their size-fields.

The new size-fields can be computed locally from the size-fields of the children.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7.7 Hashing

Dictionary:

- **S.insert(x)**: Insert an element x.
- **S.delete(x)**: Delete the element pointed to by x.
- **S.search(k)**: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7.7 Hashing

Dictionary:

- $\boldsymbol{S}$**.insert**$(\boldsymbol{x})$: Insert an element x.
- $\boldsymbol{S}$**.delete**$(\boldsymbol{x})$: Delete the element pointed to by x.
- $\boldsymbol{S}$**.search**$(\boldsymbol{k})$: Return a pointer to an element e with $\mathrm{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \le n$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \rightarrow [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0,\dots,n-1]$ hash-table.
- Hash function $h : U \to [0,\dots,n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

Definitions:

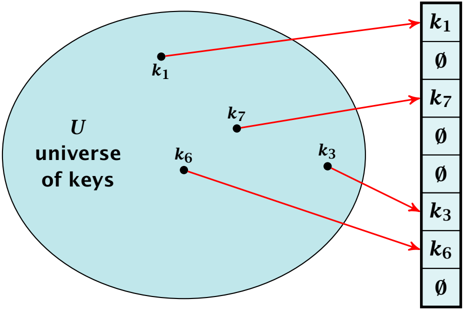
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq n$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7.7 Hashing

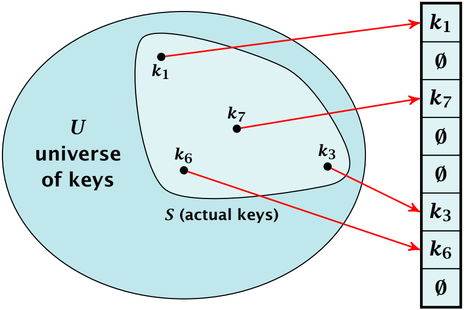
Ideally the hash function maps all keys to different memory locations.



This special case is known as Direct Addressing. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

7.7 Hashing

Suppose that we know the set S of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.



Such a hash function h is called a perfect hash function for set S.

7.7 Hashing

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions

Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7.7 Hashing

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions

Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7.7 Hashing

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions
Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7.7 Hashing

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already once $|S| \geq \omega(\sqrt{n})$.

Lemma 21

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2}} \approx 1 - e^{-\frac{m^2}{2n}} .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \ldots, n-1]$.

7.7 Hashing

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already once $|S| \geq \omega(\sqrt{n})$.

Lemma 21

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2}} \approx 1 - e^{-\frac{m^2}{2n}} \quad .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \ldots, n-1]$.

7.7 Hashing

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already once $|S| \geq \omega(\sqrt{n})$.

Lemma 21

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2}} \approx 1 - e^{-\frac{m^2}{2n}} \quad .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \ldots, n-1]$.

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}]$$

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n - \ell + 1}{n}$$

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Big(1 - \frac{j}{n}\Big)$$

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\begin{aligned}\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Bigl(1 - \frac{j}{n}\Bigr) \\ &\le \prod_{j=0}^{m-1} e^{-j/n}\end{aligned}$$

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Big(1 - \frac{j}{n}\Big)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}}$$

7.7 Hashing

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$
\begin{aligned}
\Pr[A_{m,n}] &= \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Bigl(1 - \frac{j}{n}\Bigr) \\
&\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .
\end{aligned}
$$

7.7 Hashing

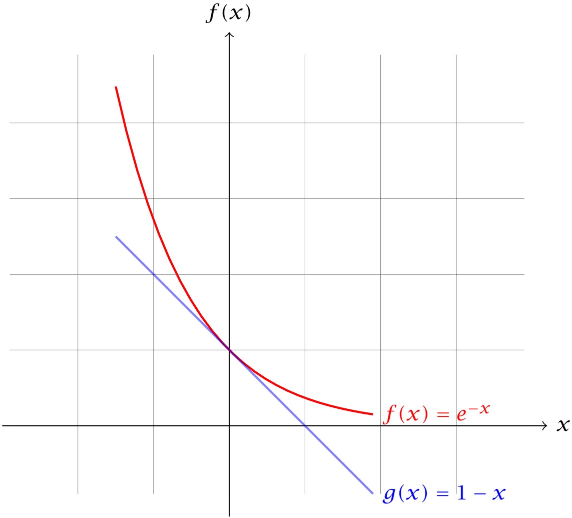
Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\Big(1-\frac{j}{n}\Big)$$

$$\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

Here the first equality follows since the ℓ-th element that is hashed has a probability of $\frac{n-\ell+1}{n}$ to not generate a collision under the condition that the previous elements did not induce collisions. □



The inequality $1 - x \leq e^{-x}$ is derived by stopping the tayler-expansion of e^{-x} after the second term.

Resolving Collisions

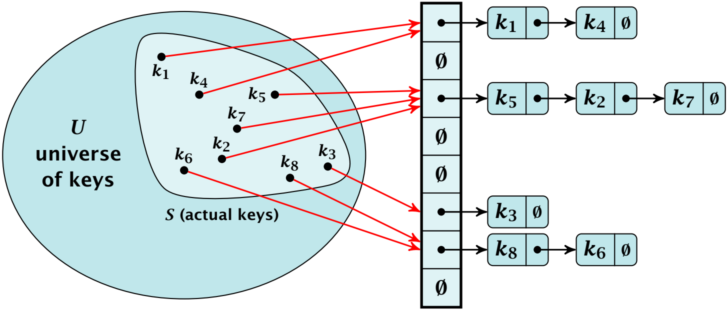
The methods for dealing with collisions can be classified into the two main types

- **open addressing**, aka. closed hashing
- **hashing with chaining.** aka. closed addressing, open hashing.

Hashing with Chaining

Arrange elements that map to the same position in a linear list.

- Access: compute $h(x)$ and search list for $\mathrm{key}[x]$.
- Insert: insert at the front of the list.



7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7.7 Hashing

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a successful search when using A;
- A^- denotes the average time for an unsuccessful search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if A is the collision resolving strategy "Hashing with Chaining" we have

$$A^{-} = 1 + \alpha\ .$$

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if A is the collision resolving strategy “Hashing with Chaining” we have

$$A^- = 1 + \alpha \ .$$

Note that this result does not depend on the hash-function that is used.

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \overbrace{\sum_{j=i+1}^{m} X_{ij}}^{\text{keys before } k_i}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do not choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the event that i and j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

cost for key k_i

Hashing with Chaining

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right]$$

Hashing with Chaining

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right]=\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)$$

Hashing with Chaining

$$\begin{aligned}\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\ &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\end{aligned}$$

Hashing with Chaining

$$\begin{aligned}\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\ &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\ &= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\end{aligned}$$

Hashing with Chaining

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)
\end{aligned}$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)\\
&= 1+\frac{m-1}{2n} = 1+\frac{\alpha}{2}-\frac{\alpha}{2m}\ .
\end{aligned}
$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)\\
&= 1+\frac{m-1}{2n} = 1+\frac{\alpha}{2}-\frac{\alpha}{2m}\ .
\end{aligned}
$$

Hence, the expected cost for a successful search is $A^+ \le 1+\frac{\alpha}{2}$.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,...,$h(k,n-1)$ form a permutation of $0,\ldots,n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,…,$h(k,n-1)$ form a permutation of $0,\ldots,n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ….

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j-th step. The values $h(k, 0)$,...,$h(k, n-1)$ form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1)$, $h(k, 2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j-th step. The values $h(k, 0)$,...,$h(k, n-1)$ form a permutation of $0, \dots, n-1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1)$, $h(k, 2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,...,$h(k,n-1)$ form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

Choices for $h(k,j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k,j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k,j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n; for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k,j)$:

- $h(k,i) = h(k) + i \mod n$. Linear probing.
- $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$. Quadratic probing.
- $h(k,i) = h_1(k) + i h_2(k) \mod n$. Double hashing.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n; for quadratic probing c_1 and c_2 have to be chosen carefully).

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 22

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^- \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 22

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^- \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Quadratic Probing

- Not as cache-efficient as Linear Probing.
- Secondary clustering: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 23

Let Q be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Quadratic Probing

- Not as cache-efficient as Linear Probing.
- Secondary clustering: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 23

Let Q be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Double Hashing

- Any probe into the hash-table usually creates a cash-miss.

Lemma 24

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

$$D^- \approx \frac{1}{1-\alpha}$$

Double Hashing

- Any probe into the hash-table usually creates a cash-miss.

Lemma 24

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

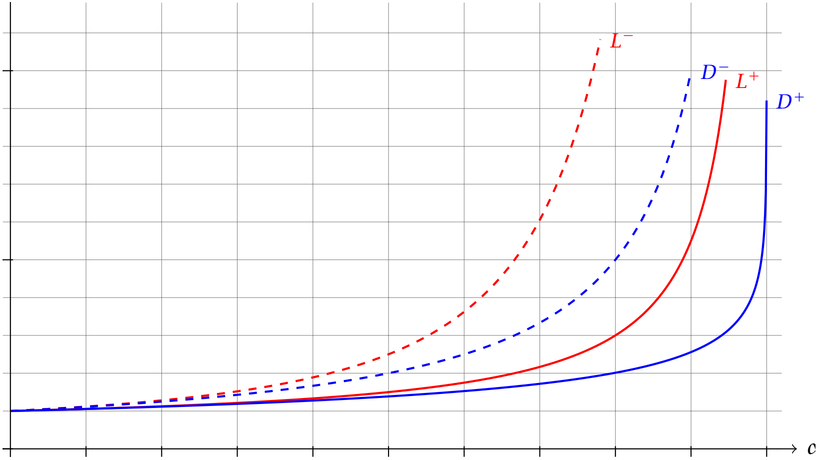
$$D^- \approx \frac{1}{1-\alpha}$$

7.7 Hashing

Some values:

α	Linear Probing		Quadratic Probing		Double Hashing	
	L^+	L^-	Q^+	Q^-	D^+	D^-
0.5	1.5	2.5	1.44	2.19	1.39	2
0.9	5.5	50.5	2.85	11.40	2.55	10
0.95	10.5	200.5	3.52	22.05	3.15	20

7.7 Hashing



Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]\\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot\\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\Pr[X \geq i]$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\Pr[X \geq i] = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}] &\\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1}\end{aligned}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i_1}]& \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i_1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} \ .\end{aligned}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X]$$

Analysis of Idealized Open Address Hashing

$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \ .$$

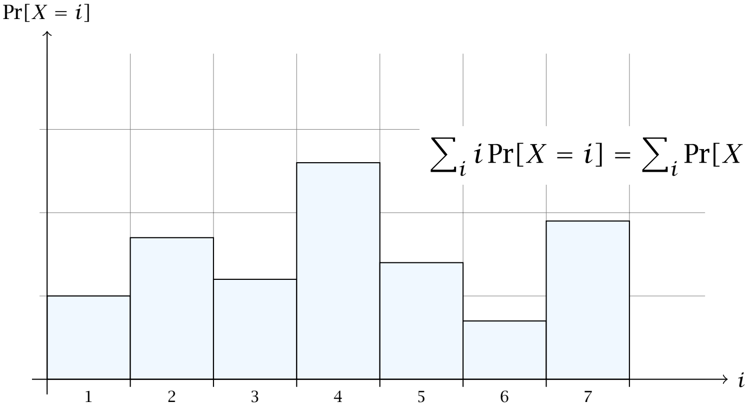
Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \enspace .$$

Analysis of Idealized Open Address Hashing

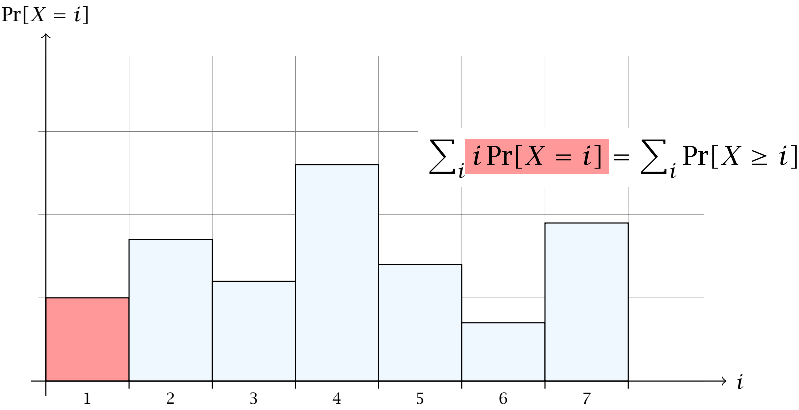
$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \ .$$

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$



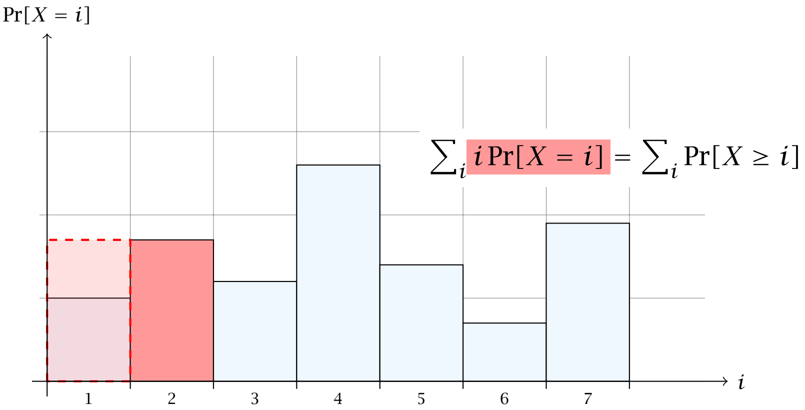
$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

$i = 1$



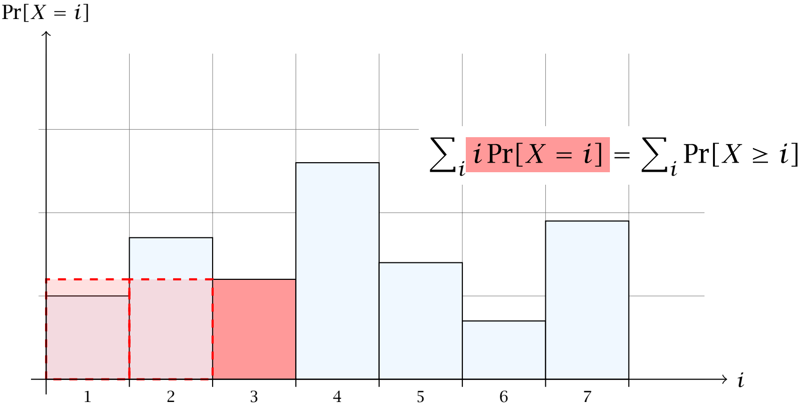
$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

$i = 2$



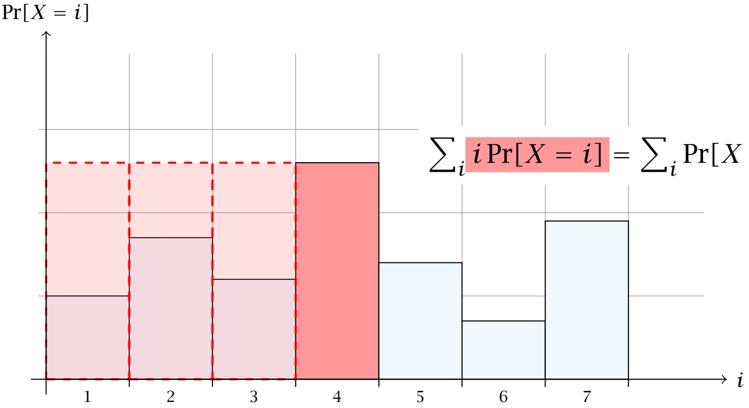
$$\sum_i \boxed{i \Pr[X = i]} = \sum_i \Pr[X \geq i]$$

$i = 3$



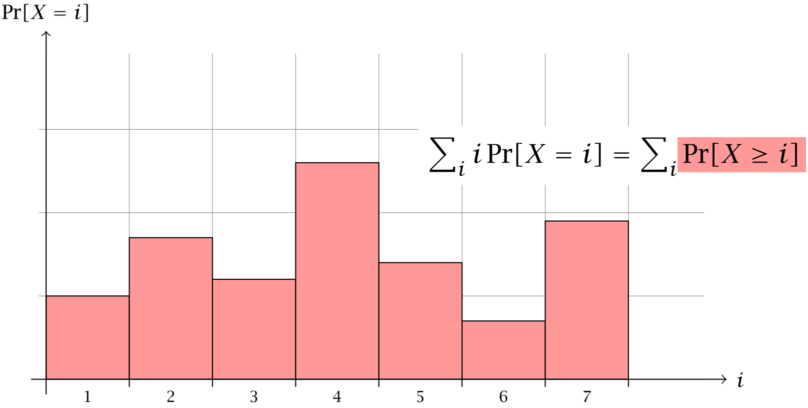
$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

$i = 4$



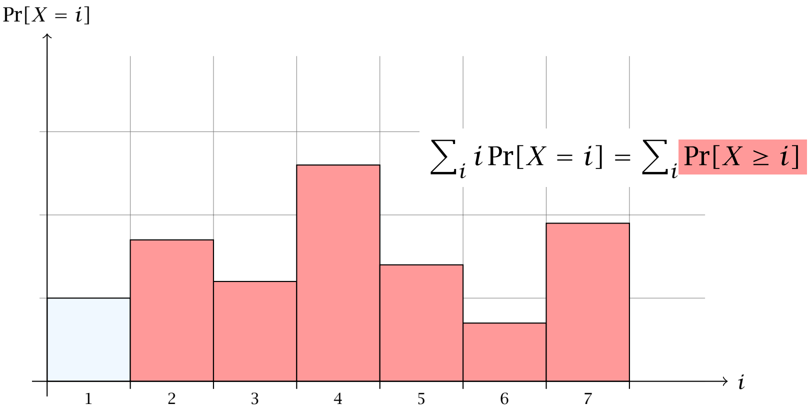
$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$

$i = 1$



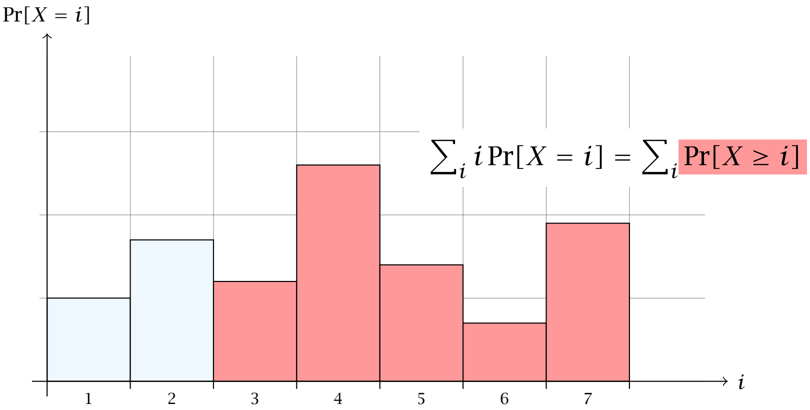
$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$

$i = 2$



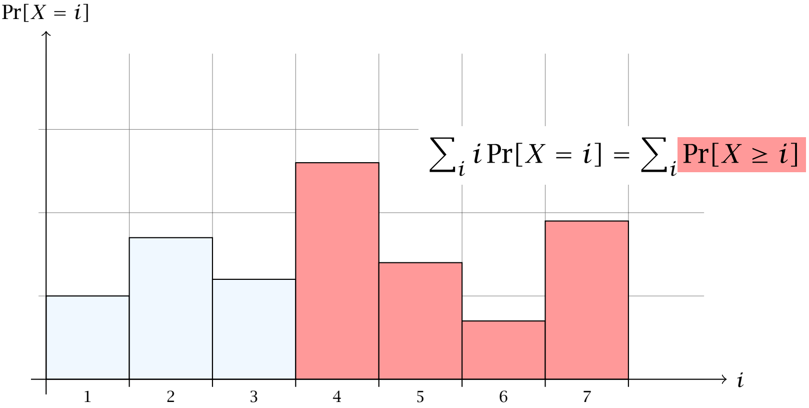
$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$

$i = 3$

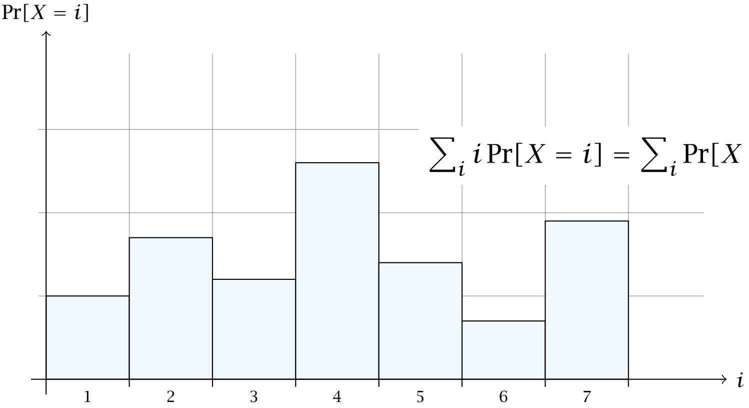


$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$

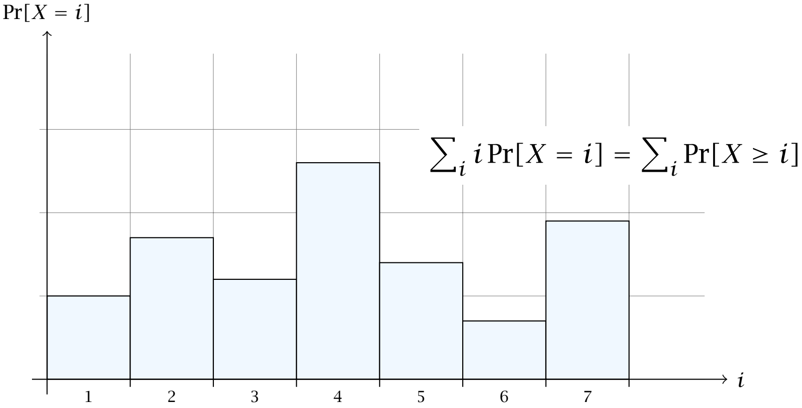
$i = 4$



$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$



$$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$$



The j-th rectangle appears in both sums j times. (j times in the first due to multiplication with j; and j times in the second for summands $i = 1, 2, \ldots, j$)

Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\le \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m}$$

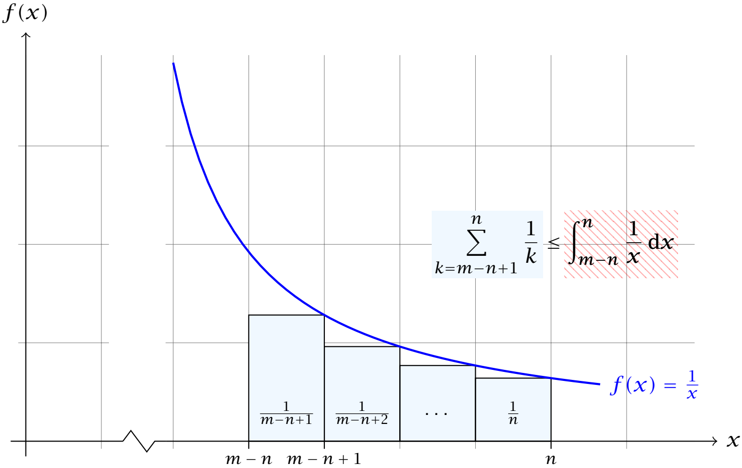
Analysis of Idealized Open Address Hashing

The number of probes in a successful for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\le \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\ .$$



$$\sum_{k=m-n+1}^{n} \frac{1}{k} \leq \int_{m-n}^{n} \frac{1}{x}\,\mathrm{d}x$$

7.7 Hashing

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

7.7 Hashing

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

7.7 Hashing

Definition 25

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called universal if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Note that this means that $\Pr[h(u_1) = h(u_2)] = \frac{1}{n}$.

7.7 Hashing

Definition 25

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ is called universal if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} \;,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Note that this means that $\Pr[h(u_1) = h(u_2)] = \frac{1}{n}$.

7.7 Hashing

Definition 26

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0, \dots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2:

$$\Pr[h(u_1) = t_1 \land h(u_2) = t_2] \leq \frac{1}{n^2} \ .$$

Note that the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

This requirement clearly implies a universal hash-function.

7.7 Hashing

Definition 26

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\ldots,n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0,\ldots,n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2:

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} \enspace .$$

Note that the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

This requirement clearly implies a universal hash-function.

7.7 Hashing

Definition 27

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called k-independent if for any choice of $\ell \le k$ distinct keys $u_1, \dots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \dots, t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \le \frac{1}{n^\ell} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

7.7 Hashing

Definition 28

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called (μ, k)-independent if for any choice of $\ell \leq k$ distinct keys $u_1, \ldots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \ldots, t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \leq \left(\frac{\mu}{n}\right)^\ell ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

7.7 Hashing

Let $U := \{0, \dots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \dots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 29

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \dots, n-1\}$.

7.7 Hashing

Let $U := \{0, \dots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \dots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 29

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \dots, n-1\}$.

7.7 Hashing

Let $U := \{0, \ldots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \ldots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \ldots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 29

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \ldots, n-1\}$.

7.7 Hashing

Let $U := \{0, \dots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \dots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 29

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U *to* $\{0, \dots, n-1\}$.

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $ax + b \not\equiv ay + b \pmod{p}$

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $ax + b \not\equiv ay + b \pmod{p}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (KÃČÃűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

$$a(x - y) \not\equiv 0 \pmod{p}$$

where we use that $\mathbb{Z}_p$ is a field (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

7.7 Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (KÃČÂűrper) and, hence, has no zero divisors (nullteilerfrei).

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

- The hash-function does not generate collisions before the $(\bmod\ n)$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

  This holds because we can compute a and b when given t_x and t_y:

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

  This holds because we can compute a and b when given t_x and t_y:

$$t_x \equiv ax + b \pmod{p}$$
$$t_y \equiv ay + b \pmod{p}$$

- The hash-function does not generate collisions before the $\pmod n$-operation. Furthermore, every choice (a,b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

  This holds because we can compute a and b when given t_x and t_y:

$$\begin{aligned} t_x &\equiv ax + b && \pmod p \\ t_y &\equiv ay + b && \pmod p \end{aligned}$$

$$\begin{aligned} t_x - t_y &\equiv a(x-y) && \pmod p \\ t_y &\equiv ay + b && \pmod p \end{aligned}$$

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to different hash-values $t_x := h_{a,b}(x)$ and $t_y := h_{a,b}(y)$.

  This holds because we can compute a and b when given t_x and t_y:

$$\begin{aligned} t_x &\equiv ax + b &\pmod{p}\\ t_y &\equiv ay + b &\pmod{p}\end{aligned}$$

$$\begin{aligned} t_x - t_y &\equiv a(x - y) &\pmod{p}\\ t_y &\equiv ay + b &\pmod{p}\end{aligned}$$

$$\begin{aligned} a &\equiv (t_x - t_y)(x - y)^{-1} &\pmod{p}\\ b &\equiv ay - t_y &\pmod{p}\end{aligned}$$

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a,b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p-1$ possible values for choosing t_y.

From the range $0, \dots, p-1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p-1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a,b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p-1$ possible values for choosing t_y.

From the range $0, \dots, p-1$ the values $t_x, t_x+n, t_x+2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a,b), $a \neq 0$) and pairs (t_x,t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x,t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p-1$ possible values for choosing t_y.

From the range $0,\dots,p-1$ the values $t_x, t_x+n, t_x+2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $(\bmod n)$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $(\bmod n)$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \le \frac{p}{n} + \frac{n-1}{n} - 1 \le \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \le \frac{p}{n} + \frac{n-1}{n} - 1 \le \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

7.7 Hashing

7.7 Hashing

It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array}\right]$$

7.7 Hashing

It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor\frac{p}{n}\right\rfloor^2}{p(p-1)} \leq \Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array}\right] \leq \frac{\left\lceil\frac{p}{n}\right\rceil^2}{p(p-1)}$$

7.7 Hashing

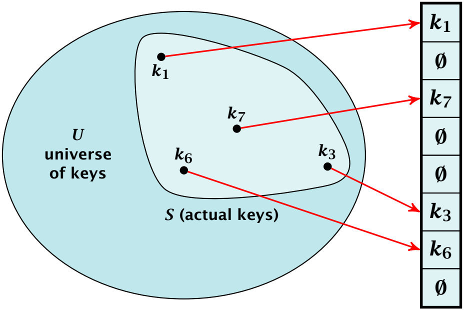
It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)} \le \Pr_{t_x \ne t_y \in \mathbb{Z}_p^2} \left[\begin{array}{c} t_x \bmod n{=}h_1 \\ \wedge \\ t_y \bmod n{=}h_2 \end{array} \right] \le \frac{\left\lceil \frac{p}{n} \right\rceil^2}{p(p-1)}$$

Note that the middle is the probability that $h(x) = h_1$ and $h(y) = h_2$. The total number of choices for (t_x, t_y) is $p(p-1)$. The number of choices for t_x (t_y) such that $t_x \bmod n = h_1$ ($t_y \bmod n = h_2$) lies between $\lfloor \frac{p}{n} \rfloor$ and $\lceil \frac{p}{n} \rceil$.

Perfect Hashing

Suppose that we know the set S of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.



Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \ .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \;\; .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\text{\#Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\mathrm{E}\Big[\sum_j m_j^2\Big]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\mathrm{E}\Big[\sum_j m_j^2\Big] = \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$
\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}
$$

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\begin{aligned}\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]\\ &= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]\end{aligned}$$

The first expectation is simply the expected number of collisions, for the first level.

Perfect Hashing

The total memory that is required by all hash-tables is $\sum_j m_j^2$.

$$\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}$$

The first expectation is simply the expected number of collisions, for the first level.

$$= 2\binom{m}{2}\frac{1}{m} + m = 2m - 1$$

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket.

We only need that the hash-function is universal!!!

Cuckoo Hashing

Goal:

Try to generate a perfect hash-table (constant worst-case search time) in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (constant worst-case search time) in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (constant worst-case search time) in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

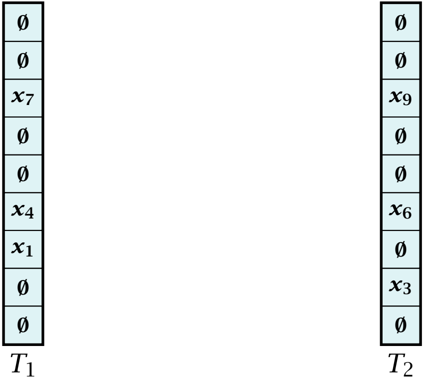
Cuckoo Hashing

Goal:
Try to generate a perfect hash-table (constant worst-case search time) in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

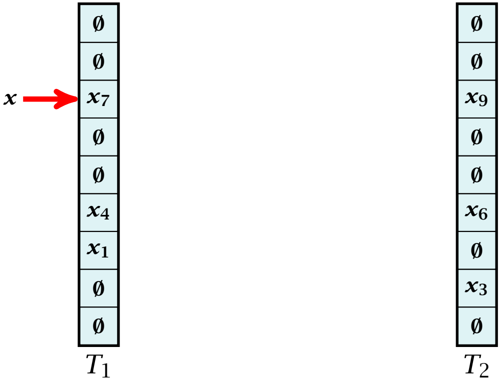
Cuckoo Hashing

Insert:



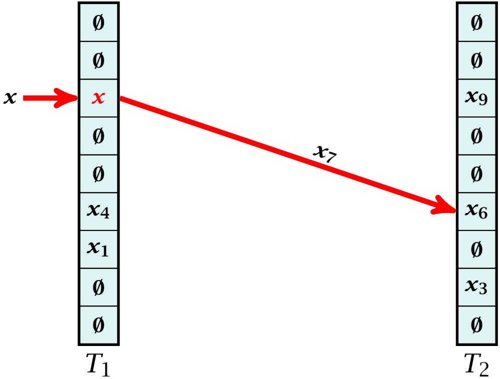
Cuckoo Hashing

Insert:



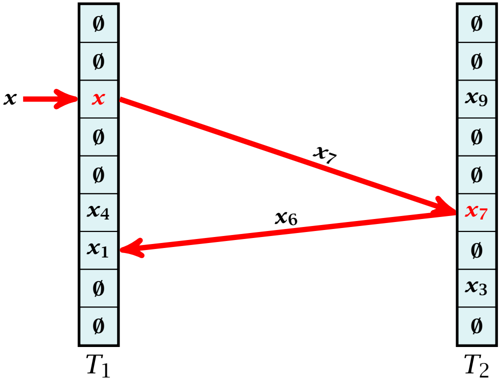
Cuckoo Hashing

Insert:



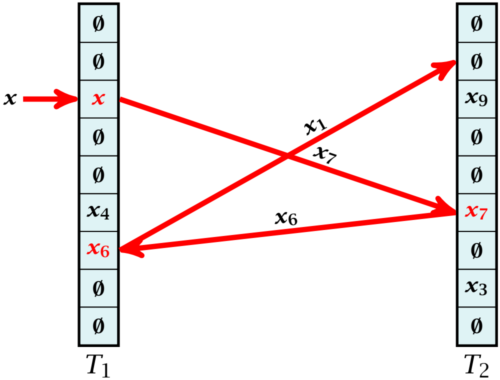
Cuckoo Hashing

Insert:



Cuckoo Hashing

Insert:



Cuckoo Hashing

Algorithm 16 Cuckoo-Insert(x)

```
if $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$ then return
steps $\leftarrow 1$
while steps $\leq$ maxsteps do
    exchange $x$ and $T_1[h_1(x)]$
    if $x$ = null then return
    exchange $x$ and $T_2[h_2(x)]$
    if $x$ = null then return
rehash() // change table-size and rehash everything
Cuckoo-Insert($x$)
```

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

Cuckoo Hashing

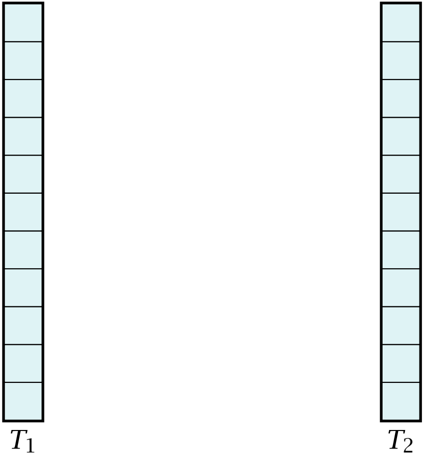
What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches ℓ different keys (apart from x)?

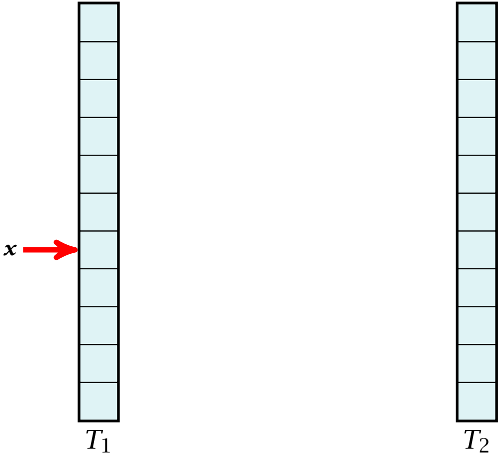
Cuckoo Hashing

Insert:



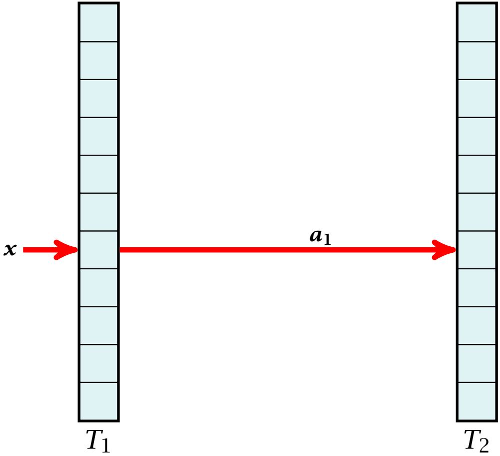
Cuckoo Hashing

Insert:



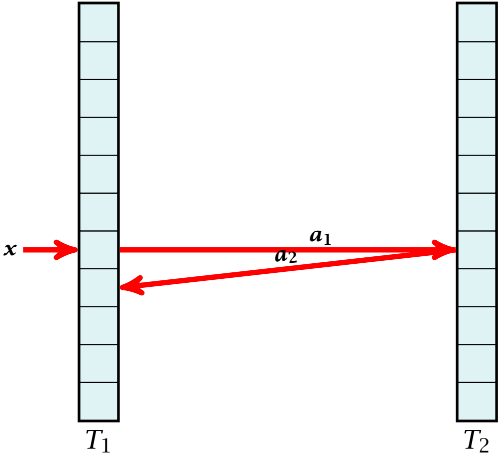
Cuckoo Hashing

Insert:



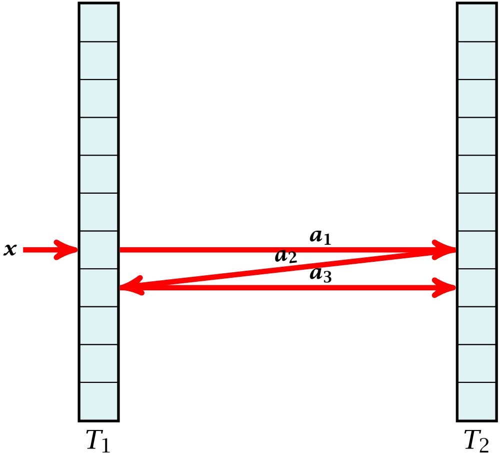
Cuckoo Hashing

Insert:



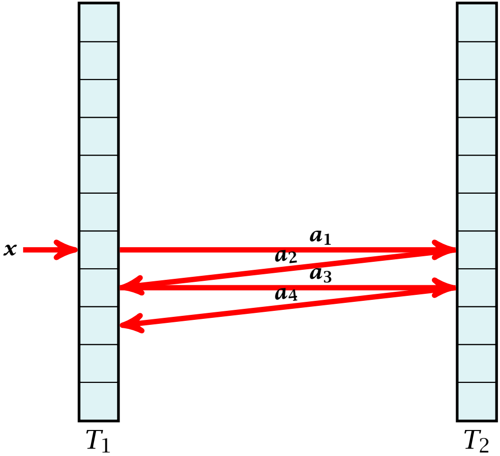
Cuckoo Hashing

Insert:



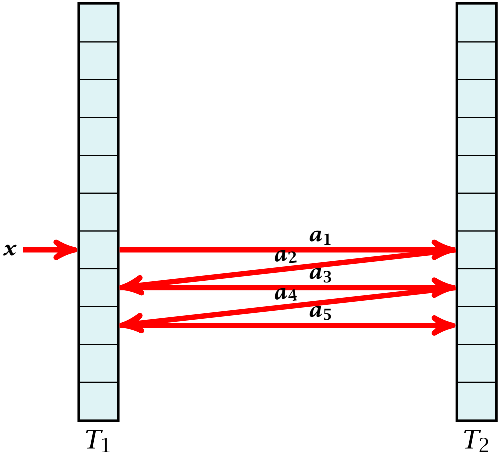
Cuckoo Hashing

Insert:



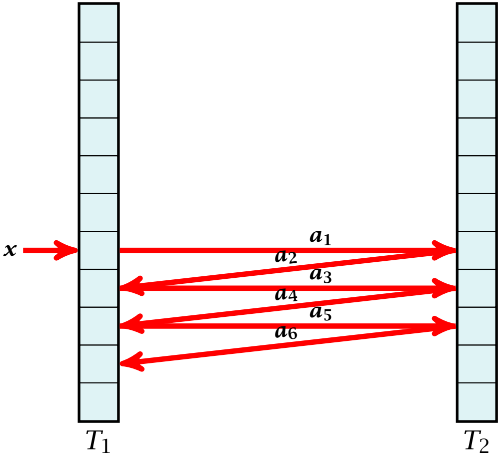
Cuckoo Hashing

Insert:



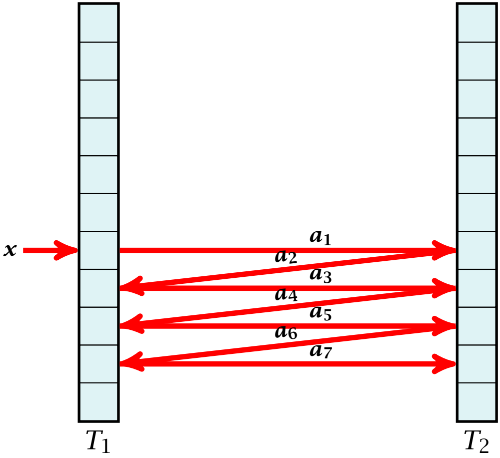
Cuckoo Hashing

Insert:



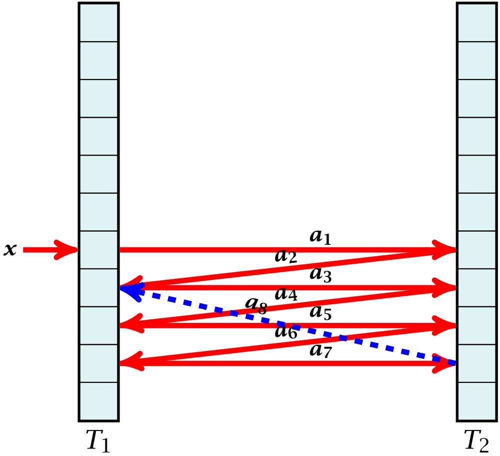
Cuckoo Hashing

Insert:



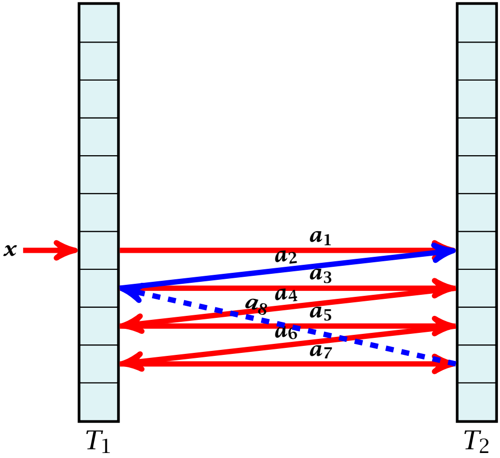
Cuckoo Hashing

Insert:



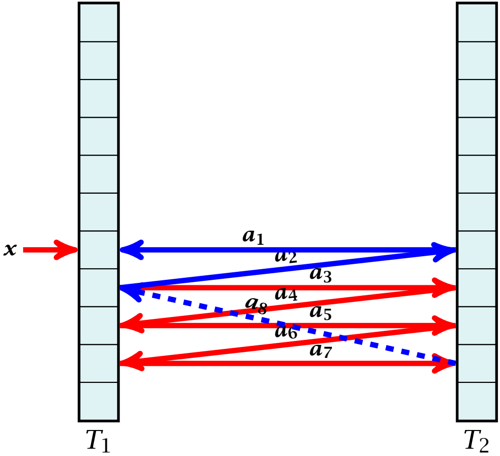
Cuckoo Hashing

Insert:



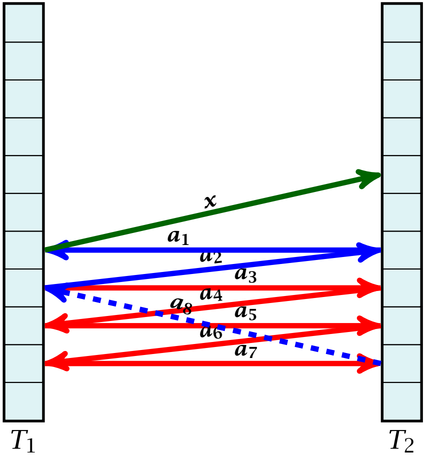
Cuckoo Hashing

Insert:



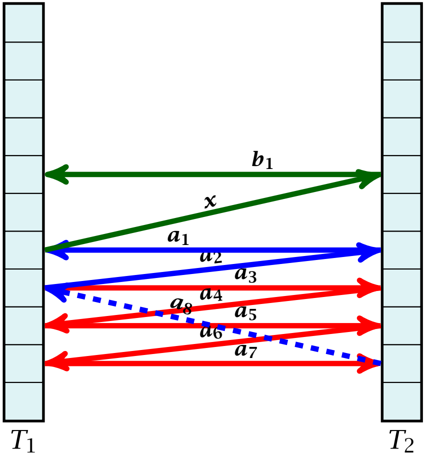
Cuckoo Hashing

Insert:



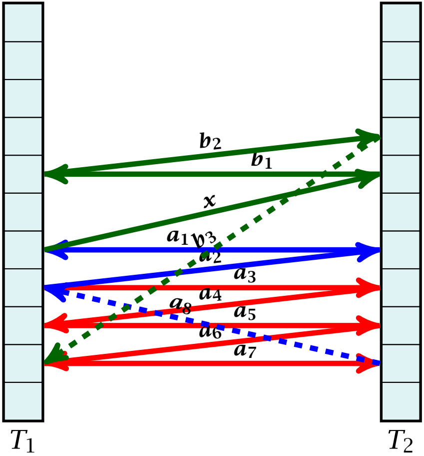
Cuckoo Hashing

Insert:



Cuckoo Hashing

Insert:



Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \ge 2$.
- An index $j_a \in \{1, \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \ge 0$.
- An index $j_b \in \{1, \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1\dots,\ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1\dots,\ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} "jumps back" in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} "jumps back" in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

A cycle-structure is defined by

- ℓ_a keys $a_1, a_2, \dots a_{\ell_a}$, $\ell_a \geq 2$,
- An index $j_a \in \{1 \dots, \ell_a - 1\}$ that defines how much the last item a_{ℓ_a} “jumps back” in the sequence.
- ℓ_b keys $b_1, b_2, \dots b_{\ell_b}$. $b \geq 0$.
- An index $j_b \in \{1 \dots, \ell_a + \ell_b\}$ that defines how much the last item b_{ℓ_b} “jumps back” in the sequence.
- An assignment of positions for the keys in both tables. Formally we have positions $p_1, \dots, p_{\ell_a}$, and $p'_1, \dots, p'_{\ell_b}$.
- The size of a cycle-structure is defined as $\ell_a + \ell_b$.

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- …
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- …

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p_1'$
- $h_1(b_1) = h_1(b_2) = p_2'$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p_1'$
- $h_1(b_1) = h_1(b_2) = p_2'$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

We say a cycle-structure is active for key x if the hash-functions are chosen in such a way that the hash-function results match the pre-defined key-positions.

- $h_1(x) = h_1(a_1) = p_1$
- $h_2(a_1) = h_2(a_2) = p_2$
- $h_1(a_2) = h_1(a_3) = p_3$
- $\dots$
- if ℓ_a is even then $h_1(a_\ell) = p_{s_a}$, otw. $h_2(a_\ell) = p_{s_a}$
- $h_2(x) = h_2(b_1) = p'_1$
- $h_1(b_1) = h_1(b_2) = p'_2$
- $\dots$

Cuckoo Hashing

Observation If we end up in an infinite loop there must exist a cycle-structure that is active for x.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)}\quad,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)} ,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

A cycle-structure is defined without knowing the hash-functions.

Whether a cycle-structure is active for key x depends on the hash-functions.

Lemma 30

A given cycle-structure of size s is active for key x with probability at most

$$\left(\frac{\mu}{n}\right)^{2(s+1)} \quad ,$$

if we use $(\mu, s+1)$-independent hash-functions.

Cuckoo Hashing

Proof.

All positions are fixed by the cycle-structure. Therefore we ask for the probability of mapping $s+1$ keys (the a-keys, the b-keys and x) to pre-specified positions in T_1, **and** to pre-specified positions in T_2.

The probability is

$$\left(\frac{\mu}{n}\right)^{s+1} \cdot \left(\frac{\mu}{n}\right)^{s+1} ,$$

since h_1 and h_2 are chosen independently. □

Cuckoo Hashing

Proof.

All positions are fixed by the cycle-structure. Therefore we ask for the probability of mapping $s+1$ keys (the a-keys, the b-keys and x) to pre-specified positions in T_1, **and** to pre-specified positions in T_2.

The probability is

$$\left(\frac{\mu}{n}\right)^{s+1} \cdot \left(\frac{\mu}{n}\right)^{s+1} ,$$

since h_1 and h_2 are chosen independently. □

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_b}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1,\dots,a_{\ell_a}$ and $b_1,\dots,b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1,\dots,p_{\ell_a}$ and $p'_1,\dots,p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1, \dots, a_{\ell_a}$ and $b_1, \dots, b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1, \dots, p_{\ell_a}$ and $p'_1, \dots, p'_{\ell_a}$.

Cuckoo Hashing

The number of cycle-structures of size s is small:

- There are at most s ways to choose ℓ_a. This fixes ℓ_b.
- There are at most s^2 ways to choose j_a, and j_b.
- There are at most m^s possibilities to choose the keys $a_1,\dots,a_{\ell_a}$ and $b_1,\dots,b_{\ell_b}$.
- There are at most n^s choices for choosing the positions $p_1,\dots,p_{\ell_a}$ and $p'_1,\dots,p'_{\ell_a}$.

Cuckoo Hashing

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

The probability that there is an active cycle-structure of size s is at most

$$s^3(mn)^s \cdot \left(\frac{\mu}{n}\right)^{2(s+1)}$$

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

The probability that there is an active cycle-structure of size s is at most

$$s^3(mn)^s \cdot \Big(\frac{\mu}{n}\Big)^{2(s+1)} = \frac{s^3}{mn}\Big(mn\Big)^{s+1}\Big(\frac{\mu^2}{n^2}\Big)^{s+1}$$

Cuckoo Hashing

Hence, there are at most $s^3(mn)^2$ cycle-structures of size s.

The probability that there is an active cycle-structure of size s is at most

$$\begin{aligned} s^3(mn)^s \cdot \Big(\frac{\mu}{n}\Big)^{2(s+1)} &= \frac{s^3}{mn}\Big(mn\Big)^{s+1}\Big(\frac{\mu^2}{n^2}\Big)^{s+1} \\ &= \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1} \end{aligned}$$

Cuckoo Hashing

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\Pr[\text{there exists an active cycle-structure}]$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$\begin{aligned}&\Pr[\text{there exists an active cycle-structure}]\\&\qquad\qquad\leq \sum_{s=2}^{\infty}\Pr[\text{there exists an act. cycle-structure of size } s]\end{aligned}$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$
\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}] \\
&\qquad \leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s] \\
&\qquad \leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1}
\end{aligned}
$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$
\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}] \\
&\qquad \leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s] \\
&\qquad \leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1} \\
&\qquad \leq \frac{1}{mn}\sum_{s=0}^{\infty} s^3\Big(\frac{1}{1+\delta}\Big)^{s}
\end{aligned}
$$

Cuckoo Hashing

If we make sure that $n \geq (1+\delta)\mu^2 m$ for a constant δ (i.e., the hash-table is not too full) we obtain

$$
\begin{aligned}
&\Pr[\text{there exists an active cycle-structure}]\\
&\qquad \leq \sum_{s=2}^{\infty} \Pr[\text{there exists an act. cycle-structure of size } s]\\
&\qquad \leq \sum_{s=2}^{\infty} \frac{s^3}{mn}\Big(\frac{\mu^2 m}{n}\Big)^{s+1}\\
&\qquad \leq \frac{1}{mn}\sum_{s=0}^{\infty} s^3\Big(\frac{1}{1+\delta}\Big)^{s}\\
&\qquad \leq \frac{1}{m^2}\cdot \mathcal{O}(1)\enspace .
\end{aligned}
$$

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \ldots, a_{\ell_a}$ and $x, b_1, b_2, \ldots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \leq 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \ldots, a_{\ell_a}$ and $x, b_1, b_2, \ldots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \ldots, a_{\ell_a}$ and $x, b_1, b_2, \ldots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \leq 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \ldots, a_{\ell_a}$ and $x, b_1, b_2, \ldots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Now assume that the insert operation takes t steps and does not create an infinite loop.

Consider the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ where the a_i's and b_i's are defined as before (but for the construction we only use keys examined during the while loop)

If the insert operation takes t steps then

$$t \le 2\ell_a + 2\ell_b + 2$$

as no key is examined more than twice.

Hence, one of the sequences $x, a_1, a_2, \dots, a_{\ell_a}$ and $x, b_1, b_2, \dots, b_{\ell_b}$ must contain at least $t/4$ keys (either $\ell_a + 1$ or $\ell_b + 1$ must be larger than $t/4$).

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \ldots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \ldots, p_\ell$ from $\{0, \ldots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \ldots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \ldots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \dots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \dots, p_\ell$ from $\{0, \dots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \dots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \dots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Define a sub-sequence of length ℓ starting with x, as a sequence $x_1, \ldots, x_\ell$ of keys with $x_1 = x$, together with $\ell + 1$ positions $p_0, p_1, \ldots, p_\ell$ from $\{0, \ldots, n-1\}$.

We say a sub-sequence is right-active for h_1 and h_2 if
$h_1(x) = h_1(x_1) = p_0$, $h_2(x_1) = h_2(x_2) = p_1$,
$h_1(x_2) = h_1(x_3) = p_2$, $h_2(x_3) = h_2(x_4) = p_3, \ldots$.

We say a sub-sequence is left-active for h_1 and h_2 if $h_2(x_1) = p_0$,
$h_1(x_1) = h_1(x_2) = p_1$, $h_2(x_2) = h_2(x_3) = p_2$,
$h_1(x_3) = h_1(x_4) = p_3, \ldots$.

For an active sequence starting with x the key x is supposed to have a collision with the second element in the sequence. This collision could either be in the table T_1 (left) or in the table T_2 (right). Therefore the above definitions differentiate between left-active and right-active.

Cuckoo Hashing

Observation:

If the insert takes $t \geq 4\ell$ steps there must either be a left-active or a right-active sub-sequence of length ℓ starting with x.

Cuckoo Hashing

The probability that a given sub-sequence is left-active (right-active) is at most

$$\left(\frac{\mu}{n}\right)^{2\ell} ,$$

if we use (μ,ℓ)-independent hash-functions. This holds since there are ℓ keys whose hash-values (two values per key) have to map to pre-specified positions.

Cuckoo Hashing

The number of sequences is at most $m^{\ell-1}p^{\ell+1}$ as we can choose $\ell - 1$ keys (apart from x) and we can choose $\ell + 1$ positions $p_0, \dots, p_\ell$.

The probability that there exists a left-active **or** right-active sequence of length ℓ is at most

$$\begin{aligned}\Pr[\text{there exists active sequ. of length } \ell]&\\ &\le 2 \cdot m^{\ell-1} \cdot n^{\ell+1} \cdot \Bigl(\frac{\mu}{n}\Bigr)^{2\ell}\\ &\le 2\Bigl(\frac{1}{1+\delta}\Bigr)^{\ell}\end{aligned}$$

Cuckoo Hashing

If the search does not run into an infinite loop the probability that it takes more than 4ℓ steps is at most

$$2\Big(\frac{1}{1+\delta}\Big)^{\ell}$$

We choose $\mathrm{maxsteps} = 4(1+2\log m)/\log(1+\delta)$. Then the probability of terminating the while-loop because of reaching $\mathrm{maxsteps}$ is only $\mathcal{O}(\frac{1}{m^2})$ ($\mathcal{O}(1/m^2)$ because of reaching an infinite loop and $1/m^2$ because the search takes $\mathrm{maxsteps}$ steps without running into a loop).

Cuckoo Hashing

The expected time for an insert under the condition that $\mathrm{maxsteps}$ is not reached is

$$\sum_{\ell\geq 0}\Pr[\text{search takes at least } \ell \text{ steps} \mid \text{iteration successful}]$$

$$\leq \sum_{\ell\geq 0} 8\Big(\frac{1}{1+\delta}\Big)^{\ell} = \mathcal{O}(1)\enspace .$$

More generally, the above expression gives a bound on the cost in the successful iteration of an insert-operation (there is exactly one successful iteration).

An iteration that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash.

Cuckoo Hashing

The expected number of unsuccessful operations is $\mathcal{O}(\frac{1}{m^2})$.
Hence, the expected cost in unsuccessful iterations is only $\mathcal{O}(\frac{1}{m})$.

Hence, the total expected cost for an insert-operation is constant.

Cuckoo Hashing

What kind of hash-functions do we need?

Since $\mathrm{maxsteps}$ is $\Theta(\log m)$ it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq \mu^2(1+\delta)m$?

- Let $\alpha := 1/(\mu^2(1+\delta))$.
- Keep track of the number of elements in the table. Whenever $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\frac{\alpha}{4}n$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \frac{\alpha}{2}n$. In order for a table-expand to occur at least $\frac{\alpha}{2}n$ insertions are required. Similar, for a table-shrink at least $\frac{\alpha}{4}$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Definition 31

Let $d \in \mathbb{N}$; $q \ge n$ be a prime; and let $\vec{a} \in \{0, \dots, q-1\}^{d+1}$. Define for $x \in \{0, \dots, q\}$

$$h_{\vec{a}}(x) := \Big(\sum_{i=0}^{d} a_i x^i \bmod q \Big) \bmod n \enspace .$$

Let $\mathcal{H}_n^d := \{h_{\vec{a}} \mid \vec{a} \in \{0, \dots, q\}^{d+1}\}$. The class $\mathcal{H}_n^d$ is $(2, d+1)$-independent.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i\Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i \Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i \Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

Fix $\ell \leq d+1$; let $x_1, \ldots, x_\ell \in \{0, \ldots, q-1\}$ be keys, and let $t_1, \ldots, t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i$ for all $i \in \{1, \ldots, \ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Fix $\ell \leq d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1,\dots,\ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Fix $\ell \leq d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1,\dots,\ell\}\}$

Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \leq \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Fix $\ell \le d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1,\dots,\ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \le \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Fix $\ell \le d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1,\dots,\ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}$$

Therefore I have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \le \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \leq \left(\frac{2}{n}\right)^\ell$$

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build($x_1, \ldots, x_n$):** Creates a data-structure that contains just the elements $x_1, \ldots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum()**: Returns an element $x \in S$ with minimum key-value key$[x]$.
- **S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build($x_1, \dots, x_n$):** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum()**: Returns an element $x \in S$ with minimum key-value key$[x]$.
- **S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**.build**$(x_1, \dots, x_n)$**:** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- S**.insert**(x)**:** Adds element x to the data-structure.
- **Element** S**.minimum()**: Returns an element $x \in S$ with minimum key-value key$[x]$.
- S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean** S**.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- S**.merge**(S'): $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build$(x_1, \ldots, x_n)$:** Creates a data-structure that contains just the elements $x_1, \ldots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum$()$**: Returns an element $x \in S$ with minimum key-value $\mathrm{key}[x]$.
- **S.delete-min$()$**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty$()$**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build$(x_1, \dots, x_n)$:** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum()**: Returns an element $x \in S$ with minimum key-value $\mathrm{key}[x]$.
- **S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build$(x_1, \ldots, x_n)$:** Creates a data-structure that contains just the elements $x_1, \ldots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum()**: Returns an element $x \in S$ with minimum key-value $\text{key}[x]$.
- **S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build($x_1, \dots, x_n$):** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **Element S.minimum()**: Returns an element $x \in S$ with minimum key-value $\text{key}[x]$.
- **S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **Boolean S.empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

An addressable Priority Queue also supports:

- **Handle S.insert(x):** Adds element x to the data-structure, and returns a handle to the object for future reference.
- **S.delete(h):** Deletes element specified through handle h.
- **S.decrease-key(h, k):** Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

8 Priority Queues

An addressable Priority Queue also supports:

- **Handle S.insert(x):** Adds element x to the data-structure, and returns a handle to the object for future reference.
- **S.delete(h):** Deletes element specified through handle h.
- **S.decrease-key(h, k):** Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

8 Priority Queues

An addressable Priority Queue also supports:

- **Handle S.insert(x):** Adds element x to the data-structure, and returns a handle to the object for future reference.
- **S.delete(h):** Deletes element specified through handle h.
- **S.decrease-key(h, k):** Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

8 Priority Queues

An addressable Priority Queue also supports:

- **Handle $\boldsymbol{S}$.insert($\boldsymbol{x}$):** Adds element x to the data-structure, and returns a handle to the object for future reference.
- **$\boldsymbol{S}$.delete($\boldsymbol{h}$):** Deletes element specified through handle h.
- **$\boldsymbol{S}$.decrease-key($\boldsymbol{h}, \boldsymbol{k}$)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Dijkstra’s Shortest Path Algorithm

Algorithm 17 Shortest-Path($G = (V, E, d), s \in V$)

```
Input: weighted graph $G = (V, E, d)$; start vertex $s$;
Output: key-field of every node contains distance from $s$;
$S$.build(); // build empty priority queue
for all $v \in V \setminus \{s\}$ do
    $v.\mathrm{key} \leftarrow \infty$;
    $h_v \leftarrow S$.insert($v$);
$s.\mathrm{key} \leftarrow 0$; $S$.insert($s$);
while $S$.empty() = false do
    $v \leftarrow S$.delete-min();
    for all $x \in V$ s.t. $(v, x) \in E$ do
        if $x.\mathrm{key} > v.\mathrm{key} + d(v, x)$ then
            $S$.decrease-key($h_x$, $v.\mathrm{key} + d(v, x)$);
            $x.\mathrm{key} \leftarrow v.\mathrm{key} + d(v, x)$;
```

Prim's Minimum Spanning Tree Algorithm

Algorithm 18 Prim-MST($G = (V, E, d), s \in V$)

```
Input: weighted graph $G = (V, E, d)$; start vertex $s$;
Output: pred-fields encode MST;
$S$.build(); // build empty priority queue
for all $v \in V \setminus \{s\}$ do
    $v.\mathrm{key} \leftarrow \infty$;
    $h_v \leftarrow S$.insert($v$);
$s.\mathrm{key} \leftarrow 0$; $S$.insert($s$);
while $S$.empty() $=$ false do
    $v \leftarrow S$.delete-min();
    for all $x \in V$ s.t. $\{v, x\} \in E$ do
        if $x.\mathrm{key} > d(v, x)$ then
            $S$.decrease-key($h_x$,$d(v, x)$);
            $x.\mathrm{key} \leftarrow d(v, x)$;
            $x.\mathrm{pred} \leftarrow v$;
```

Analysis of Dijkstra and Prim

Both algorithms require:

- 1 build() operation
- $|V|$ insert() operations
- $|V|$ delete-min() operations
- $|V|$ is-empty() operations
- $|E|$ decrease-key() operations

How good a running time can we obtain?

Analysis of Dijkstra and Prim

Both algorithms require:

- 1 build() operation
- $|V|$ insert() operations
- $|V|$ delete-min() operations
- $|V|$ is-empty() operations
- $|E|$ decrease-key() operations

How good a running time can we obtain?

8 Priority Queues

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\log n$	1

Note that most applications use **build()** only to create an empty heap which then costs time 1.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

8 Priority Queues

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n\log n$	$n\log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n\log n$	$\log n$	1

Note that most applications use **build()** only to create an empty heap which then costs time 1.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

8 Priority Queues

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\log n$	1

Note that most applications use **build()** only to create an empty heap which then costs time 1.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

8 Priority Queues

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\log n$	1

Note that most applications use **build()** only to create an empty heap which then costs time 1.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

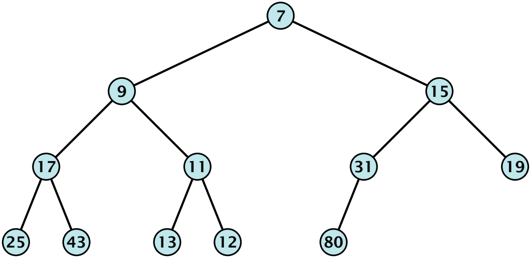
Fibonacci heaps only give an amortized guarantee.

8 Priority Queues

Using Binary Heaps, Prim and Dijkstra run in time $\mathcal{O}((|V| + |E|)\log|V|)$.

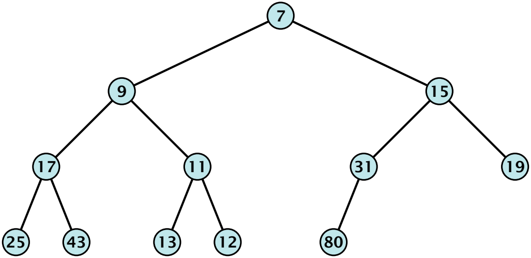
Using Fibonacci Heaps, Prim and Dijkstra run in time $\mathcal{O}(|V|\log|V| + |E|)$.

8.1 Binary Heaps



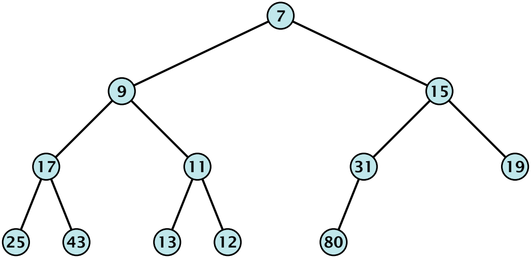
8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.



8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.
- Heap property: A node's key is not larger than the key of one of its children.



Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.

Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.

Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.

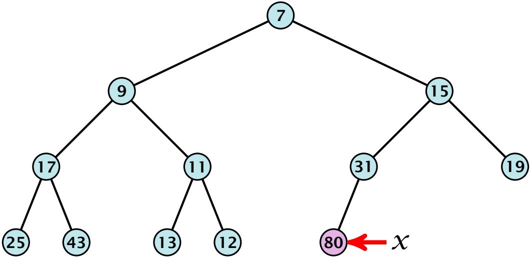
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

go up until the last edge used was a right edge.
go left; go right until you reach a leaf

if you hit the root on the way up, go to the rightmost element



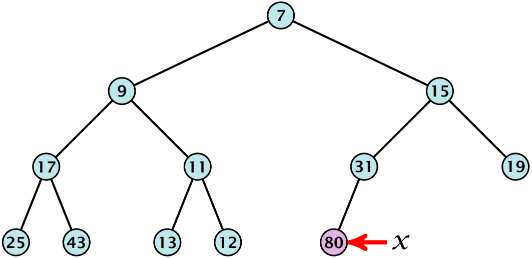
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

go up until the last edge used was a right edge.
go left; go right until you reach a leaf

if you hit the root on the way up, go to the rightmost element



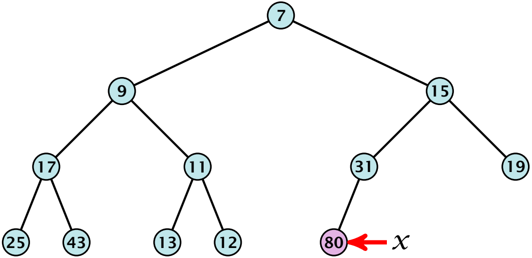
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



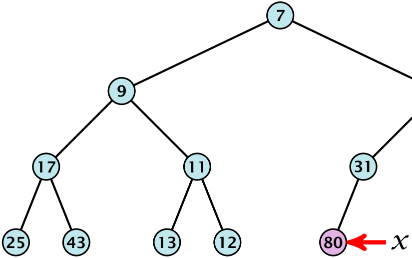
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



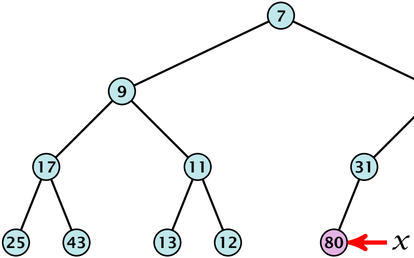
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



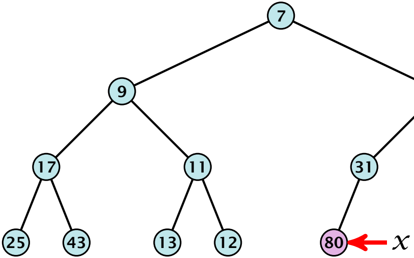
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x
  (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element;
  insert a new element as a left child;



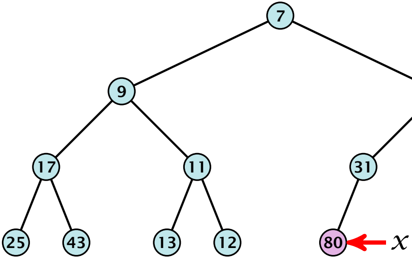
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element;
  insert a new element as a left child;



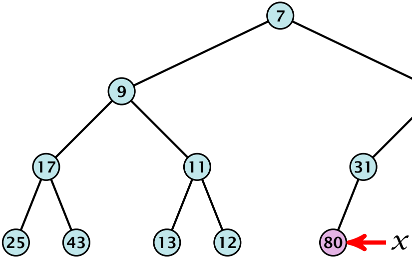
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

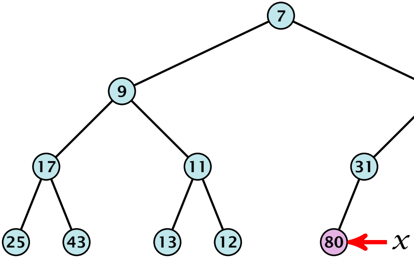
  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element;
  insert a new element as a left child;



Insert

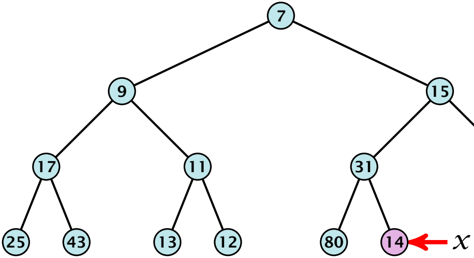
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

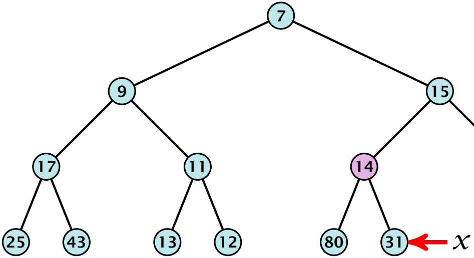
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

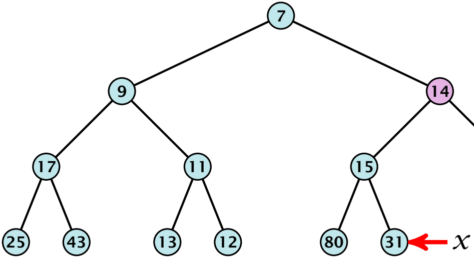
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

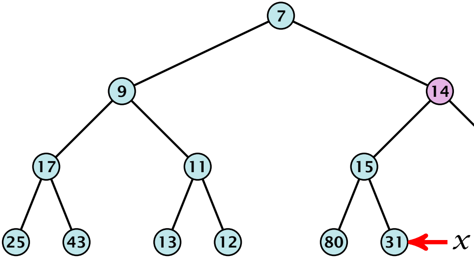
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Insert

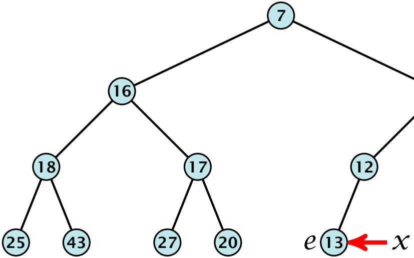
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Delete

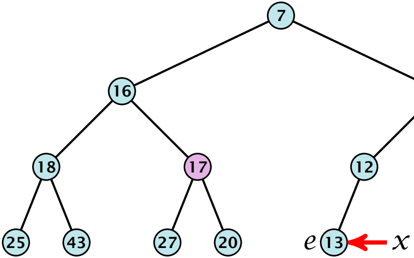
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

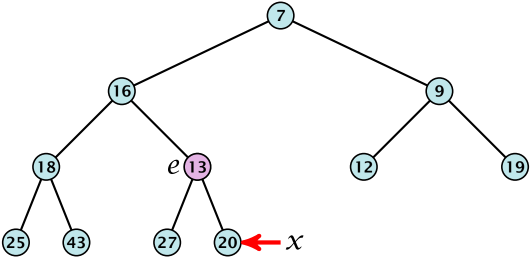
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

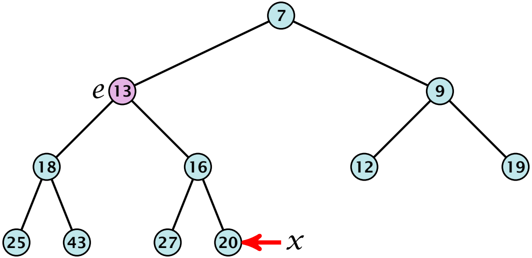
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

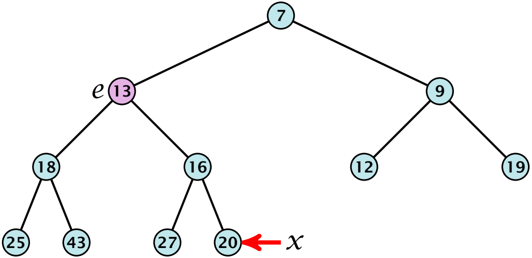
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

Delete

1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

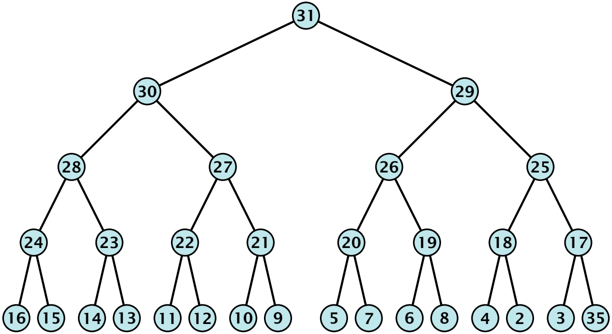
Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.
- **insert(k):** insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- **delete(h):** swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.

Build Heap

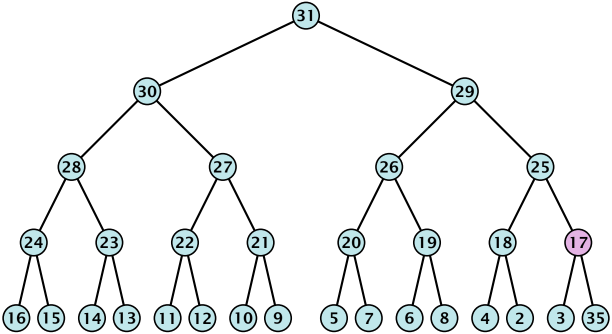
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

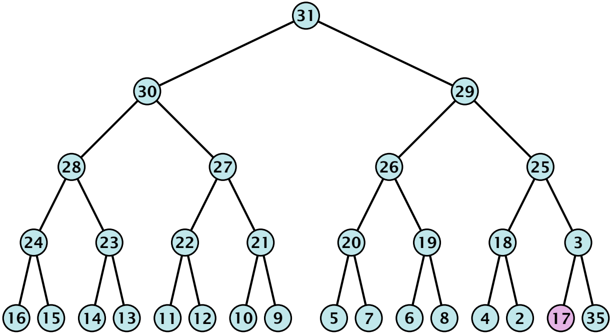
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

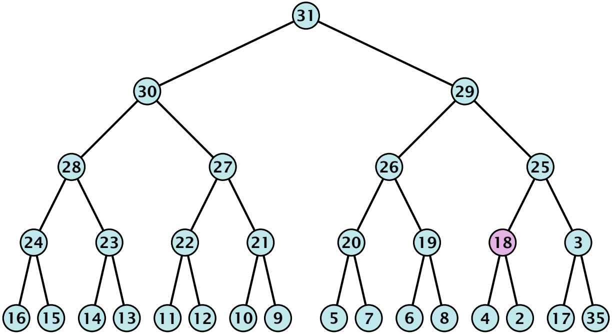
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

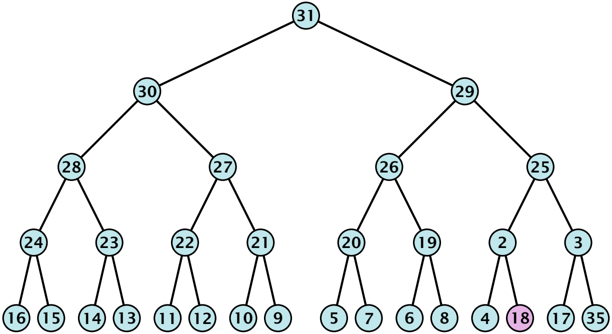
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

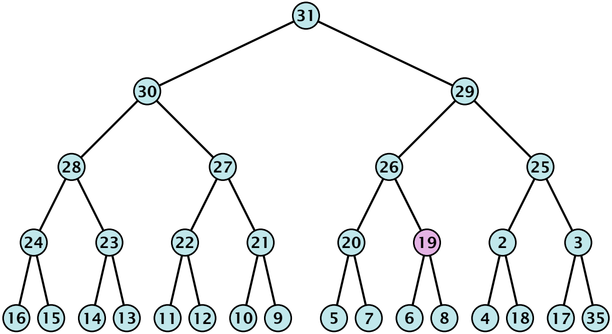
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

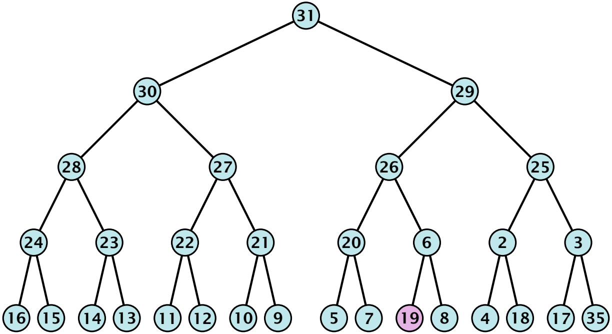
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

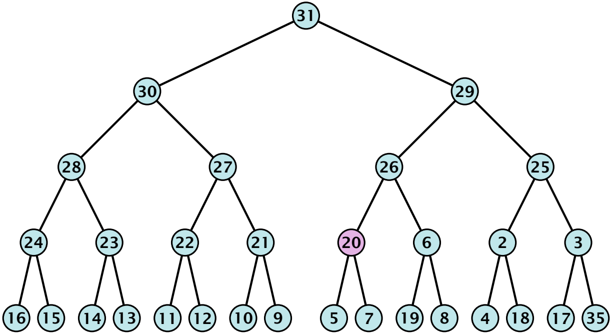
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

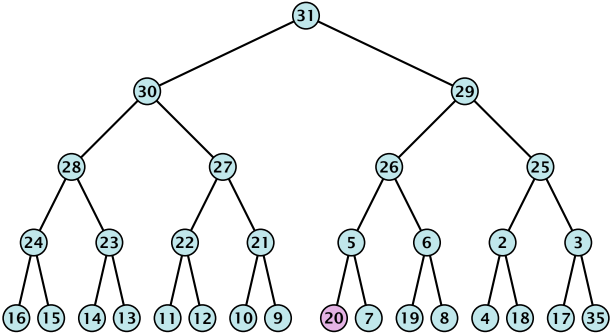
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

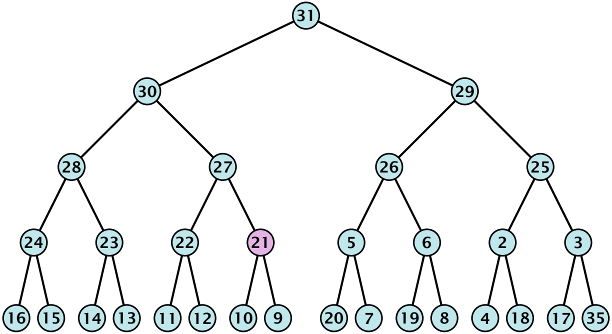
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

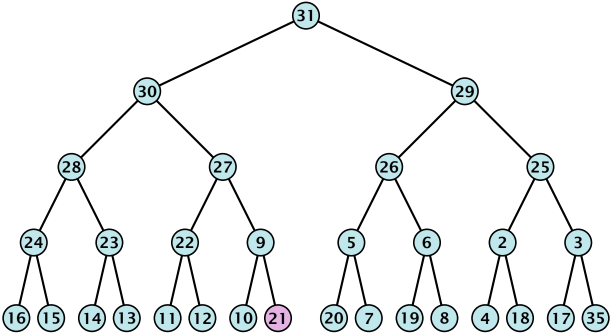
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

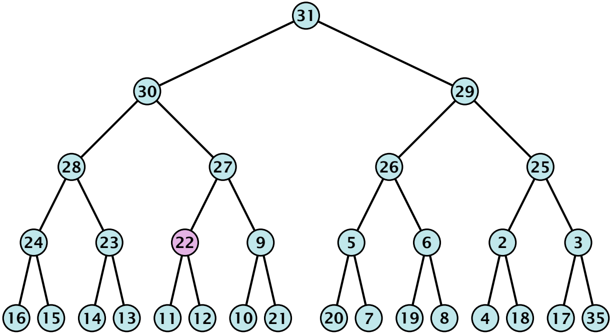
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

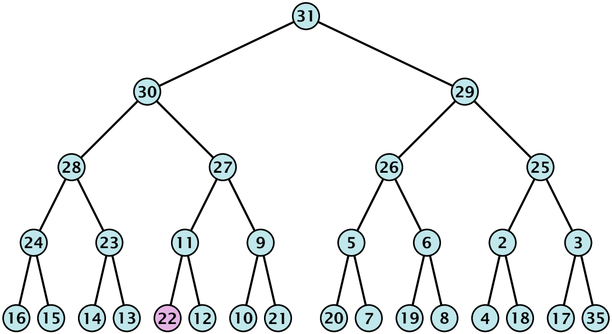
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

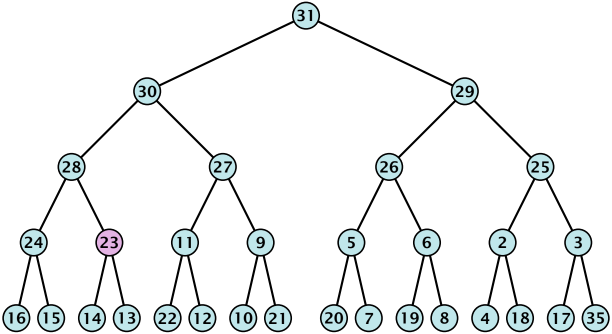
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^{h}) = \mathcal{O}(n)$$

Build Heap

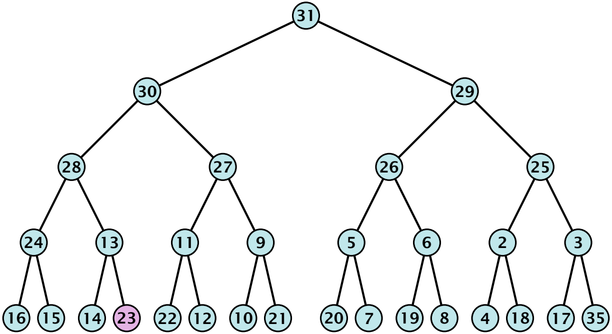
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

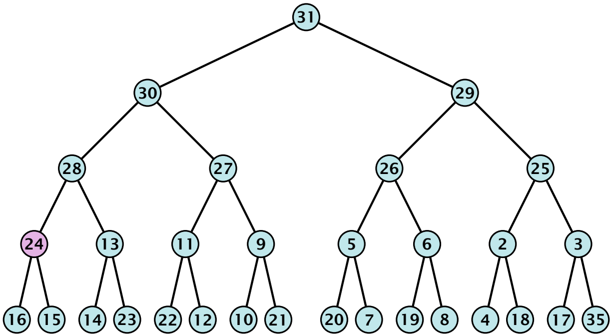
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

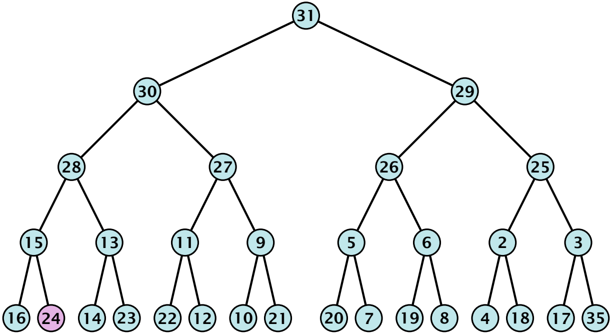
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

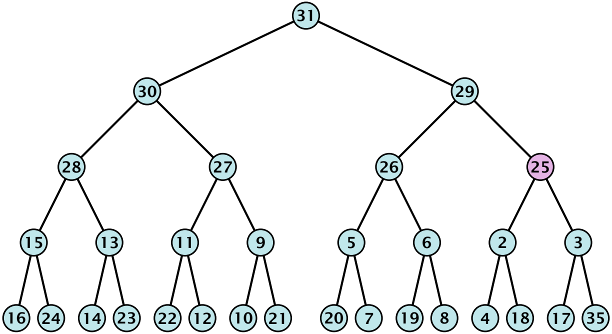
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

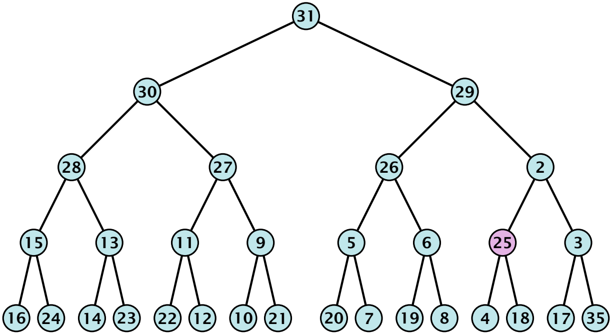
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

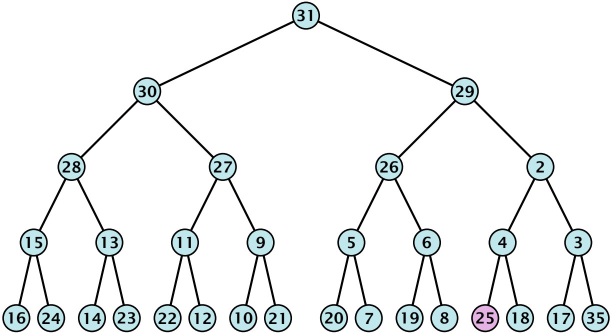
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

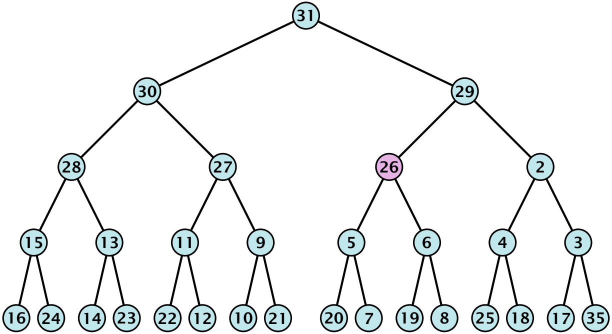
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

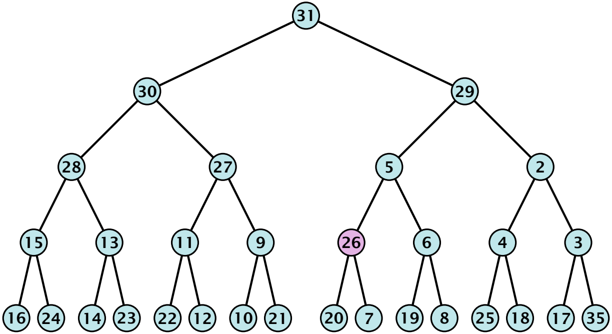
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

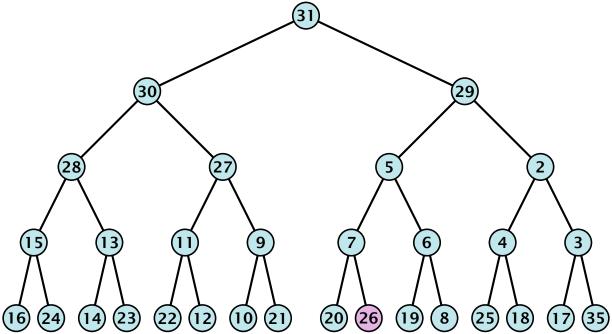
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

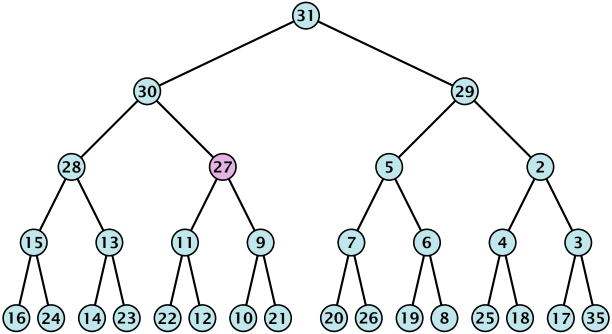
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

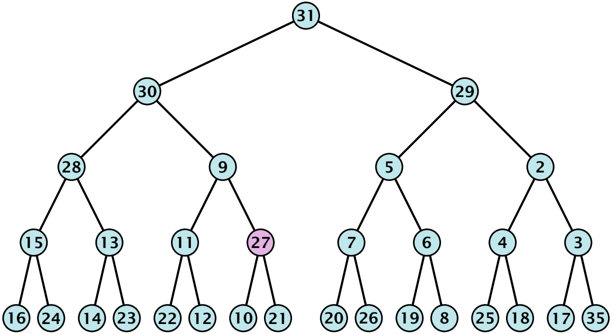
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

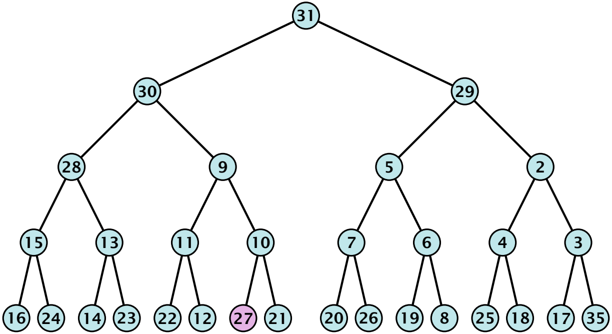
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

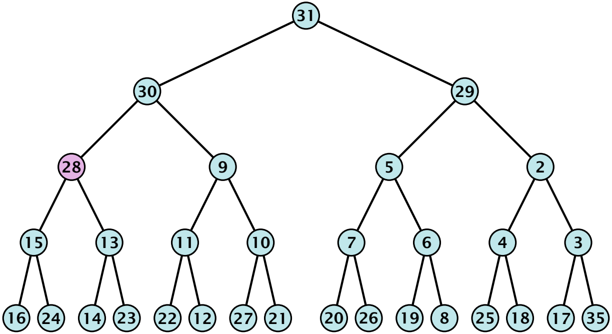
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

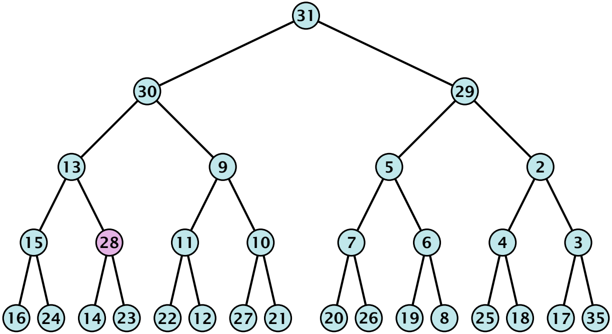
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

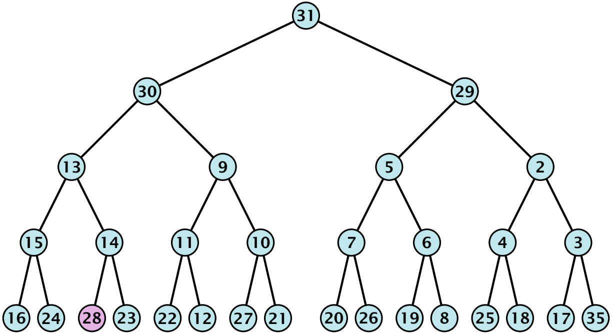
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^\ell \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

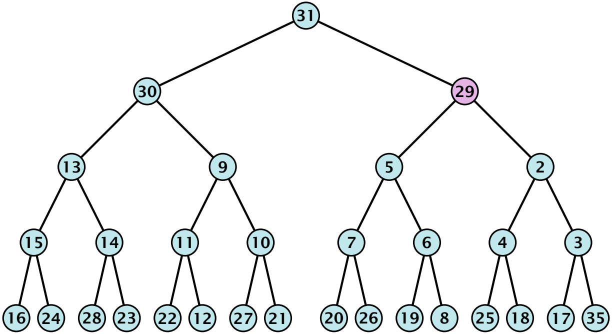
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

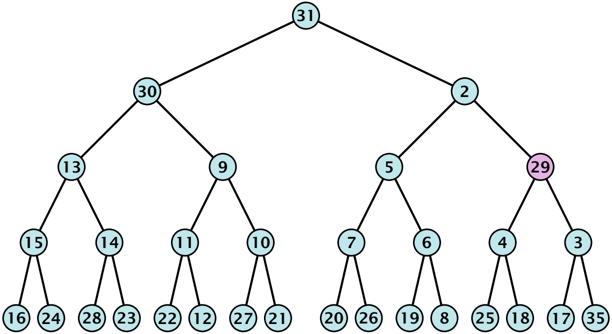
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

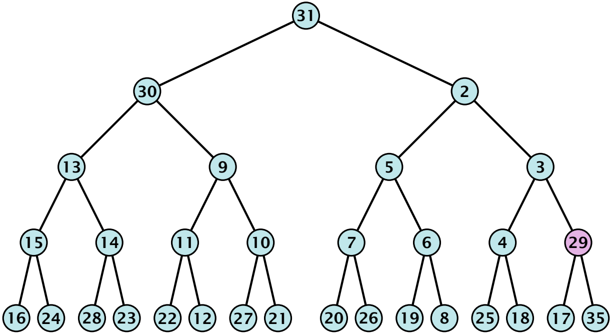
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^{h}) = \mathcal{O}(n)$$

Build Heap

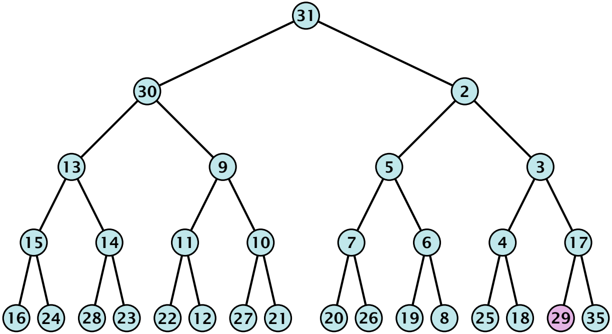
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

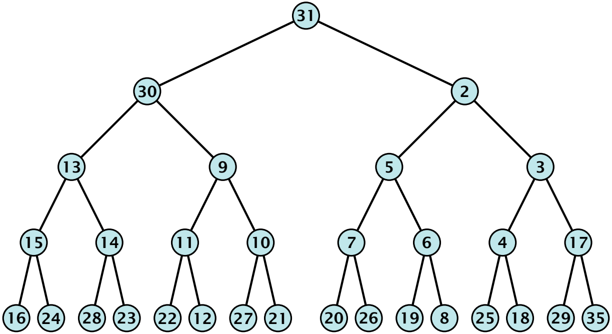
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

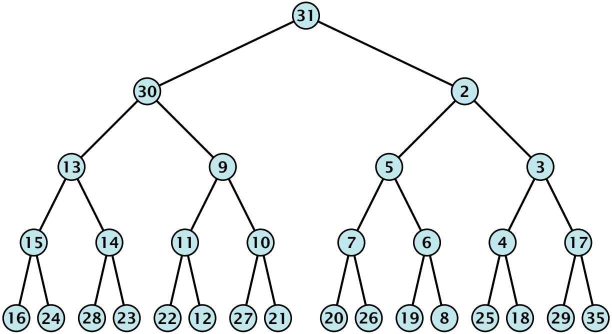
We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h-\ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Build Heap

We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Binary Heaps

Operations:

- **minimum():** Return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** Check whether root-pointer is null. Time $\mathcal{O}(1)$.
- **insert(k):** Insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- **delete(h):** Swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.
- **build($x_1, \ldots, x_n$):** Insert elements arbitrarily; then do sift-down operations starting with the lowest layer in the tree. Time $\mathcal{O}(n)$.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\ldots,n-1]$ be an array

- The parent of i-th element is at position $\lfloor\frac{i-1}{2}\rfloor$.
- The left child of i-th element is at position $2i+1$.
- The right child of i-th element is at position $2i+2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain there positions and therefore there are not stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\ldots,n-1]$ be an array

- The parent of i-th element is at position $\lfloor\frac{i-1}{2}\rfloor$.
- The left child of i-th element is at position $2i+1$.
- The right child of i-th element is at position $2i+2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain there positions and therefore there are not stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\ldots,n-1]$ be an array

- The parent of i-th element is at position $\lfloor\frac{i-1}{2}\rfloor$.
- The left child of i-th element is at position $2i+1$.
- The right child of i-th element is at position $2i+2$.

Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

The resulting binary heap is not addressable. The elements don't maintain there positions and therefore there are not stable handles.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\dots,n-1]$ be an array

- The parent of i-th element is at position $\lfloor\frac{i-1}{2}\rfloor$.
- The left child of i-th element is at position $2i+1$.
- The right child of i-th element is at position $2i+2$.

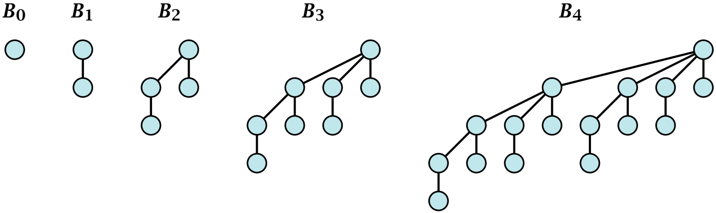
Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

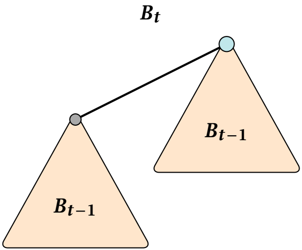
The resulting binary heap is not addressable. The elements don't maintain there positions and therefore there are not stable handles.

8.2 Binomial Heaps

Operation	Binary Heap	BST	Binomial Heap	Fibonacci Heap*
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\boldsymbol{\log n}$	1

Binomial Trees





Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

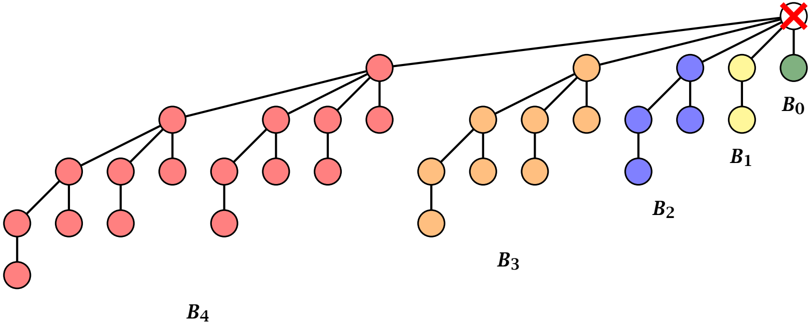
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

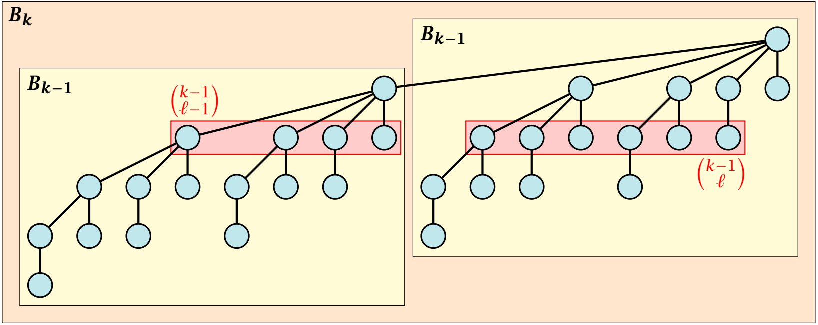
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees



Deleting the root of B_5 leaves sub-trees B_4, B_3, B_2, and B_1.

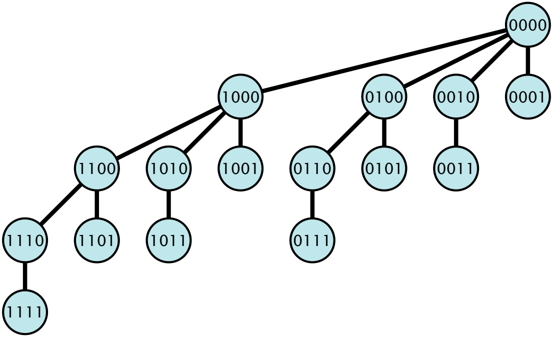
Binomial Trees



The number of nodes on level ℓ in tree B_k is therefore

$$\binom{k-1}{\ell-1} + \binom{k-1}{\ell} = \binom{k}{\ell}$$

Binomial Trees

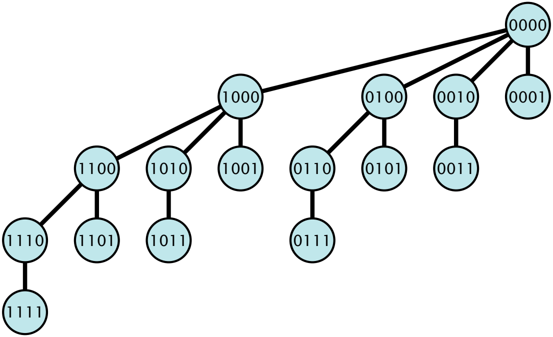


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees

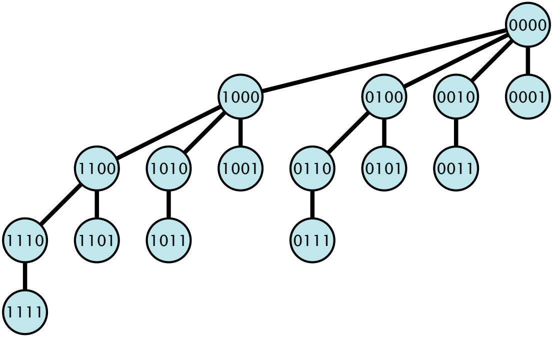


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees

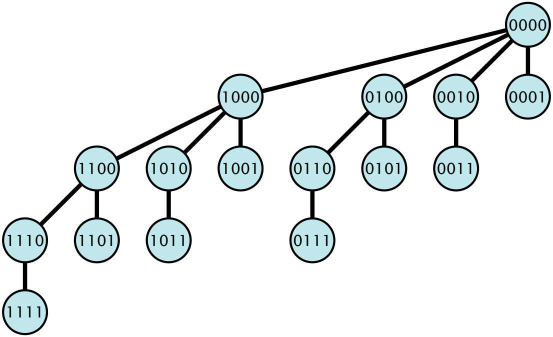


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

Binomial Trees



The binomial tree B_k is a sub-graph of the hypercube H_k.

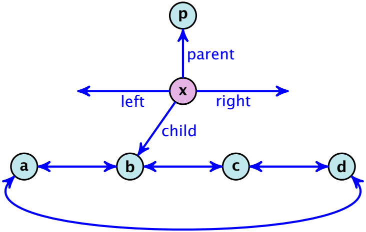
The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

8.2 Binomial Heaps

How do we implement trees with non-constant degree?

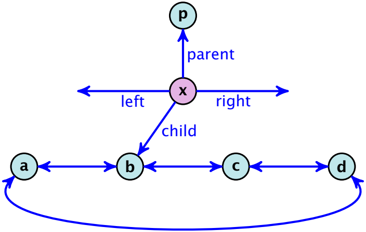
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have children then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

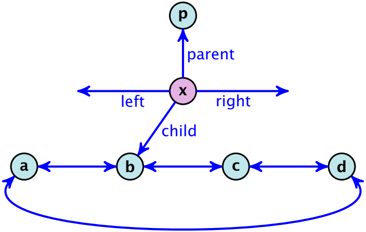
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have children then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

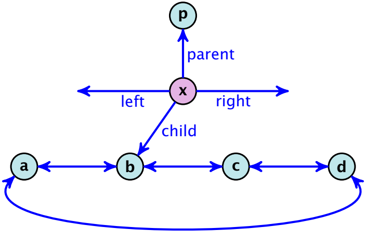
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have children then $x.\text{left} = x.\text{right} = x$).



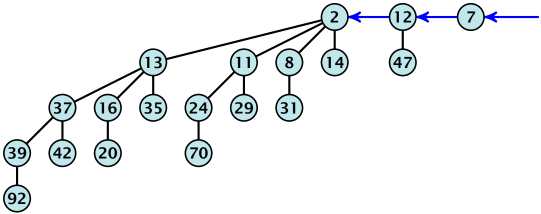
8.2 Binomial Heaps

How do we implement trees with non-constant degree?

- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have children then $x.\text{left} = x.\text{right} = x$).



Binomial Heap

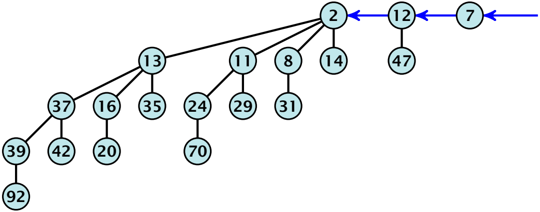


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap

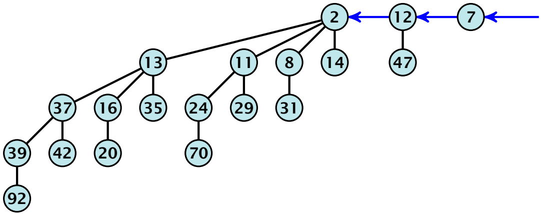


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap

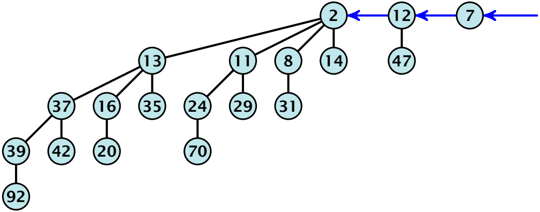


In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap



In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

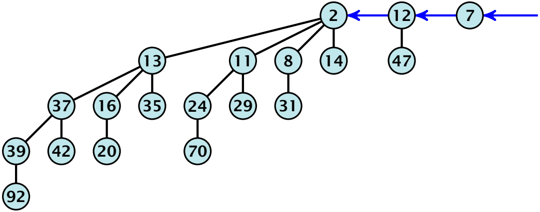
Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap

Properties of a heap with n keys:

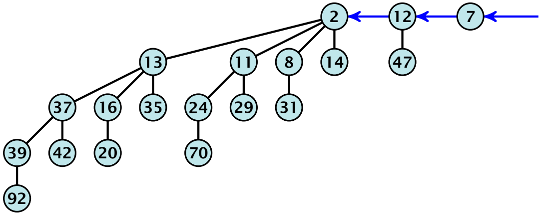
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

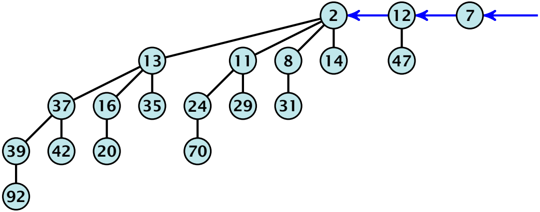
- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

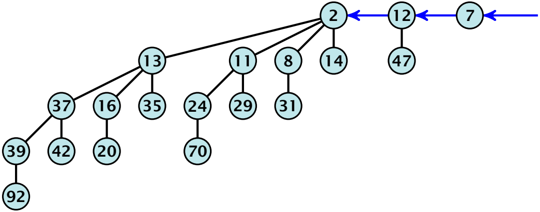
- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

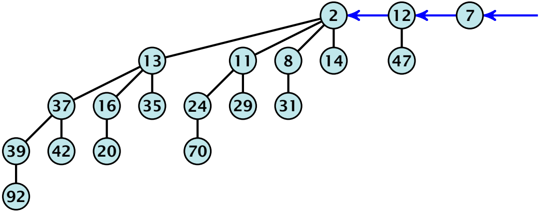
- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

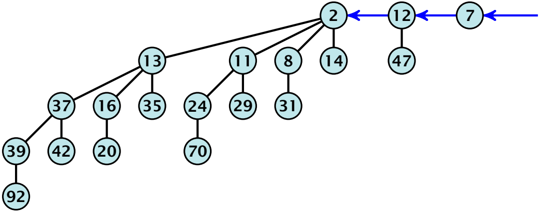
- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

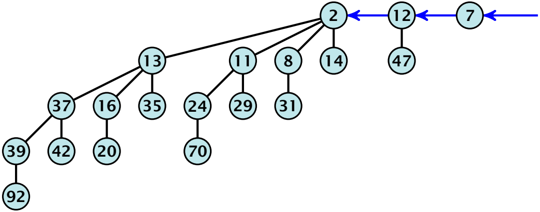
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap

Properties of a heap with n keys:

- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



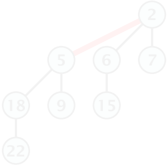
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



For more trees the technique is analogous to binary addition.

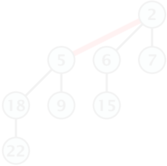
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



For more trees the technique is analogous to binary addition.

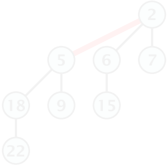
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



For more trees the technique is analogous to binary addition.

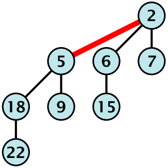
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

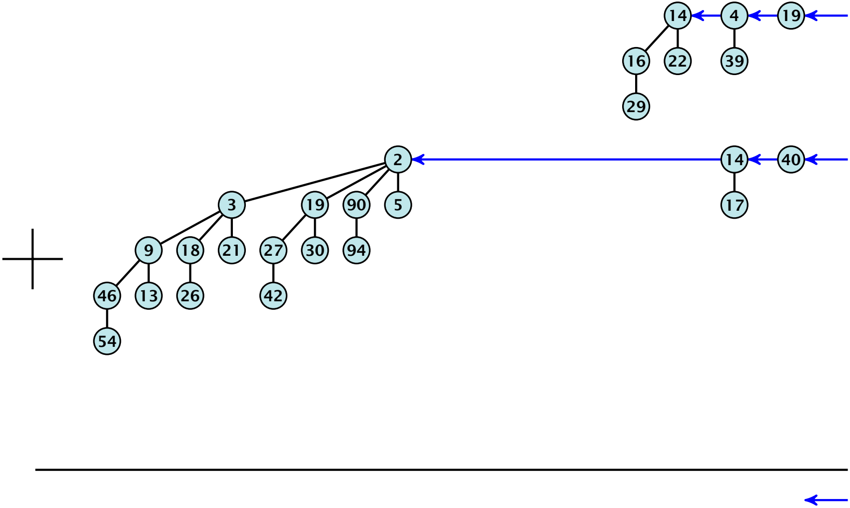
A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

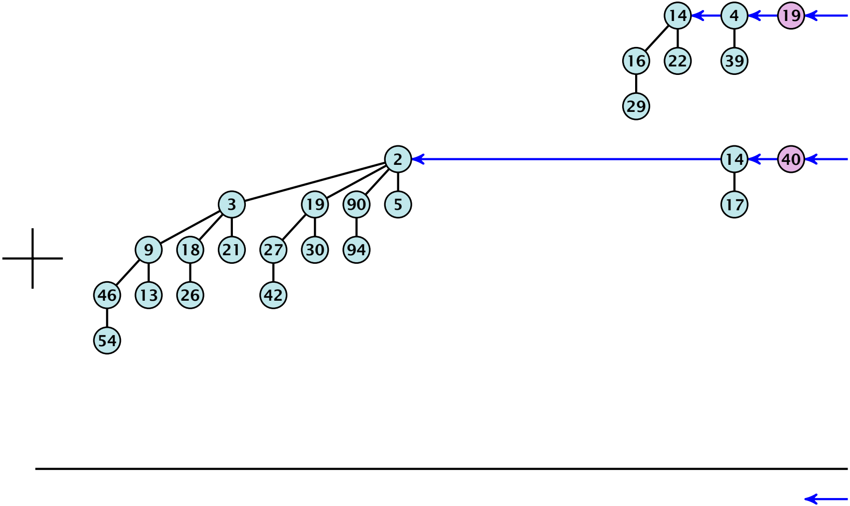
Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

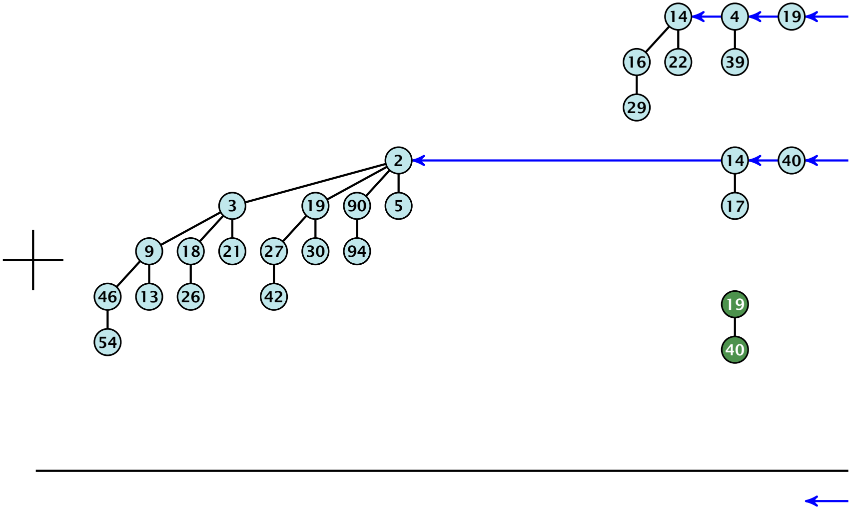
Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.

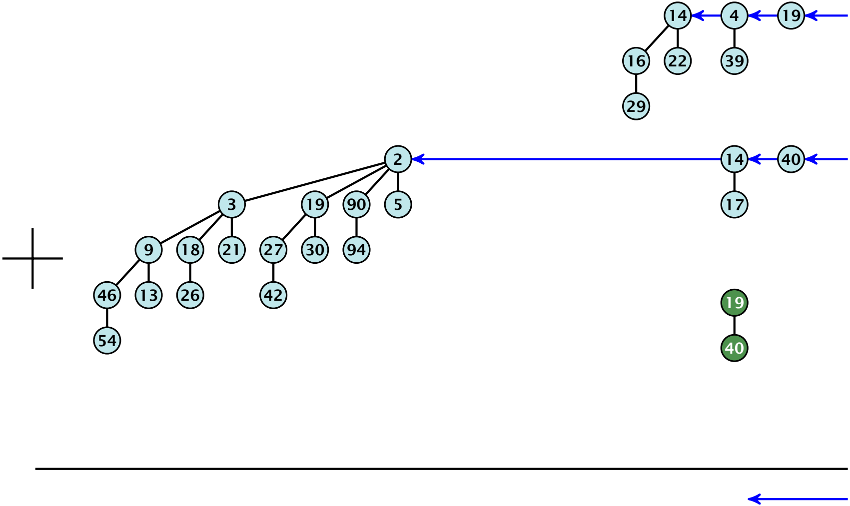


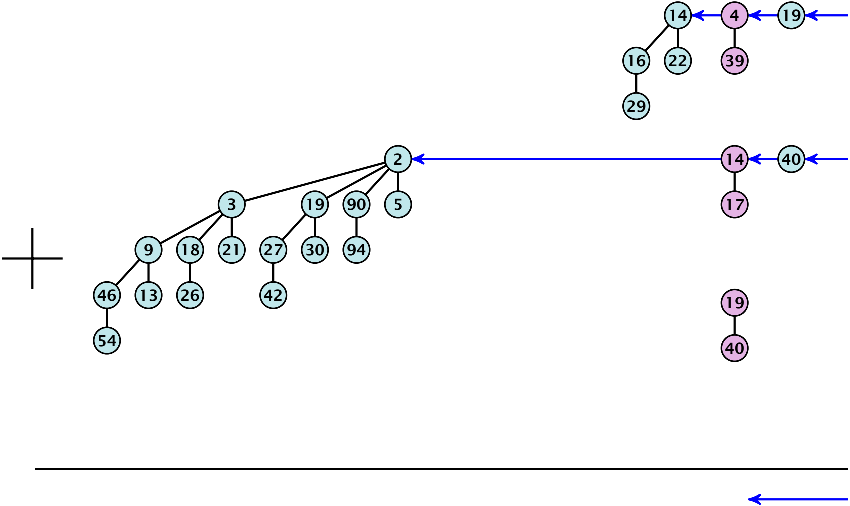
For more trees the technique is analogous to binary addition.

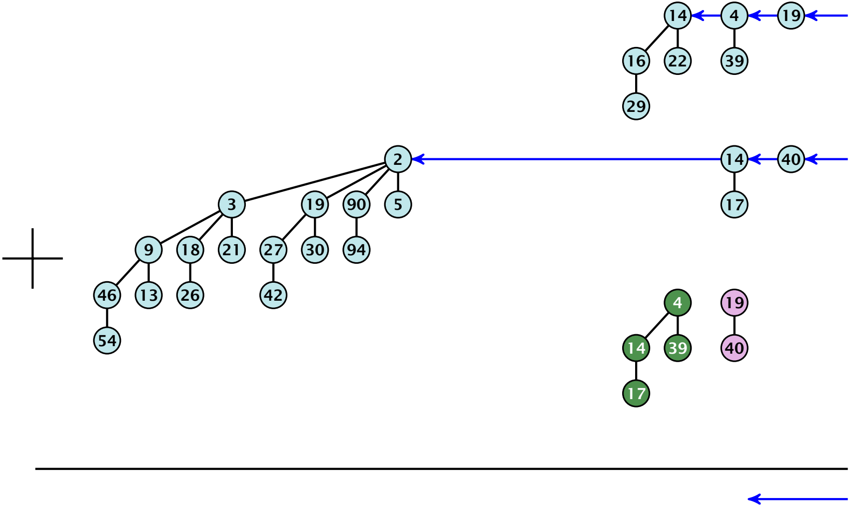

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54

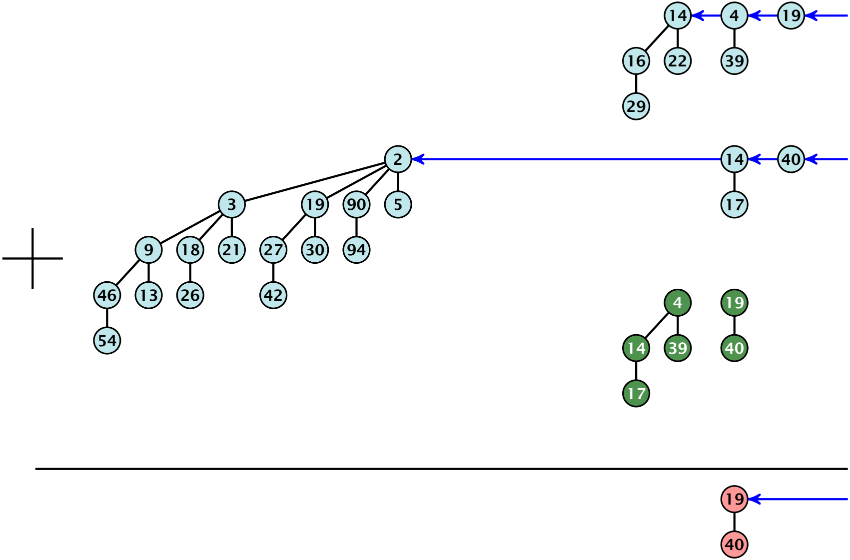

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

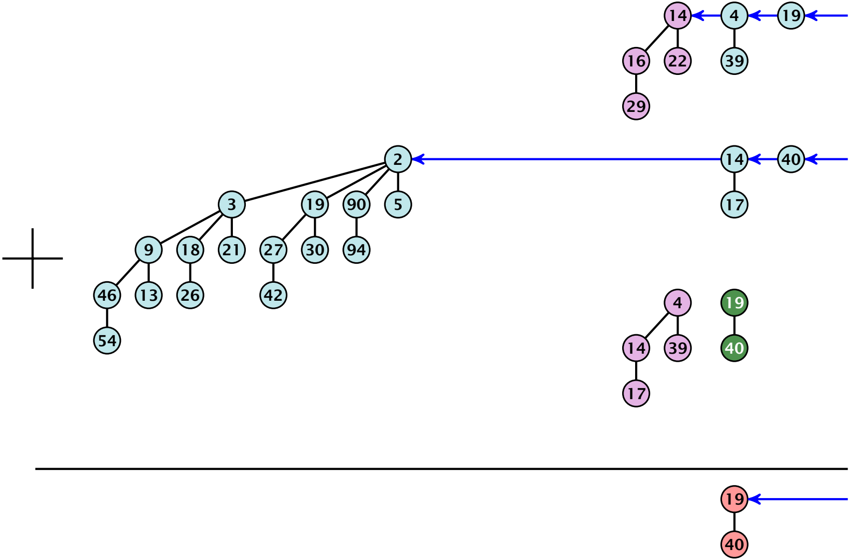

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

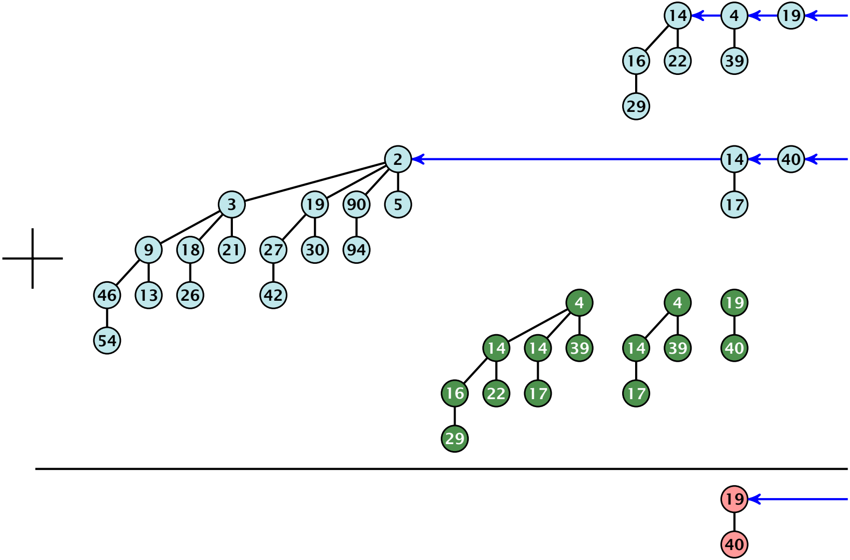

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

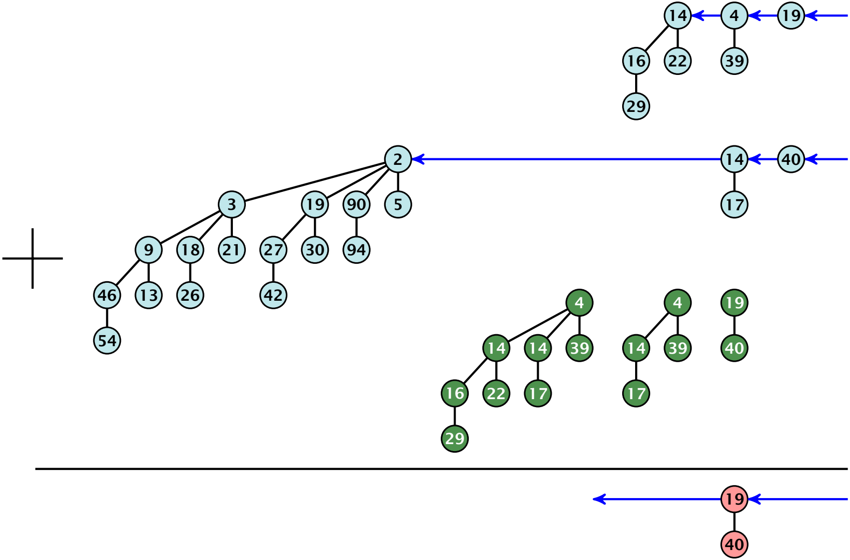

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
19
14
39
40
17

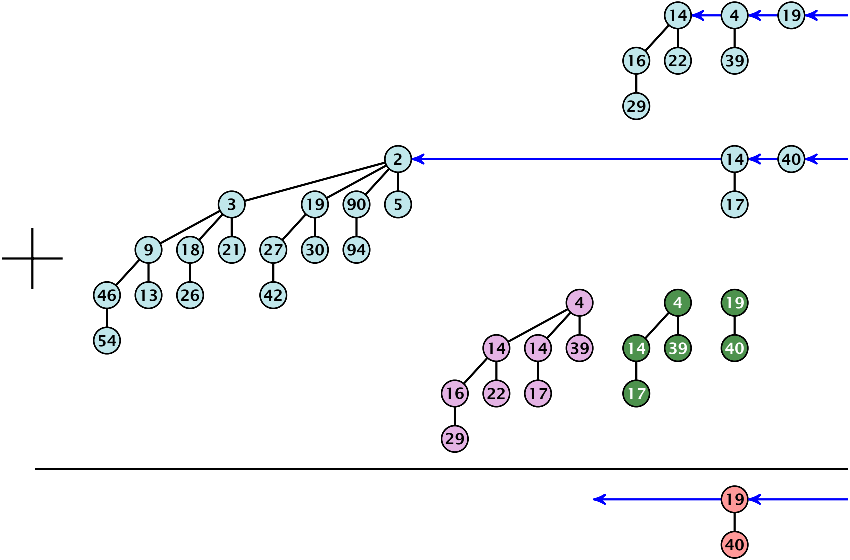


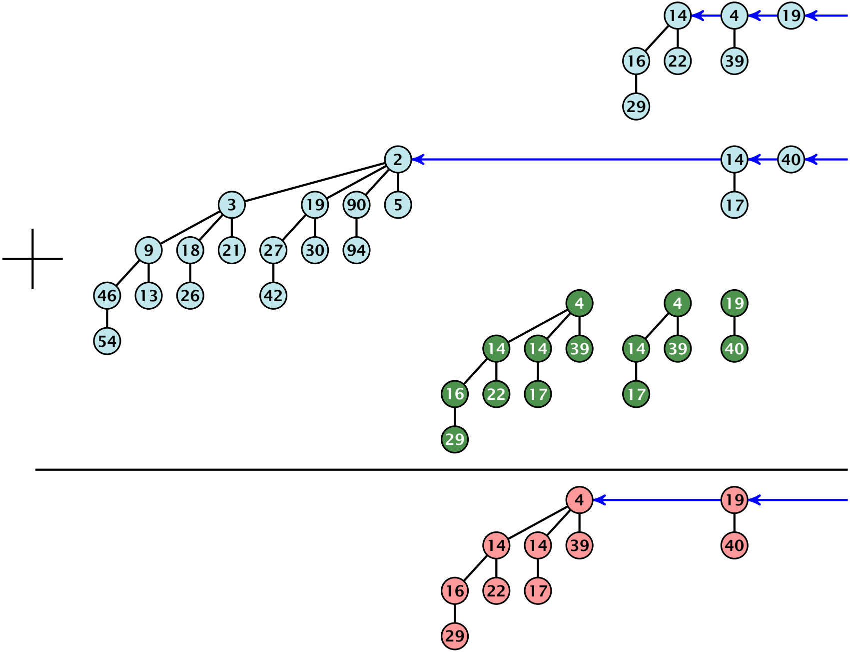
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
19
14
39
40
17
19
40

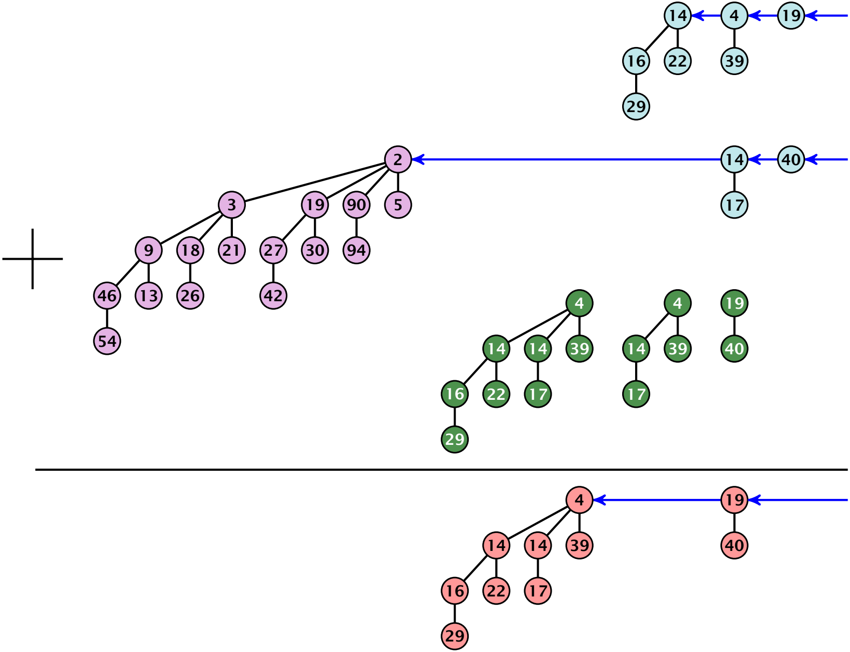

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
19
14
39
40
17
19
40

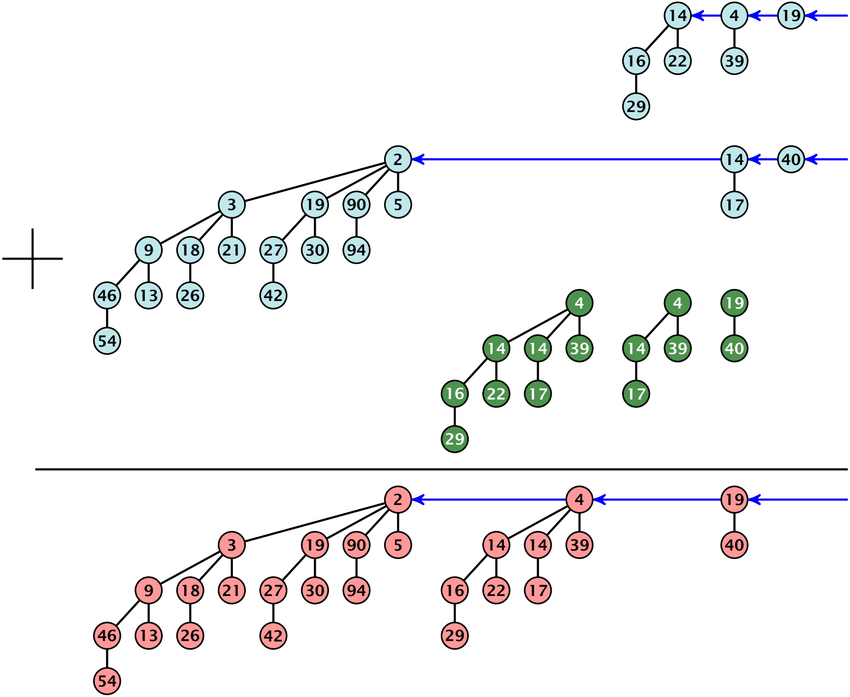

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40

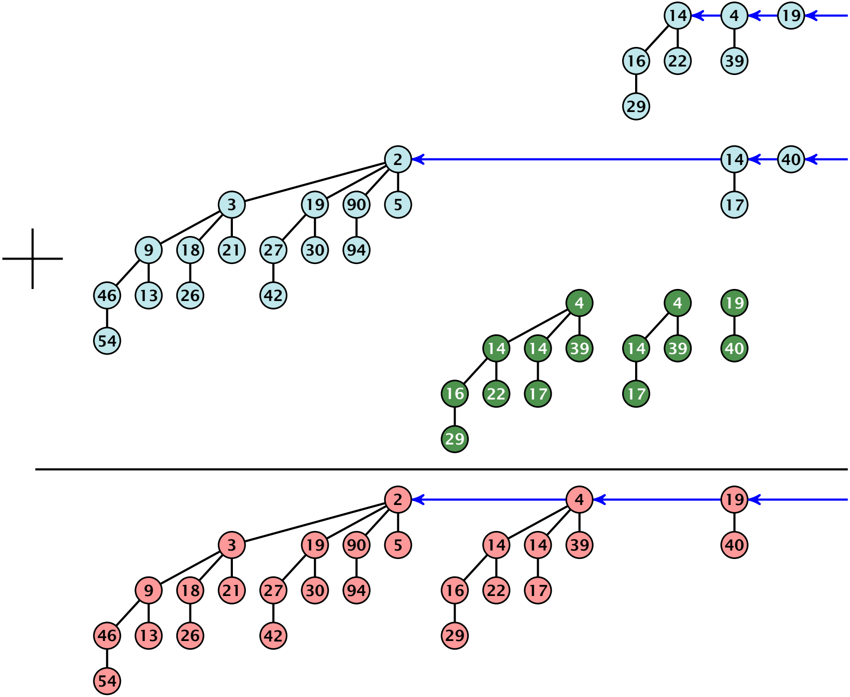

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
4
19
14
14
39
40
16
22
17
29


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
4
19
14
14
39
40
16
22
17
29


14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
+
4 4 19
14 14 39 14 39 40
16 22 17 17
29
2 4 19
3 19 90 5 14 14 39 40
9 18 21 27 30 94 16 22 17
46 13 26 42 29
54


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
2
4
19
3
19
90
5
14
14
39
40
9
18
21
27
30
94
16
22
17
46
13
26
42
29
54

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

***S*.insert(*x*):**

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.minimum():

- Find the minimum key-value among all roots.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

***S*.delete(handle *h*):**

- Execute S.decrease-key($h, -\infty$).
- Execute S.delete-min().
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min().
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min$()$.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min$()$.
- Time: $\mathcal{O}(\log n)$.

Amortized Analysis

Definition 32

A data structure with operations $\mathrm{op}_1(), \dots, \mathrm{op}_k()$ has amortized running times $t_1, \dots, t_k$ for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structre) that operate on at most n elements, and let k_i denote the number of occurences of $\mathrm{op}_i()$ within this sequence. Then the actual running time must be at most $\sum_i k_i t_i(n)$.

Amortized Analysis

Definition 32

A data structure with operations $\mathrm{op}_1(), \ldots, \mathrm{op}_k()$ has amortized running times $t_1, \ldots, t_k$ for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structre) that operate on at most n elements, and let k_i denote the number of occurences of $\mathrm{op}_i()$ within this sequence. Then the actual running time must be at most $\sum_i k_i t_i(n)$.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \enspace .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \enspace .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \enspace .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Example: Stack

Stack

- $\mathbf{S.\,push()}$
- $\mathbf{S.\,pop()}$
- $\mathbf{S.\,multipop(k)}$**:** removes k items from the stack. If the stack currently contains less than k items it empties the stack.

Actual cost:

- $\mathbf{S.\,push()}$: cost 1.
- $\mathbf{S.\,pop()}$: cost 1.
- $\mathbf{S.\,multipop(k)}$: cost $\min\{\text{size}, k\}$.

Example: Stack

Stack

- $\mathbf{S.\,push()}$
- $\mathbf{S.\,pop()}$
- $\mathbf{S.\,multipop(}\boldsymbol{k}\mathbf{)}$**:** removes k items from the stack. If the stack currently contains less than k items it empties the stack.

Actual cost:

- $\mathbf{S.\,push()}$: cost 1.
- $\mathbf{S.\,pop()}$: cost 1.
- $\mathbf{S.\,multipop(}\boldsymbol{k}\mathbf{)}$: cost $\min\{\text{size}, k\}$.

Example: Stack

Use potential function $\Phi(S) =$ number of elements on the stack.

Amortized cost:

- $S.$ push(): cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- $S.$ pop(): cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- $S.$ multipop(k): cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \le 0 \ .$$

Example: Stack

Use potential function $\Phi(S)$ = number of elements on the stack.

Amortized cost:

- $\boldsymbol{S.\,\mathrm{push}()}$: cost

$$\hat{C}_{\mathrm{push}} = C_{\mathrm{push}} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- $\boldsymbol{S.\,\mathrm{pop}()}$: cost

$$\hat{C}_{\mathrm{pop}} = C_{\mathrm{pop}} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- $\boldsymbol{S.\,\mathrm{multipop}(k)}$: cost

$$\hat{C}_{\mathrm{mp}} = C_{\mathrm{mp}} + \Delta\Phi = \min\{\mathrm{size}, k\} - \min\{\mathrm{size}, k\} \leq 0 \ .$$

Example: Stack

Use potential function $\Phi(S)$ = number of elements on the stack.

Amortized cost:

- $\boldsymbol{S.\mathrm{push}()}$: cost

$$\hat{C}_{\mathrm{push}} = C_{\mathrm{push}} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- $\boldsymbol{S.\mathrm{pop}()}$: cost

$$\hat{C}_{\mathrm{pop}} = C_{\mathrm{pop}} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- $\boldsymbol{S.\mathrm{multipop}(k)}$: cost

$$\hat{C}_{\mathrm{mp}} = C_{\mathrm{mp}} + \Delta\Phi = \min\{\mathrm{size}, k\} - \min\{\mathrm{size}, k\} \le 0 \ .$$

Example: Stack

Use potential function $\Phi(S)$ = number of elements on the stack.

Amortized cost:

- $\boldsymbol{S}$. **push()**: cost

  $$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- $\boldsymbol{S}$. **pop()**: cost

  $$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- $\boldsymbol{S}$. **multipop(*k*)**: cost

  $$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \leq 0 \ .$$

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an n-bit binary counter may require to examine n-bits, and maybe change them.

Actual cost:

- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k + 1$, where k is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k = 1$).

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an n-bit binary counter may require to examine n-bits, and maybe change them.

Actual cost:

- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k + 1$, where k is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k = 1$).

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an n-bit binary counter may require to examine n-bits, and maybe change them.

Actual cost:

- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k+1$, where k is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k=1$).

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1: cost

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \le 2 .$$

- Changing bit from 1 to 0: cost 0.

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \le 0 .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1: cost

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \le 2 \enspace .$$

- Changing bit from 1 to 0: cost 0.

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \le 0 \enspace .$$

- Increment. Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1: cost

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- Changing bit from 1 to 0: cost 0.

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- Increment. Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1: cost

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \leq 2 \enspace .$$

- Changing bit from 1 to 0: cost 0.

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \leq 0 \enspace .$$

- Increment. Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1: cost

$$\hat{C}_{0\to1} = C_{0\to1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- Changing bit from 1 to 0: cost 0.

$$\hat{C}_{1\to0} = C_{1\to0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

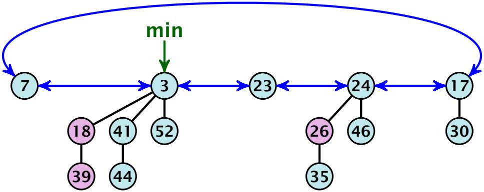
- Increment. Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to0} + \hat{C}_{0\to1} \le 2$.

8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

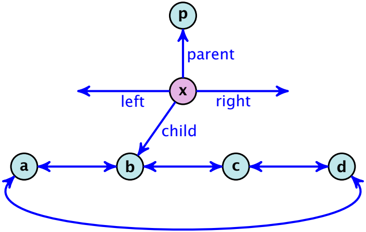
Structure is much more relaxed than binomial heaps.



8.3 Fibonacci Heaps

How do we implement trees with non-constant degree?

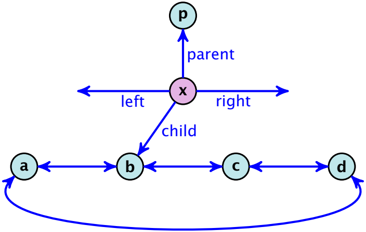
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.3 Fibonacci Heaps

How do we implement trees with non-constant degree?

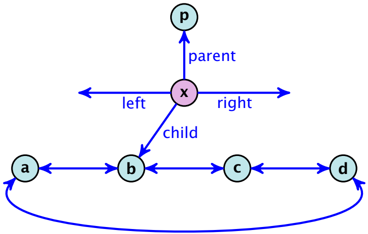
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.3 Fibonacci Heaps

How do we implement trees with non-constant degree?

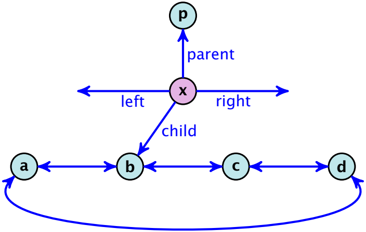
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.3 Fibonacci Heaps

How do we implement trees with non-constant degree?

- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\mathrm{left}$ and $x.\mathrm{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\mathrm{left} = x.\mathrm{right} = x$).



8.3 Fibonacci Heaps

- Given a pointer to a node x we can splice out the sub-tree rooted at x in constant time.
- We can add a child-tree T to a node x in constant time if we are given a pointer to x and a pointer to the root of T.

8.3 Fibonacci Heaps

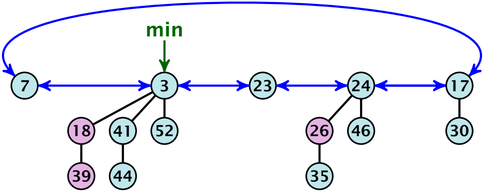
Additional implementation details:

- Every node x stores its degree in a field $x.\mathrm{degree}$. Note that this can be updated in constant time when adding a child to x.
- Every node stores a boolean value $x.\mathrm{marked}$ that specifies whether x is marked or not.

8.3 Fibonacci Heaps

The potential function:

- $t(S)$ denotes the number of trees in the heap.
- $m(S)$ denotes the number of marked nodes.
- We use the potential function $\Phi(S) = t(S) + 2m(S)$.



The potential is $\Phi(S) = 5 + 2 \cdot 3 = 11$.

8.3 Fibonacci Heaps

We assume that one unit of potential can pay for a constant amount of work, where the constant is chosen “big enough” (to take care of the constants that occur).

To make this more explicit we use c to denote the amount of work that a unit of potential can pay for.

8.3 Fibonacci Heaps

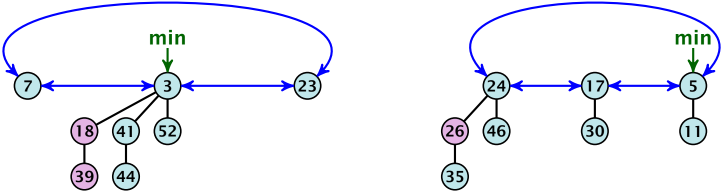
$S.$ **minimum()**

- Access through the min-pointer.
- Actual cost $\mathcal{O}(1)$.
- No change in potential.
- Amortized cost $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

$S.\mathbf{merge}(S')$

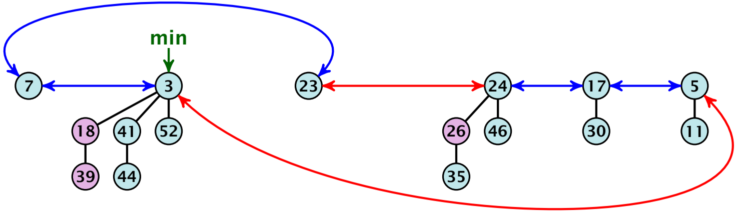
- Merge the root lists.
- Adjust the min-pointer



8.3 Fibonacci Heaps

$S.\mathbf{merge}(S')$

- Merge the root lists.
- Adjust the min-pointer



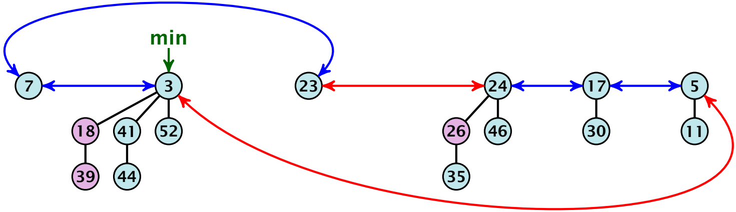
Running time:

- Actual cost $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

$S.\mathbf{merge}(S')$

- Merge the root lists.
- Adjust the min-pointer



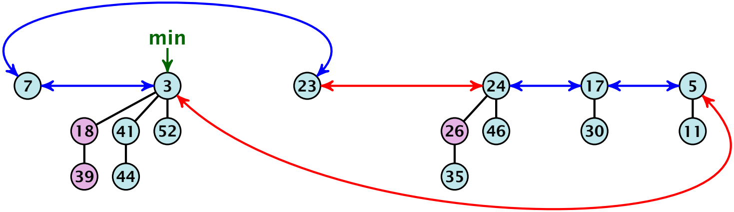
Running time:

- Actual cost $\mathcal{O}(1)$.
- No change in potential.

8.3 Fibonacci Heaps

$S.\,\mathbf{merge}(S')$

- Merge the root lists.
- Adjust the min-pointer



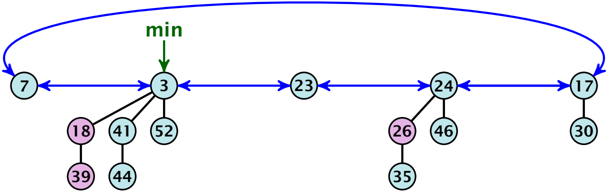
Running time:

- Actual cost $\mathcal{O}(1)$.
- No change in potential.
- Hence, amortized cost is $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

$S.\,\mathrm{insert}(x)$

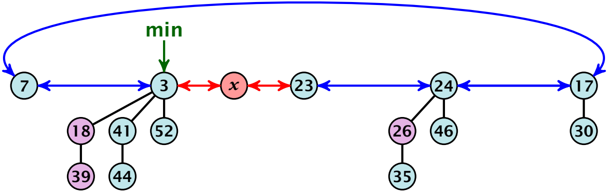
- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.



8.3 Fibonacci Heaps

$S.$ **insert**(x)

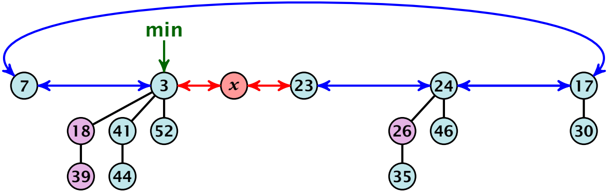
- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.



8.3 Fibonacci Heaps

***S*. insert(*x*)**

- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.

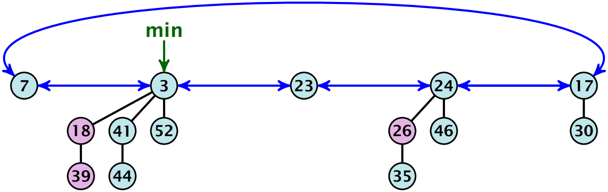


Running time:

- Actual cost $\mathcal{O}(1)$.
- Change in potential is $+1$.
- Amortized cost is $c + \mathcal{O}(1) = \mathcal{O}(1)$.

8.3 Fibonacci Heaps

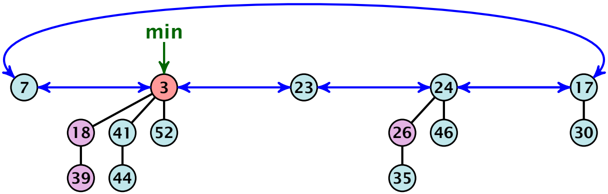
$S.$ **delete-min**(x)



8.3 Fibonacci Heaps

$S.$ delete-min(x)

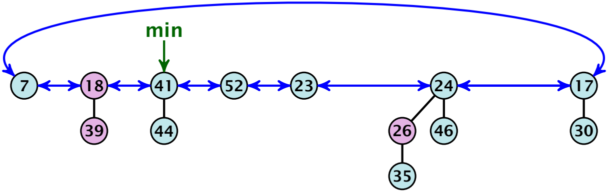
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

S. delete-min(x)

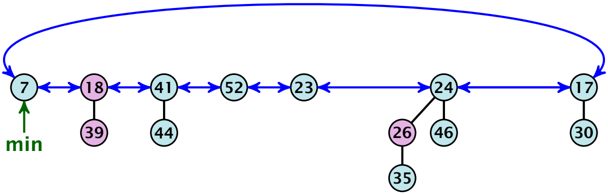
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

S. delete-min(x)

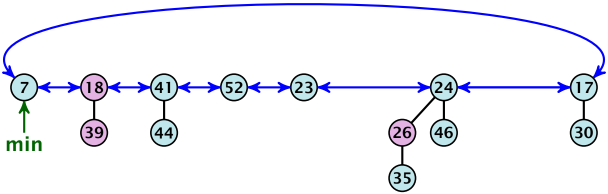
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

S. delete-min(x)

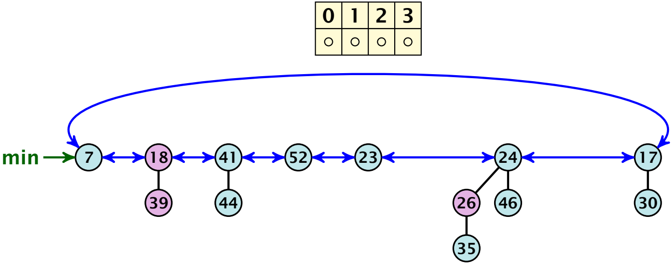
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



- Consolidate root-list so that no roots have the same degree. Time $t \cdot \mathcal{O}(1)$ (see next slide).

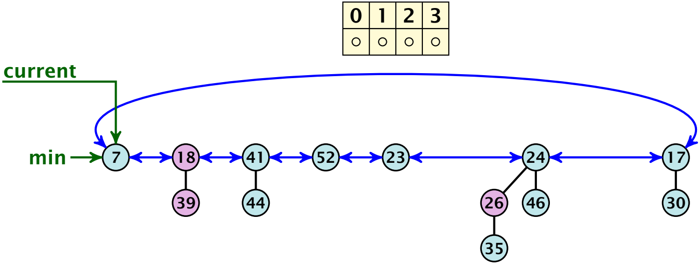
8.3 Fibonacci Heaps

Consolidate:



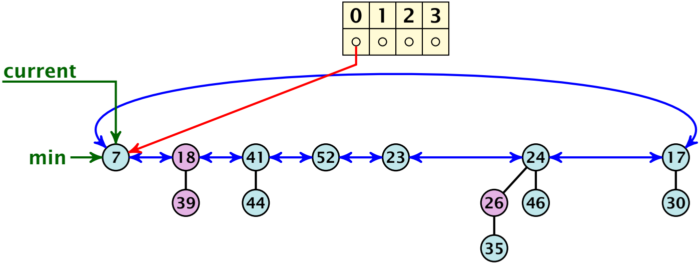
8.3 Fibonacci Heaps

Consolidate:



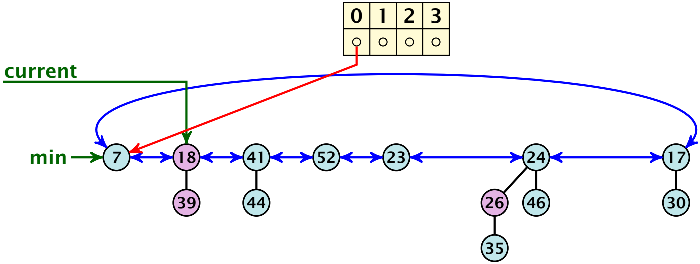
8.3 Fibonacci Heaps

Consolidate:



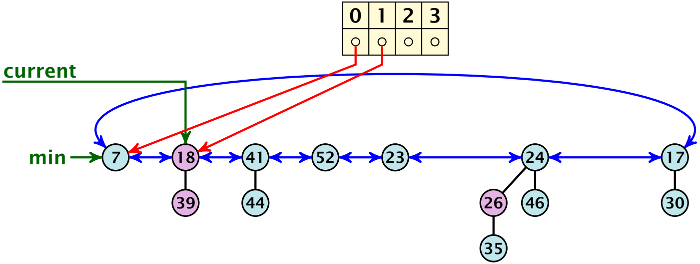
8.3 Fibonacci Heaps

Consolidate:



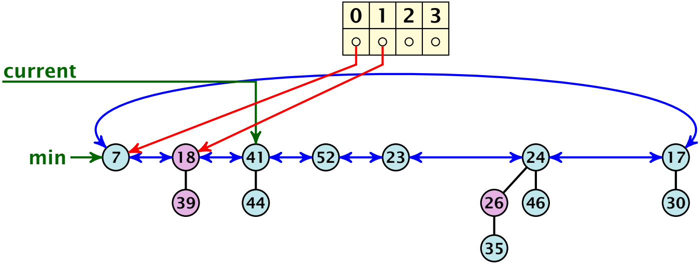
8.3 Fibonacci Heaps

Consolidate:



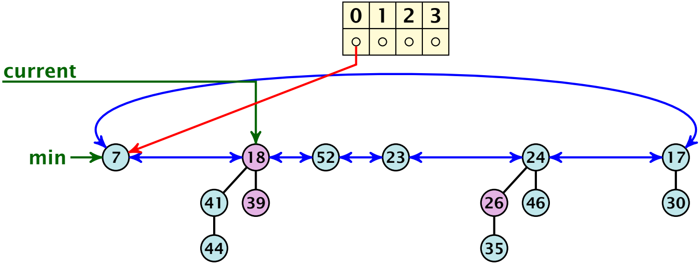
8.3 Fibonacci Heaps

Consolidate:



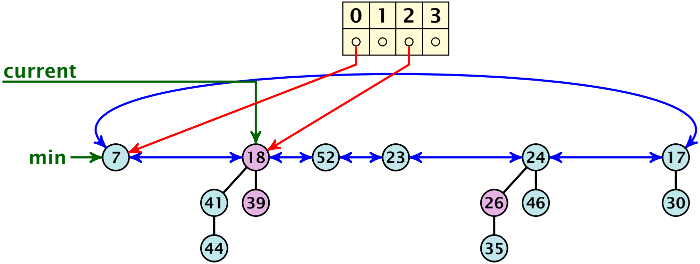
8.3 Fibonacci Heaps

Consolidate:



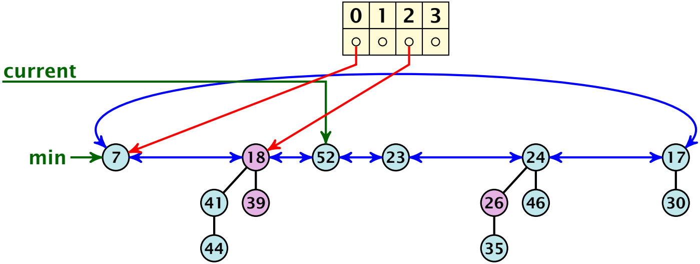
8.3 Fibonacci Heaps

Consolidate:



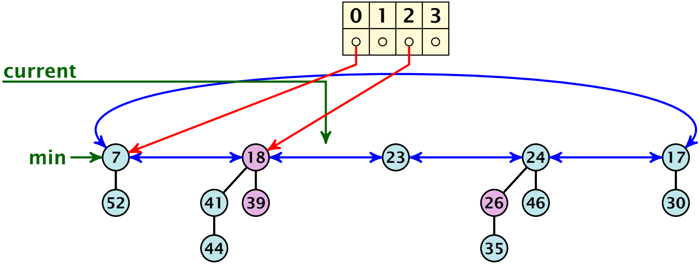
8.3 Fibonacci Heaps

Consolidate:



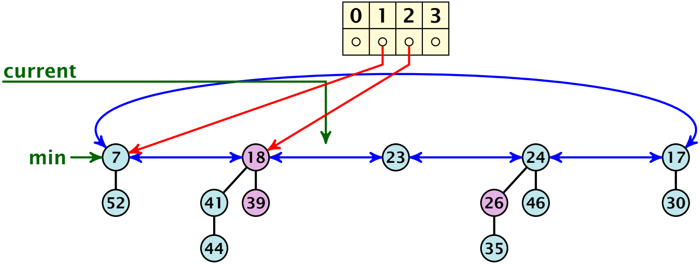
8.3 Fibonacci Heaps

Consolidate:



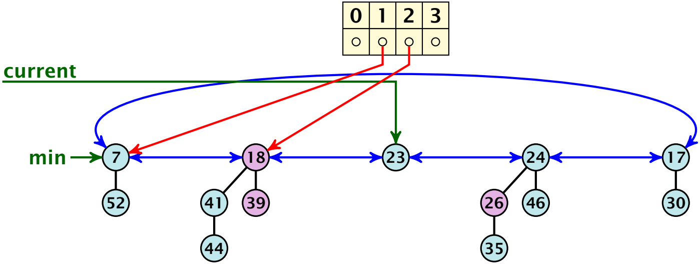
8.3 Fibonacci Heaps

Consolidate:



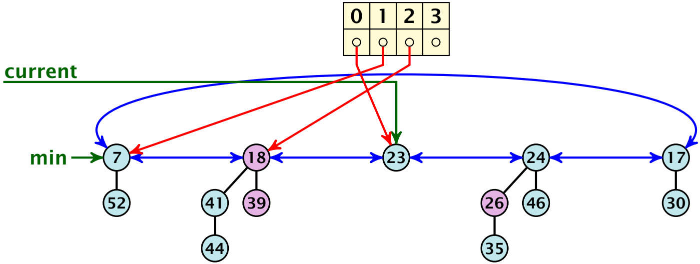
8.3 Fibonacci Heaps

Consolidate:



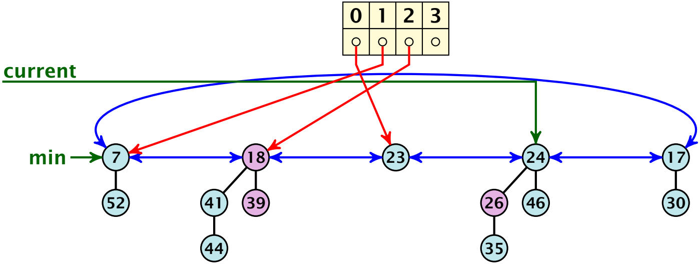
8.3 Fibonacci Heaps

Consolidate:



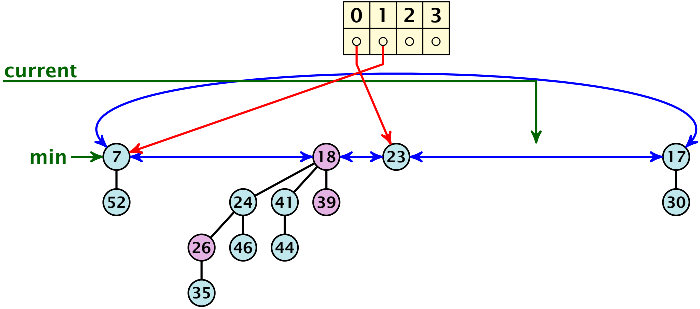
8.3 Fibonacci Heaps

Consolidate:



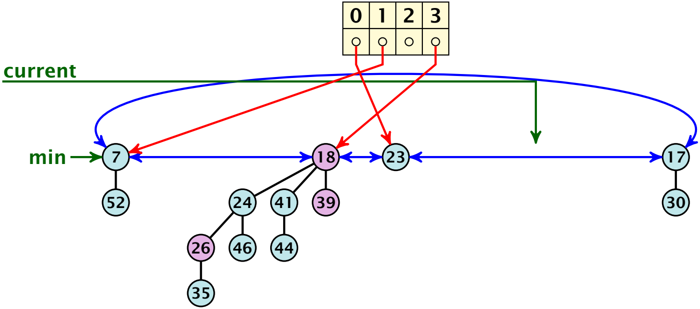
8.3 Fibonacci Heaps

Consolidate:



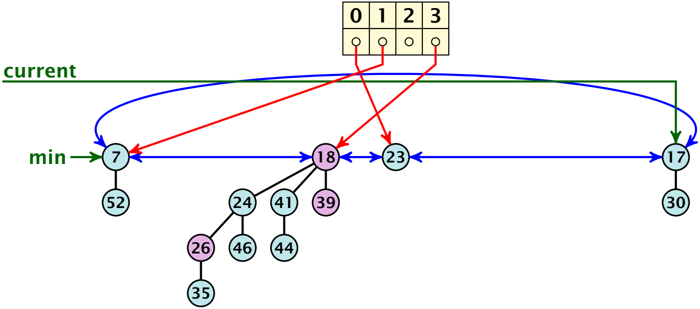
8.3 Fibonacci Heaps

Consolidate:



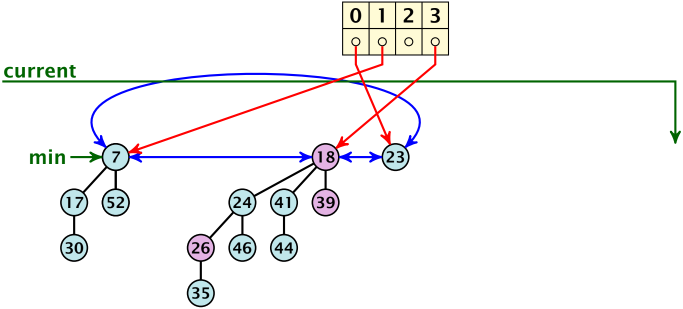
8.3 Fibonacci Heaps

Consolidate:



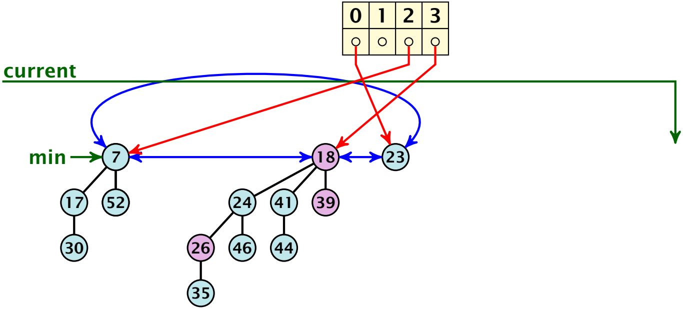
8.3 Fibonacci Heaps

Consolidate:



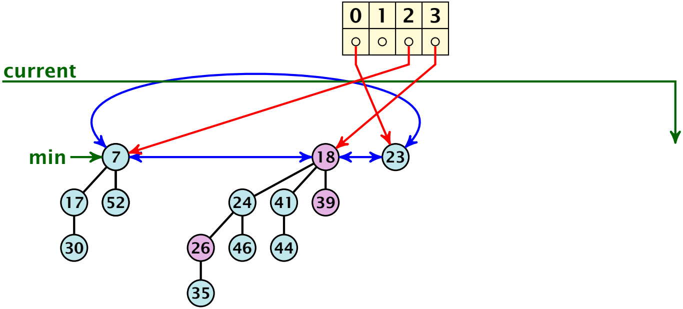
8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.

Amortized cost for delete-min()

- $t' \le D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \le D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1)$$

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$
\begin{aligned}
&c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\
&\qquad\qquad \leq (c_1 + c)D_n + (c_1 - c)t + c
\end{aligned}
$$

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$\begin{aligned} & c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\ & \qquad \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \end{aligned}$$

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$\begin{aligned} & c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\ & \qquad \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq \mathcal{O}(D_n) \end{aligned}$$

8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$.
  Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \le D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \le D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$
\begin{aligned}
&c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\
&\qquad \le (c_1 + c)D_n + (c_1 - c)t + c \le 2c(D_n + 1) \le \mathcal{O}(D_n)
\end{aligned}
$$

for $c \ge c_1$.

8.3 Fibonacci Heaps

If the input trees of the consolidation procedure are binomial trees (for example only singleton vertices) then the output will be a set of distinct binomial trees, and, hence, the Fibonacci heap will be (more or less) a Binomial heap right after the consolidation.

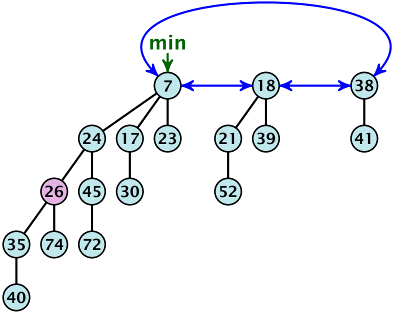
If we do not have delete or decrease-key operations then $D_n \leq \log n$.

8.3 Fibonacci Heaps

If the input trees of the consolidation procedure are binomial trees (for example only singleton vertices) then the output will be a set of distinct binomial trees, and, hence, the Fibonacci heap will be (more or less) a Binomial heap right after the consolidation.

If we do not have delete or decrease-key operations then $D_n \leq \log n$.

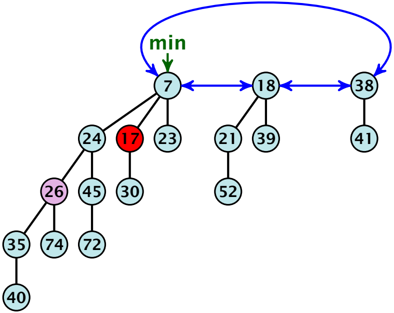
Fibonacci Heaps: decrease-key(handle h, v)



Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by h. Nothing else to do.

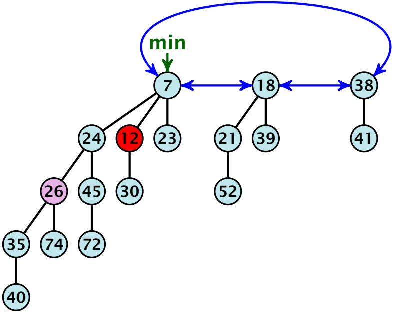
Fibonacci Heaps: decrease-key(handle h, v)



Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by h. Nothing else to do.

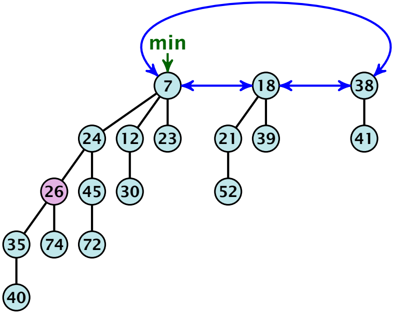
Fibonacci Heaps: decrease-key(handle h, v)



Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by h. Nothing else to do.

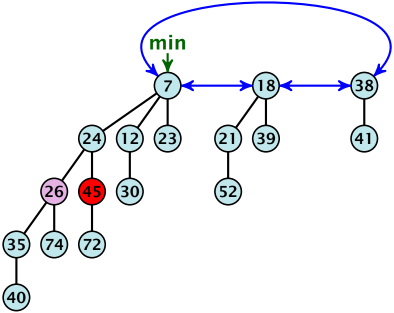
Fibonacci Heaps: decrease-key(handle h, v)



Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by h. Nothing else to do.

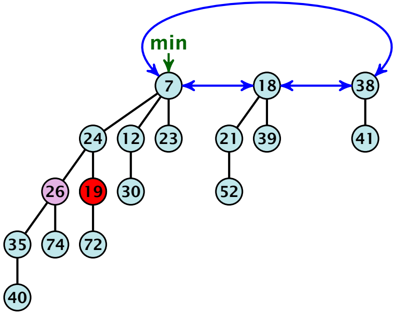
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x.

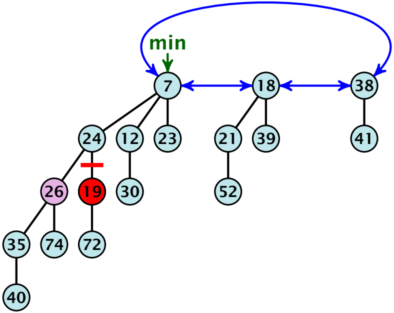
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x.

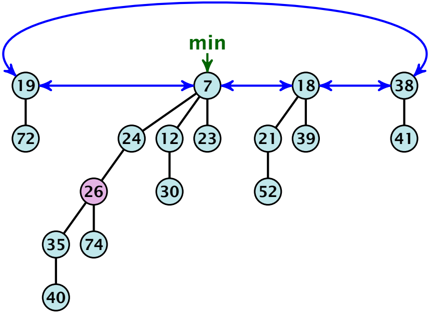
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x.

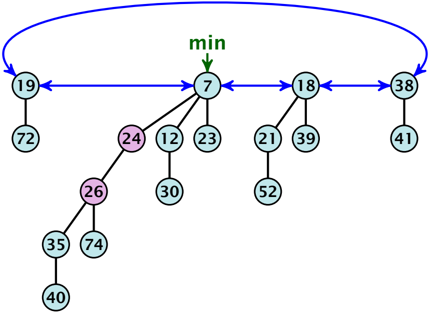
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x.

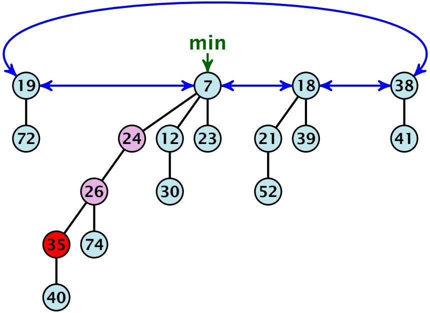
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x.

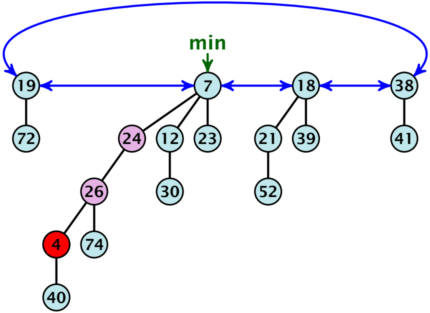
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

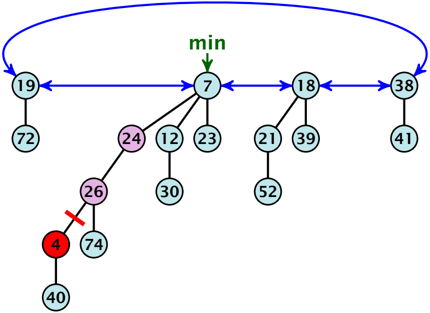
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

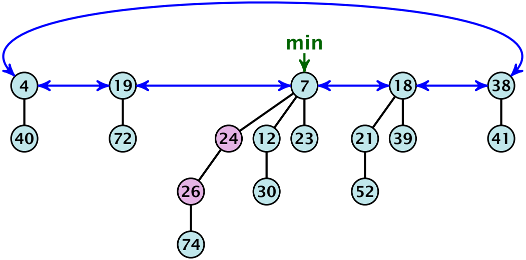
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

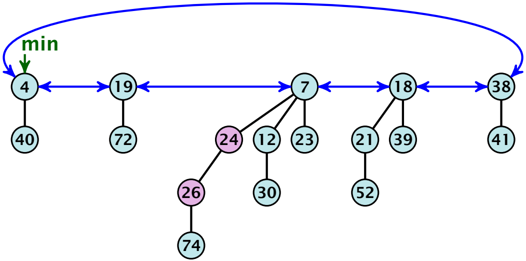
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

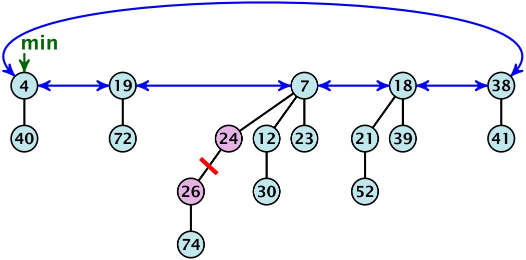
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

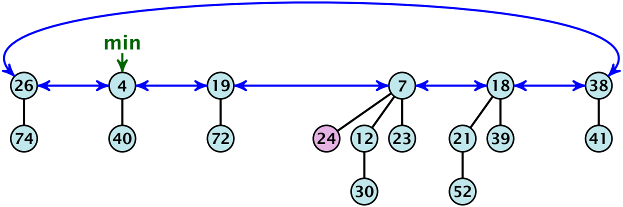
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

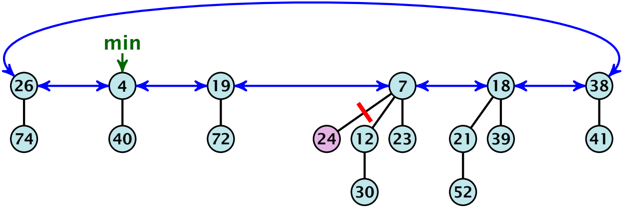
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

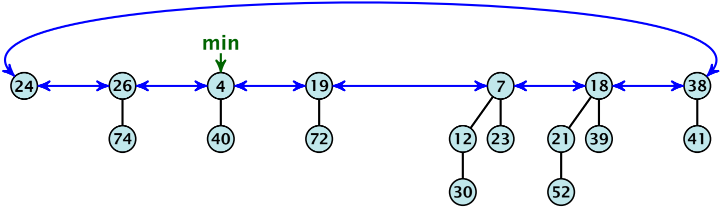
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

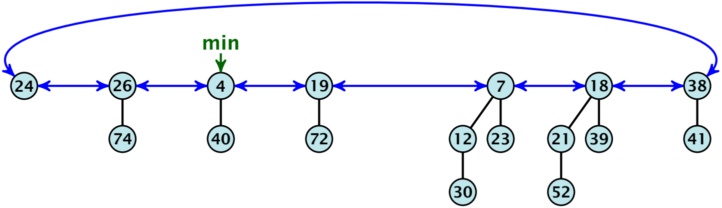
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

Fibonacci Heaps: decrease-key(handle h, v)

Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Execute the following:

```
$p \leftarrow \text{parent}[x]$;
while ($p$ is marked)
    $pp \leftarrow \text{parent}[p]$;
    cut of $p$; make it into a root; unmark it;
    $p \leftarrow pp$;
if $p$ is unmarked and not a root mark it;
```

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

if $c \ge c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

  if $c \ge c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

  $$c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c = \mathcal{O}(1) ,$$

  if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c = \mathcal{O}(1)\,,$$

if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

  $$c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c = \mathcal{O}(1),$$

  if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \le (c_2 - c)\ell + 4c = \mathcal{O}(1)\ ,$$

if $c \ge c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \le (c_2 - c)\ell + 4c = \mathcal{O}(1)\,,$$

if $c \ge c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c = \mathcal{O}(1)\ ,$$

if $c \geq c_2$.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut marks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

$$c_2(\ell + 1) + c(4 - \ell) \le (c_2 - c)\ell + 4c = \mathcal{O}(1)\,,$$

if $c \ge c_2$.

Delete node

$\boldsymbol{H}.\,\textbf{delete}(\boldsymbol{x})$:

- decrease value of x to $-\infty$.
- delete-min.

Amortized cost: $\mathcal{O}(D(n))$

- $\mathcal{O}(1)$ for decrease-key.
- $\mathcal{O}(D(n))$ for delete-min.

8.3 Fibonacci Heaps

Lemma 33

Let x be a node with degree k and let $y_1, \dots, y_k$ denote the children of x in the order that they were linked to x. Then

$$\mathrm{degree}(y_i) \geq \begin{cases} 0 & \textit{if } i = 1 \\ i - 2 & \textit{if } i \geq 1 \end{cases}$$

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \ldots, y_{i-1}$ were already linked to x.
- Hence, at this time $\mathrm{degree}(x) \geq i-1$, and therefore also $\mathrm{degree}(y_i) \geq i-1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\mathrm{degree}(y_i) \geq i-2$.

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \dots, y_{i-1}$ were already linked to x.
- Hence, at this time $\text{degree}(x) \geq i - 1$, and therefore also $\text{degree}(y_i) \geq i - 1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\text{degree}(y_i) \geq i - 2$.

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \ldots, y_{i-1}$ were already linked to x.
- Hence, at this time $\mathrm{degree}(x) \geq i - 1$, and therefore also $\mathrm{degree}(y_i) \geq i - 1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\mathrm{degree}(y_i) \geq i - 2$.

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \dots, y_{i-1}$ were already linked to x.
- Hence, at this time $\mathrm{degree}(x) \geq i - 1$, and therefore also $\mathrm{degree}(y_i) \geq i - 1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\mathrm{degree}(y_i) \geq i - 2$.

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k
- $s_0 = 1$ and $s_1 = 2$.

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k
- $s_0 = 1$ and $s_1 = 2$.

Let x be a degree k node of size s_k and let $y_1, \dots, y_k$ be its children.

$$s_k = 2 + \sum_{i=2}^{k} \text{size}(y_i)$$

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k
- $s_0 = 1$ and $s_1 = 2$.

Let x be a degree k node of size s_k and let $y_1, \ldots, y_k$ be its children.

$$\begin{aligned} s_k &= 2 + \sum_{i=2}^{k} \operatorname{size}(y_i) \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \end{aligned}$$

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k
- $s_0 = 1$ and $s_1 = 2$.

Let x be a degree k node of size s_k and let $y_1, \ldots, y_k$ be its children.

$$\begin{aligned} s_k &= 2 + \sum_{i=2}^{k} \mathrm{size}(y_i) \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &= 2 + \sum_{i=0}^{k-2} s_i \end{aligned}$$

8.3 Fibonacci Heaps

Definition 34

Consider the following non-standard Fibonacci type sequence:

$$F_k = \begin{cases} 1 & \text{if } k = 0 \\ 2 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Facts:

1. $F_k \geq \phi^k$.
2. For $k \geq 2$: $F_k = 2 + \sum_{i=0}^{k-2} F_i$.

The above facts can be easily proved by induction. From this it follows that $s_k \geq F_k \geq \phi^k$, which gives that the maximum degree in a Fibonacci heap is logarithmic.

9 van Emde Boas Trees

Dynamic Set Data Structure S:

- $S.\text{insert}(x)$
- $S.\text{delete}(x)$
- $S.\text{search}(x)$
- $S.\text{min}()$
- $S.\text{max}()$
- $S.\text{succ}(x)$
- $S.\text{pred}(x)$

9 van Emde Boas Trees

For this chapter we ignore the problem of storing satellite data:

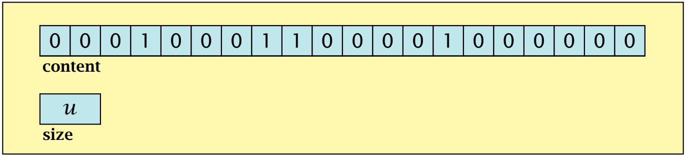
- **S. insert(x):** Inserts x into S.
- **S. delete(x):** Deletes x from S. Usually assumes that $x \in S$.
- **S. member(x):** Returns 1 if $x \in S$ and 0 otw.
- **S. min():** Returns the value of the minimum element in S.
- **S. max():** Returns the value of the maximum element in S.
- **S. succ(x):** Returns successor of x in S. Returns null if x is maximum or larger than any element in S. Note that x needs not to be in S.
- **S. pred(x):** Returns the predecessor of x in S. Returns null if x is minimum or smaller than any element in S. Note that x needs not to be in S.

9 van Emde Boas Trees

Can we improve the existing algorithms when the keys are from a restricted set?

In the following we assume that the keys are from $\{0, 1, \ldots, u - 1\}$, where u denotes the size of the universe.

Implementation 1: Array



0	0	0	1	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0	0

content

u

size

one array of u bits

Use an array that encodes the indicator function of the dynamic set.

Implementation 1: Array

```
Algorithm 19 array.insert($x$)
content[$x$] ← 1;
```

```
Algorithm 20 array.delete($x$)
content[$x$] ← 0;
```

```
Algorithm 21 array.member($x$)
return content[$x$];
```

- Note that we assume that x is valid, i.e., it falls within the array boundaries.
- Obviously(?) the running time is constant.

Implementation 1: Array

Algorithm 22 array.max()

```
for ($i$ = size − 1; $i \geq 0$; $i$−−) do
    if content[$i$] = 1 then return $i$;
return null;
```

Algorithm 23 array.min()

```
for ($i = 0$; $i <$ size; $i$++) do
    if content[$i$] = 1 then return $i$;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 22 array.max()

```
for ($i$ = size $-1$; $i \geq 0$; $i$--) do
    if content[$i$] $= 1$ then return $i$;
return null;
```

Algorithm 23 array.min()

```
for ($i = 0$; $i <$ size; $i$++) do
    if content[$i$] $= 1$ then return $i$;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

```
Algorithm 22 array.max()
for ($i$ = size − 1; $i \geq 0$; $i$−−) do
    if content[$i$] = 1 then return $i$;
return null;
```

```
Algorithm 23 array.min()
for ($i = 0$; $i$ < size; $i$++) do
    if content[$i$] = 1 then return $i$;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

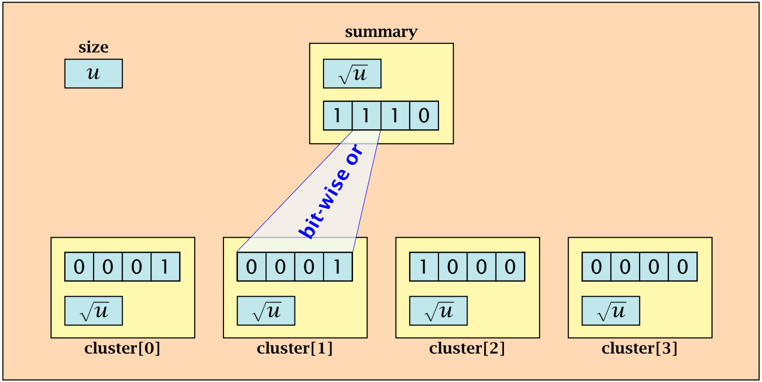
Implementation 1: Array

```
Algorithm 24 array.succ($x$)
for ($i = x + 1$; $i <$ size; $i$++) do
    if content[$i$] $= 1$ then return $i$;
return null;
```

```
Algorithm 25 array.pred($x$)
for ($i = x - 1$; $i \geq 0$; $i$--) do
    if content[$i$] $= 1$ then return $i$;
return null;
```

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array



- $\sqrt{u}$ cluster-arrays of $\sqrt{u}$ bits.
- One summary-array of $\sqrt{u}$ bits. The i-th bit in the summary array stores the bit-wise or of the bits in the i-th cluster.

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\mathrm{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\mathrm{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

Algorithm 26 member(x)

```
return cluster[high(x)].member(low(x));
```

Algorithm 27 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 26 member(x)

```
return cluster[high($x$)].member(low($x$));
```

Algorithm 27 insert(x)

```
cluster[high($x$)].insert(low($x$));
summary.insert(high($x$));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

```
Algorithm 26 member($x$)
return cluster[high($x$)].member(low($x$));
```

```
Algorithm 27 insert($x$)
cluster[high($x$)].insert(low($x$));
summary.insert(high($x$));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

```
Algorithm 28 delete($x$)
cluster[high($x$)].delete(low($x$));
if cluster[high($x$)].min() = null then
    summary.delete(high($x$));
```

- The running time is dominated by the cost of a minimum computation, which will turn out to be $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 28 delete(x)

```
cluster[high($x$)].delete(low($x$));
if cluster[high($x$)].min() = null then
    summary.delete(high($x$));
```

- The running time is dominated by the cost of a minimum computation, which will turn out to be $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 29 max()

```
$maxcluster \leftarrow$ summary.max();
if $maxcluster$ = null return null;
$offs \leftarrow$ cluster[$maxcluster$].max()
return $maxcluster \circ offs$;
```

Algorithm 30 min()

```
$mincluster \leftarrow$ summary.min();
if $mincluster$ = null return null;
$offs \leftarrow$ cluster[$mincluster$].min();
return $mincluster \circ offs$;
```

- Running time is roughly $2\sqrt{u} = \mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array

Algorithm 29 max()

```
$maxcluster \leftarrow$ summary.max();
if $maxcluster$ = null return null;
$offs \leftarrow$ cluster[$maxcluster$].max()
return $maxcluster \circ offs$;
```

Algorithm 30 min()

```
$mincluster \leftarrow$ summary.min();
if $mincluster$ = null return null;
$offs \leftarrow$ cluster[$mincluster$].min();
return $mincluster \circ offs$;
```

- Running time is roughly $2\sqrt{u} = \mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array

Algorithm 29 max()

```
$maxcluster \leftarrow$ summary.max();
if $maxcluster$ = null return null;
$offs \leftarrow$ cluster[$maxcluster$].max()
return $maxcluster \circ offs$;
```

Algorithm 30 min()

```
$mincluster \leftarrow$ summary.min();
if $mincluster$ = null return null;
$offs \leftarrow$ cluster[$mincluster$].min();
return $mincluster \circ offs$;
```

- Running time is roughly $2\sqrt{u} = \mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array

```
Algorithm 31 succ($x$)
$m \leftarrow \text{cluster}[\text{high(x)}].\text{succ}(\text{low}(x))$
if $m \neq \text{null}$ then return $\text{high}(x) \circ m$;
$\mathit{succcluster} \leftarrow \text{summary}.\text{succ}(\text{high}(x))$;
if $\mathit{succcluster} \neq \text{null}$ then
    $\mathit{offs} \leftarrow \text{cluster}[\mathit{succcluster}].\text{min}()$;
    return $\mathit{succcluster} \circ \mathit{offs}$;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 31 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

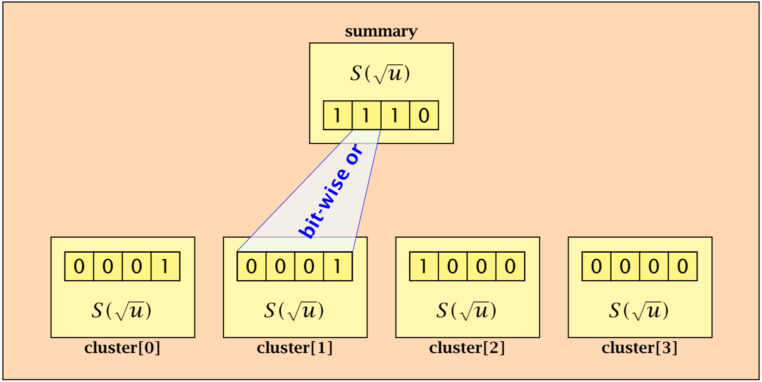
```
Algorithm 32 pred($x$)
$m \leftarrow$ cluster[high(x)].pred(low($x$))
if $m \neq$ null then return high($x$) $\circ$ $m$;
$predcluster \leftarrow$ summary.pred(high($x$));
if $predcluster \neq$ null then
    $offs \leftarrow$ cluster[$predcluster$].max();
    return $predcluster \circ offs$;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 3: Recursion

Instead of using sub-arrays, we build a recursive data-structure.

$S(u)$ is a dynamic set data-structure representing u bits:



Implementation 3: Recursion

We assume that $u = 2^{2^k}$ for some k.

The data-structure $S(2)$ is defined as an array of 2-bits (end of the recursion).

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\text{min}()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\text{min}()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

Algorithm 33 member(x)

```
return cluster[high($x$)].member(low($x$));
```

- $T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 34 insert(x)

```
cluster[high($x$)].insert(low($x$));
summary.insert(high($x$));
```

- $T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$

Implementation 3: Recursion

Algorithm 35 delete(x)

```
cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null then
        summary.delete(high(x));
```

- $T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1.$

Implementation 3: Recursion

Algorithm 36 min()

```
$mincluster \leftarrow \text{summary.min}()$;
if $mincluster = \text{null}$ return null;
$offs \leftarrow \text{cluster}[mincluster].\text{min}()$;
return $mincluster \circ offs$;
```

- $T_{\min}(u) = 2T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 37 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
return null;
```

- $T_{\text{succ}}(u) = 2T_{\text{succ}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1.$

Implementation 3: Recursion

$$T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1 \text{:}$$

Implementation 3: Recursion

$\boldsymbol{T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{mem}}(2^\ell)$.

Implementation 3: Recursion

$T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^{\ell})$.Then

Implementation 3: Recursion

$\boldsymbol{T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{mem}}(2^{\ell})$.Then

$$X(\ell)$$

Implementation 3: Recursion

$T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1$:

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{mem}}(2^\ell)$.Then

$$X(\ell) = T_{\mathrm{mem}}(2^\ell)$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$X(\ell) = T_{\text{mem}}(2^\ell) = T_{\text{mem}}(u)$$

Implementation 3: Recursion

$\boldsymbol{T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{mem}}(2^\ell)$.Then

$$X(\ell) = T_{\mathrm{mem}}(2^\ell) = T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) &= T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 \end{aligned}$$

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) &= T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \enspace . \end{aligned}$$

Implementation 3: Recursion

$T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$:

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) &= T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \enspace . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\log \ell)$, and hence $T_{\text{mem}}(u) = \mathcal{O}(\log\log u)$.

Implementation 3: Recursion

$$T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{ins}}(u) = 2T_{\mathrm{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{ins}}(2^{\ell})$.

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{ins}}(u) = 2T_{\mathrm{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{ins}}(2^\ell)$. Then

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^{\ell})$. Then

$$X(\ell)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 \end{aligned}$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^{\ell})$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^{\ell}) &= T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^{\ell})$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^{\ell}) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

Implementation 3: Recursion

$$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1.}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) = T_{\text{ins}}(u) &= 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

The same holds for $T_{\max}(u)$ and $T_{\min}(u)$.

Implementation 3: Recursion

$$T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$.

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u))}.$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\mathrm{min}}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$X(\ell)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$X(\ell) = T_{\mathrm{del}}(2^\ell)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\text{del}}(\sqrt{u}) + \Theta(\log(u))}.$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$X(\ell) = T_{\text{del}}(2^\ell) = T_{\text{del}}(u)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u))}.$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$X(\ell) = T_{\mathrm{del}}(2^\ell) = T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log u)$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u))}.$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\mathrm{del}}(2^\ell) = T_{\mathrm{del}}(u) &= 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log u) \\ &= 2T_{\mathrm{del}}(2^{\frac{\ell}{2}}) + \Theta(\ell) \end{aligned}$$

Implementation 3: Recursion

$$\boldsymbol{T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\text{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{del}}(2^\ell) &= T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + \Theta(\log u) \\ &= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + \Theta(\ell) = 2X(\tfrac{\ell}{2}) + \Theta(\ell) \ . \end{aligned}$$

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\mathrm{del}}(2^\ell) = T_{\mathrm{del}}(u) &= 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log u) \\ &= 2T_{\mathrm{del}}(2^{\frac{\ell}{2}}) + \Theta(\ell) = 2X(\tfrac{\ell}{2}) + \Theta(\ell) \enspace . \end{aligned}$$

Using Master theorem gives $X(\ell) = \Theta(\ell \log \ell)$, and hence $T_{\mathrm{del}}(u) = \mathcal{O}(\log u \log\log u)$.

Implementation 3: Recursion

$$\boldsymbol{T_{\mathrm{del}}(u) = 2T_{\mathrm{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1 = 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log(u)).}$$

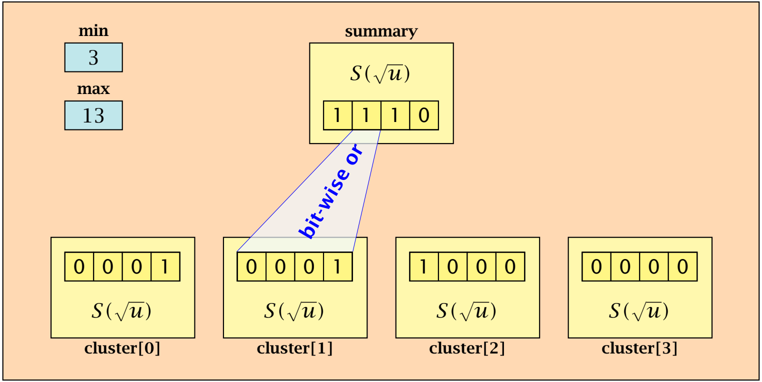
Set $\ell := \log u$ and $X(\ell) := T_{\mathrm{del}}(2^{\ell})$. Then

$$\begin{aligned} X(\ell) = T_{\mathrm{del}}(2^{\ell}) = T_{\mathrm{del}}(u) &= 2T_{\mathrm{del}}(\sqrt{u}) + \Theta(\log u) \\ &= 2T_{\mathrm{del}}(2^{\frac{\ell}{2}}) + \Theta(\ell) = 2X(\tfrac{\ell}{2}) + \Theta(\ell) \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \Theta(\ell \log \ell)$, and hence $T_{\mathrm{del}}(u) = \mathcal{O}(\log u \log\log u)$.

The same holds for $T_{\mathrm{pred}}(u)$ and $T_{\mathrm{succ}}(u)$.

Implementation 4: van Emde Boas Trees



- The bit referenced by min is not set within sub-datastructures.
- The bit referenced by max is set within sub-datastructures (if $\max \neq \min$).

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for min and max are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

```
Algorithm 38 max()
    return max;
```

```
Algorithm 39 min()
    return min;
```

- Constant time.

Implementation 4: van Emde Boas Trees

```
Algorithm 40 member($x$)
if $x = \min$ then return 1; // TRUE
return $\mathrm{cluster}[\mathrm{high}(x)].\mathrm{member}(\mathrm{low}(x))$;
```

- $T_{\mathrm{mem}}(u) = T_{\mathrm{mem}}(\sqrt{u}) + 1 \Longrightarrow T(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 41 succ(x)

```
if $\min \neq \text{null} \wedge x < \min$ then return $\min$;
$maxincluster \leftarrow \text{cluster}[\text{high}(x)].\max()$;
if $maxincluster \neq \text{null} \wedge \text{low}(x) < maxincluster$ then
    $offs \leftarrow \text{cluster}[\text{high}(x)].\text{succ}(\text{low}(x))$;
    return $\text{high}(x) \circ offs$;
else
    $succcluster \leftarrow \text{summary}.\text{succ}(\text{high}(x))$;
    if $succcluster = \text{null}$ then return null;
    $offs \leftarrow \text{cluster}[succcluster].\min()$;
    return $succcluster \circ offs$;
```

- $T_{\text{succ}}(u) = T_{\text{succ}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{succ}}(u) = \mathcal{O}(\log \log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 42 insert(x)

```
if min = null then
    min = $x$; max = $x$;
else
    if $x$ < min then exchange $x$ and min;
    if cluster[high($x$)].min = null; then
        summary.insert(high($x$));
        cluster[high($x$)].insert(low($x$));
    else
        cluster[high($x$)].insert(low($x$));
    if $x$ > max then max = $x$;
```

- $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{ins}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Note that the recusive call in Line 7 takes constant time as the if-condition in Line 5 ensures that we are inserting in an empty sub-tree.

The only non-constant recursive calls are the call in Line 6 and in Line 9. These are mutually exclusive, i.e., only one of these calls will actually occur.

From this we get that $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 43 delete(x)

```
if min = max then
    min = null; max = null;
else
    if $x$ = min then
        $firstcluster \leftarrow$ summary.min();
        $offs \leftarrow$ cluster[$firstcluster$].min();
        $x \leftarrow firstcluster \circ offs$;
        min $\leftarrow x$;
    cluster[high($x$)].delete(low($x$));
```

continued...

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 43 delete(x)

```
if $\text{min} = \text{max}$ then
    $\text{min} = \text{null}; \text{max} = \text{null};$
else
    if $x = \text{min}$ then                                  find new minimum
        $\textit{firstcluster} \leftarrow \text{summary.min}();$
        $\textit{offs} \leftarrow \text{cluster}[\textit{firstcluster}].\text{min}();$
        $x \leftarrow \textit{firstcluster} \circ \textit{offs};$
        $\text{min} \leftarrow x;$
    $\text{cluster}[\text{high}(x)].\text{delete}(\text{low}(x));$
```

continued...

Implementation 4: van Emde Boas Trees

- Assumes that x is contained in the structure.

Algorithm 43 delete(x)

```
if $\min = \max$ then
    $\min = \text{null}; \max = \text{null};$
else
    if $x = \min$ then
        $\mathit{firstcluster} \leftarrow \text{summary.min}();$
        $\mathit{offs} \leftarrow \text{cluster}[\mathit{firstcluster}].\text{min}();$
        $x \leftarrow \mathit{firstcluster} \circ \mathit{offs};$
        $\min \leftarrow x;$
    $\text{cluster}[\text{high}(x)].\text{delete}(\text{low}(x));$        delete
continued...
```

Implementation 4: van Emde Boas Trees

Algorithm 43 delete(x)

...continued

```
if cluster[high($x$)].min() = null then
    summary.delete(high($x$));
    if $x$ = max then
        $summax$ ← summary.max();
        if $summax$ = null then max ← min;
        else
            $offs$ ← cluster[$summax$].max();
            max ← $summax \circ offs$
else
    if $x$ = max then
        $offs$ ← cluster[high($x$)].max();
        max ← high($x$) $\circ$ $offs$;
```

Implementation 4: van Emde Boas Trees

Algorithm 43 delete(x)

...continued

```
if $\text{cluster}[\text{high}(x)].\text{min}() = \text{null}$ then
    $\text{summary}.\text{delete}(\text{high}(x))$;
    if $x = \text{max}$ then
        $summax \leftarrow \text{summary}.\text{max}()$;
        if $summax = \text{null}$ then $\text{max} \leftarrow \text{min}$;
        else
            $offs \leftarrow \text{cluster}[summax].\text{max}()$;
            $\text{max} \leftarrow summax \circ offs$
else
    if $x = \text{max}$ then
        $offs \leftarrow \text{cluster}[\text{high}(x)].\text{max}()$;
        $\text{max} \leftarrow \text{high}(x) \circ offs$;
```

Implementation 4: van Emde Boas Trees

Algorithm 43 delete(x)

...continued

```
if cluster[high($x$)].min() = null then
    summary.delete(high($x$));
    if $x$ = max then
        $summax$ ← summary.max();
        if $summax$ = null then max ← min;
        else
            $offs$ ← cluster[$summax$].max();
            max ← $summax \circ offs$
else
    if $x$ = max then
        $offs$ ← cluster[high($x$)].max();
        max ← high($x$) $\circ$ $offs$;
```

Implementation 4: van Emde Boas Trees

Algorithm 43 delete(x)

...continued

fix maximum

```
    if cluster[high($x$)].min() = null then
        summary.delete(high($x$));
        if $x$ = max then
            $summax \leftarrow$ summary.max();
            if $summax$ = null then max $\leftarrow$ min;
            else
                $offs \leftarrow$ cluster[$summax$].max();
                max $\leftarrow summax \circ offs$
    else
        if $x$ = max then
            $offs \leftarrow$ cluster[high($x$)].max();
            max $\leftarrow$ high($x$) $\circ\ offs$;
```

Implementation 4: van Emde Boas Trees

Note that only one of the possible recusive calls in Line 9 and Line 11 in the deletion-algorithm may take non-constant time.

To see this observe that the call in Line 11 only occurs if the cluster where x was deleted is now empty. But this means that the call in Line 9 deleted the last element in $\text{cluster}[\text{high}(x)]$. Such a call only takes constant time.

Hence, we get a recurrence of the form

$$T_{\text{del}}(u) = T_{\text{del}}(\sqrt{u}) + c \enspace .$$

This gives $T_{\text{del}}(u) = \mathcal{O}(\log\log u)$.

9 van Emde Boas Trees

Space requirements:

- The space requirement fulfills the recurrence

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + \mathcal{O}(\sqrt{u}) \ .$$

- Note that we cannot solve this recurrence by the Master theorem as the branching factor is not constant.
- One can show by induction that the space requirement is $S(u) = \mathcal{O}(u)$. Exercise.

10 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- $\mathcal{P}.\mathbf{makeset}(x)$: Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- $\mathcal{P}.\mathbf{find}(x)$: Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- $\mathcal{P}.\mathbf{union}(x, y)$: Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

10 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}$. makeset(x):** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}$. find(x):** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- **$\mathcal{P}$. union(x, y):** Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

10 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}$. makeset(x):** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}$. find(x):** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- **$\mathcal{P}$. union(x, y):** Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

10 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}$. makeset(x):** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}$. find(x):** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- **$\mathcal{P}$. union(x, y):** Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

10 Union Find

Applications:

- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- Kruskals Minimum Spanning Tree Algorithm

10 Union Find

Applications:

- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- Kruskals Minimum Spanning Tree Algorithm

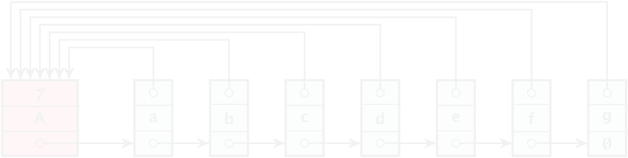
10 Union Find

Algorithm 44 Kruskal-MST($G = (V, E), w$)

```
A ← ∅;
for all v ∈ V do
    v.set ← P.makeset(v.label)
sort edges in non-decreasing order of weight w
for all (u, v) ∈ E in non-decreasing order do
    if P.find(u.set) ≠ P.find(v.set) then
        A ← A ∪ {(u, v)}
        P.union(u.set, v.set)
```

List Implementation

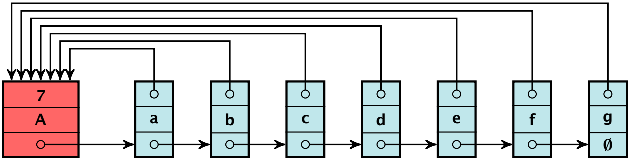
- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- makeset(x) can be performed in constant time.
- find(x) can be performed in constant time.

List Implementation

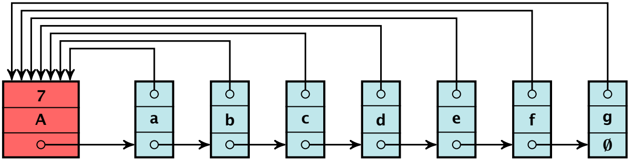
- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- makeset(x) can be performed in constant time.
- find(x) can be performed in constant time.

List Implementation

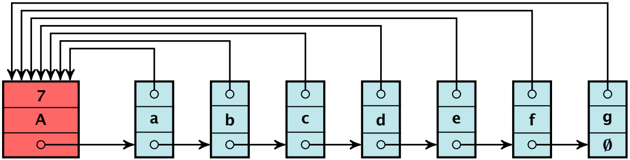
- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- $\text{makeset}(x)$ can be performed in constant time.
- $\text{find}(x)$ can be performed in constant time.

List Implementation

- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



- $\text{makeset}(x)$ can be performed in constant time.
- $\text{find}(x)$ can be performed in constant time.

List Implementation

union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_y.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

List Implementation

union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_y.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

List Implementation

union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_y.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

List Implementation

union(x, y)

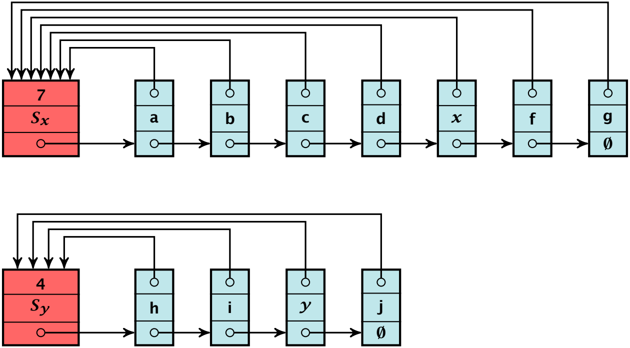
- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_y.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

List Implementation

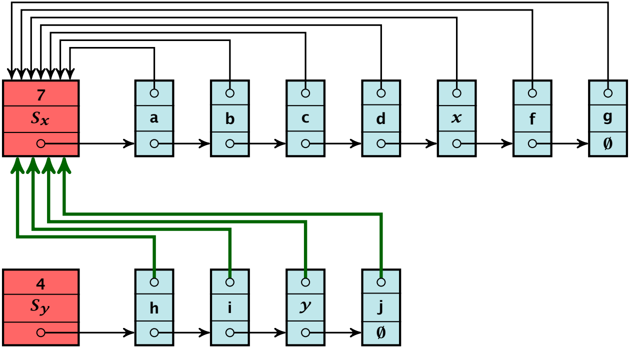
union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_y.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

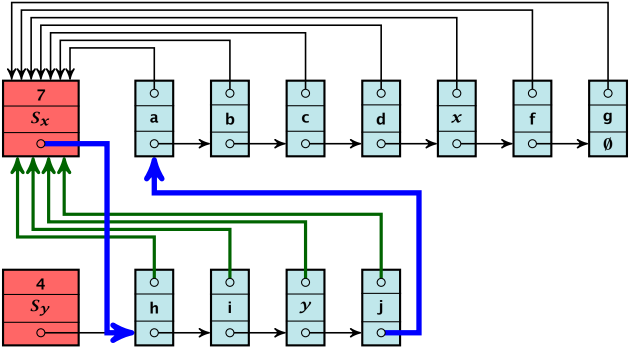
List Implementation



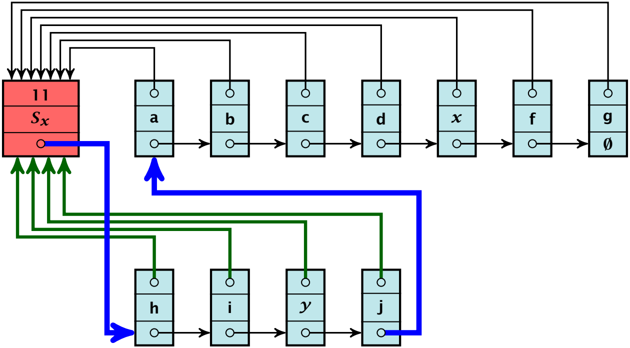
List Implementation



List Implementation



List Implementation



List Implementation

Running times:

- $\text{find}(x)$: constant
- $\text{makeset}(x)$: constant
- $\text{union}(x, y)$: $\mathcal{O}(n)$, where n denotes the number of elements contained in the set system.

List Implementation

Lemma 35

The list implementation for the ADT union find fulfills the following amortized time bounds:

- $\text{find}(x)$: $\mathcal{O}(1)$.
- $\text{makeset}(x)$: $\mathcal{O}(\log n)$.
- $\text{union}(x, y)$: $\mathcal{O}(1)$.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

List Implementation

Lemma 36

An element is charged at most $\lfloor \log_2 n \rfloor$ times, where n is the total number of elements in the set system.

Proof.

Whenever an element x is charged the number of elements in x's set doubles. This can happen at most $\lfloor \log n \rfloor$ times. $\square$

List Implementation

Lemma 36

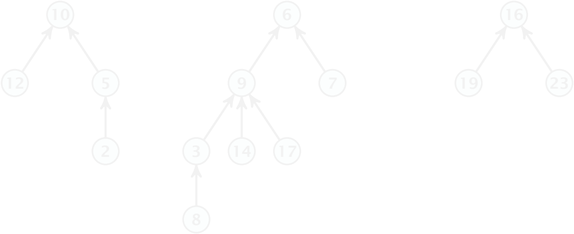
An element is charged at most $\lfloor \log_2 n \rfloor$ *times, where* n *is the total number of elements in the set system.*

Proof.

Whenever an element x is charged the number of elements in x's set doubles. This can happen at most $\lfloor \log n \rfloor$ times. ☐

Implementation via Trees

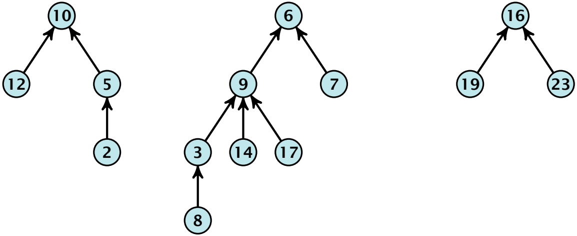
- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.
- Example:



Set system $\{2, 5, 10, 12\}$, $\{3, 6, 7, 8, 9, 14, 17\}$, $\{16, 19, 23\}$.

Implementation via Trees

- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.
- Example:



Set system $\{2, 5, 10, 12\}$, $\{3, 6, 7, 8, 9, 14, 17\}$, $\{16, 19, 23\}$.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\text{level}(x))$, where $\text{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\text{level}(x))$, where $\text{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\mathrm{level}(x))$, where $\mathrm{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\text{level}(x))$, where $\text{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

To support union we store the size of a tree in its root.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

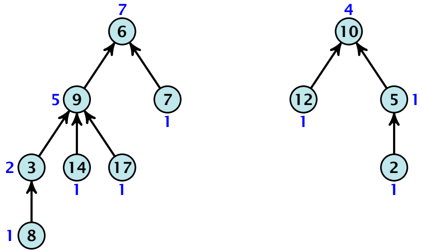
- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \mathrm{find}(x)$; $b \leftarrow \mathrm{find}(y)$. Then: $\mathrm{link}(a, b)$.
- $\mathrm{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

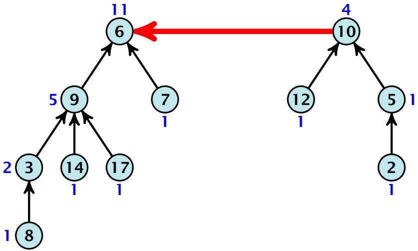


Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.

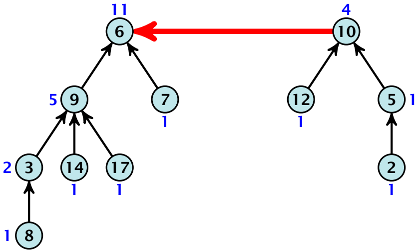


Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.



- Time: constant for $\text{link}(a, b)$ plus two find-operations.

Implementation via Trees

Lemma 37

The running time (non-amortized!!!) for $\mathrm{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\mathrm{size}(c)$ stays fixed, while the value of $\mathrm{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

□

Implementation via Trees

Lemma 37

The running time (non-amortized!!!) for $\operatorname{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\operatorname{size}(p) \geq 2\operatorname{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\operatorname{size}(c)$ stays fixed, while the value of $\operatorname{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\operatorname{size}(p) \geq 2\operatorname{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

□

Implementation via Trees

Lemma 37

The running time (non-amortized!!!) for $\mathrm{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\mathrm{size}(c)$ stays fixed, while the value of $\mathrm{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\mathrm{size}(p) \geq 2\,\mathrm{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

$\square$

Implementation via Trees

Lemma 37

The running time (non-amortized!!!) for $\text{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

- When we attach a tree with root c to become a child of a tree with root p, then $\text{size}(p) \geq 2\,\text{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\text{size}(c)$ stays fixed, while the value of $\text{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\text{size}(p) \geq 2\,\text{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

□

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

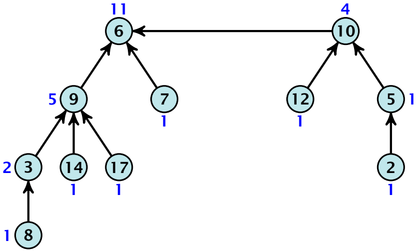
- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

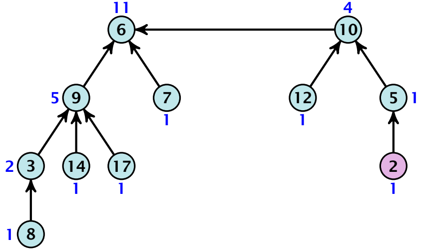


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

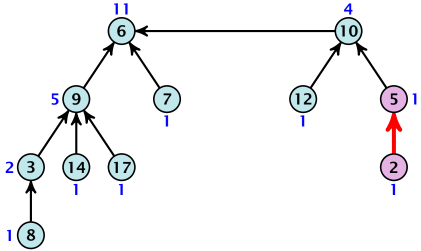


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

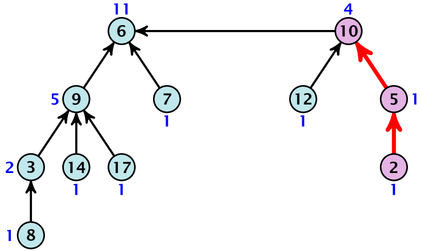


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

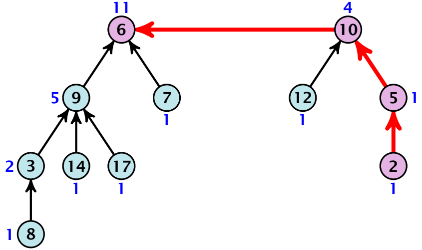


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

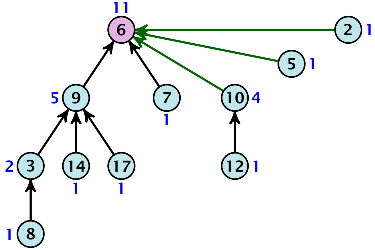


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.

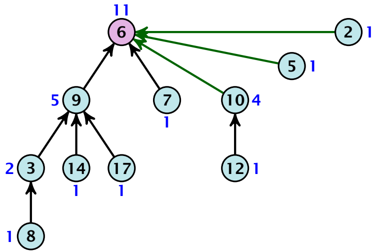


- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time $\mathcal{O}(\log n)$.

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time $\mathcal{O}(\log n)$.

Amortized Analysis

Definitions:

- size(v): the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- rank(v): $\lfloor \log(\text{size}(v)) \rfloor$.
- $\Longrightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$.

Lemma 38

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Definitions:

- $\text{size}(v)$: the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- $\text{rank}(v)$: $\lfloor \log(\text{size}(v)) \rfloor$.
- $\Rightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$.

Lemma 38

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Definitions:

- $\text{size}(v)$: the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- $\text{rank}(v)$: $\lfloor \log(\text{size}(v)) \rfloor$.
- $\Rightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$.

Lemma 38

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Definitions:

- $\text{size}(v)$: the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- $\text{rank}(v)$: $\lfloor \log(\text{size}(v)) \rfloor$.
- $\Longrightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$.

Lemma 38

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Definitions:

- $\mathrm{size}(v)$: the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- $\mathrm{rank}(v)$: $\lfloor\log(\mathrm{size}(v))\rfloor$.
- $\Longrightarrow \mathrm{size}(v) \geq 2^{\mathrm{rank}(v)}$.

Lemma 38

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Lemma 39

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node sees at most one rank s node, but every rank s node is seen by at least 2^s different nodes. ☐

Amortized Analysis

Lemma 39

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. □

Amortized Analysis

Lemma 39

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. ☐

Amortized Analysis

Lemma 39

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. ☐

Amortized Analysis

Lemma 39

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. ☐

Amortized Analysis

We define

$$\operatorname{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\operatorname{tow}(i-1)} & \text{otw.} \end{cases}$$

Amortized Analysis

We define

$$\operatorname{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\operatorname{tow}(i-1)} & \text{otw.} \end{cases}$$

$$\operatorname{tow}(i) = \left. 2^{2^{2^{2^{2}}}} \right\} i \text{ times}$$

Amortized Analysis

We define

$$\operatorname{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\operatorname{tow}(i-1)} & \text{otw.} \end{cases} \qquad \operatorname{tow}(i) = \left.2^{2^{2^{2^{2}}}}\right\} i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \operatorname{tow}(i) \geq n\} \ .$$

Amortized Analysis

We define

$$\operatorname{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\operatorname{tow}(i-1)} & \text{otw.} \end{cases} \qquad \operatorname{tow}(i) = 2^{2^{2^{2^{2}}}} \Big\} i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \operatorname{tow}(i) \geq n\} \ .$$

Theorem 40

Union find with path compression fulfills the following amortized running times:

- $\operatorname{makeset}(x) : \mathcal{O}(\log^*(n))$
- $\operatorname{find}(x) : \mathcal{O}(\log^*(n))$
- $\operatorname{union}(x, y) : \mathcal{O}(\log^*(n))$

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank rank(v) is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1,\ldots,\text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor\log n\rfloor) \leq \log^*(n)-1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\mathrm{rank}(v)$ is in rank group $\log^*(\mathrm{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\mathrm{tow}(g-1)+1, \dots, \mathrm{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1, \ldots, \text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1, \dots, \text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\operatorname{rank}(v)$ is in rank group $\log^*(\operatorname{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\operatorname{tow}(g-1)+1, \dots, \operatorname{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1)+1, \dots, \text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\mathrm{parent}[v]$ as follows:

- If $\mathrm{parent}[v]$ is the root we charge the cost to the find-account.
- If the group number of $\mathrm{rank}(v)$ is the same as that of $\mathrm{rank}(\mathrm{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to parent$[v]$ as follows:

- If parent$[v]$ is the root we charge the cost to the find-account.
- If the group-number of rank(v) is the same as that of rank$(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\mathrm{parent}[v]$ as follows:

- If $\mathrm{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\mathrm{rank}(v)$ is the same as that of $\mathrm{rank}(\mathrm{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\mathrm{parent}[v]$ as follows:

- If $\mathrm{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\mathrm{rank}(v)$ is the same as that of $\mathrm{rank}(\mathrm{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Rightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\text{tow}(g) - \text{tow}(g-1) \le \text{tow}(g)$.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Rightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\mathrm{tow}(g) - \mathrm{tow}(g-1) \le \mathrm{tow}(g)$.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Rightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\text{tow}(g) - \text{tow}(g-1) \leq \text{tow}(g)$.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Rightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\mathrm{tow}(g) - \mathrm{tow}(g-1) \le \mathrm{tow}(g)$.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Longrightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\text{tow}(g) - \text{tow}(g-1) \leq \text{tow}(g)$.

What is the total charge made to nodes?

- The total charge is at most

$$\sum_g n(g) \cdot \mathrm{tow}(g) \quad ,$$

where $n(g)$ is the number of nodes in group g.

What is the total charge made to nodes?

- The total charge is at most

$$\sum_g n(g) \cdot \mathrm{tow}(g) \ ,$$

where $n(g)$ is the number of nodes in group g.

For $g \geq 1$ we have

$$n(g)$$

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s}$$

For $g \geq 1$ we have

$$n(g) \leq \sum_{s=\operatorname{tow}(g-1)+1}^{\operatorname{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\operatorname{tow}(g-1)+1}} \sum_{s=0}^{\operatorname{tow}(g)-\operatorname{tow}(g-1)-1} \frac{1}{2^s}$$

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\
&\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s}
\end{aligned}
$$

For $g \geq 1$ we have

$$
\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\
&\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2
\end{aligned}
$$

For $g \geq 1$ we have

$$\begin{aligned}
n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\
&\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\
&\leq \frac{n}{2^{\mathrm{tow}(g-1)}}
\end{aligned}$$

For $g \geq 1$ we have

$$\begin{aligned} n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \enspace . \end{aligned}$$

For $g \geq 1$ we have

$$\begin{aligned} n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \enspace . \end{aligned}$$

Hence,

$$\sum_g n(g)\,\mathrm{tow}(g)$$

For $g \geq 1$ we have

$$\begin{aligned} n(g) &\leq \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\text{tow}(g)-\text{tow}(g-1)-1} \frac{1}{2^s} \\ &\leq \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\text{tow}(g-1)+1}} \cdot 2 \\ &\leq \frac{n}{2^{\text{tow}(g-1)}} = \frac{n}{\text{tow}(g)} \enspace . \end{aligned}$$

Hence,

$$\sum_g n(g)\,\text{tow}(g) \leq n(0)\,\text{tow}(0) + \sum_{g \geq 1} n(g)\,\text{tow}(g)$$

For $g \geq 1$ we have

$$\begin{aligned} n(g) &\leq \sum_{s=\mathrm{tow}(g-1)+1}^{\mathrm{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\mathrm{tow}(g)-\mathrm{tow}(g-1)-1} \frac{1}{2^s} \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \leq \frac{n}{2^{\mathrm{tow}(g-1)+1}} \cdot 2 \\ &\leq \frac{n}{2^{\mathrm{tow}(g-1)}} = \frac{n}{\mathrm{tow}(g)} \enspace . \end{aligned}$$

Hence,

$$\sum_g n(g)\,\mathrm{tow}(g) \leq n(0)\,\mathrm{tow}(0) + \sum_{g\geq 1} n(g)\,\mathrm{tow}(g) \leq n\log^*(n)$$

Amortized Analysis

Without loss of generality we can assume that all $\mathrm{makeset}$-operations occur at the start.

This means if we inflate the cost of $\mathrm{makeset}$ to $\log^* n$ and add this to the node account of v then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

Amortized Analysis

Without loss of generality we can assume that all $\mathrm{makeset}$-operations occur at the start.

This means if we inflate the cost of $\mathrm{makeset}$ to $\log^* n$ and add this to the node account of v then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

There is also a lower bound of $\Omega(\alpha(m,n))$.

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

There is also a lower bound of $\Omega(\alpha(m,n))$.

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m, n))$, where $\alpha(m, n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

There is also a lower bound of $\Omega(\alpha(m, n))$.

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x,y-1)) & \text{otw.} \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

- $A(0,y) = y+1$
- $A(1,y) = y+2$
- $A(2,y) = 2y+3$
- $A(3,y) = 2^{y+3} - 3$
- $A(4,y) = \underbrace{2^{2^{2^{2}}}}_{y+3 \text{ times}} - 3$

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x,y-1)) & \text{otw.} \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

- $A(0,y) = y + 1$
- $A(1,y) = y + 2$
- $A(2,y) = 2y + 3$
- $A(3,y) = 2^{y+3} - 3$
- $A(4,y) = \underbrace{2^{2^{2^{2}}}}_{y+3 \text{ times}} - 3$

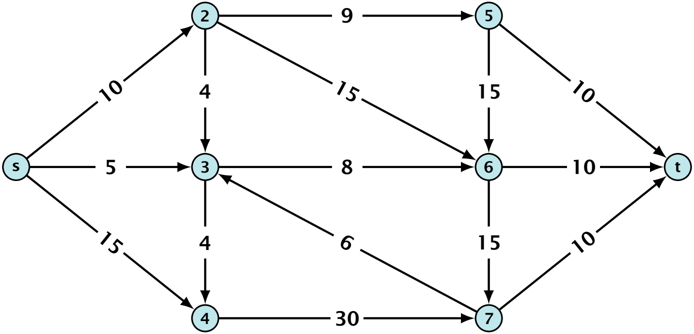
Part IV

Flows and Cuts

11 Introduction

Flow Network

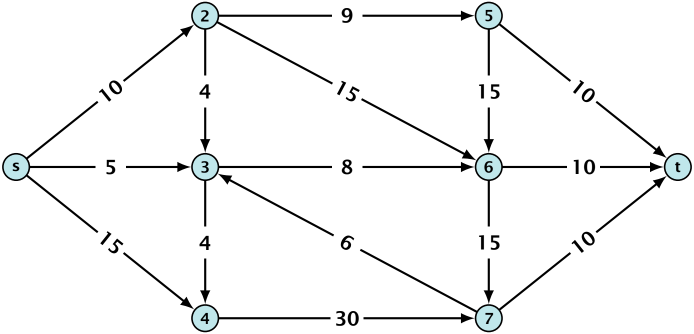
- directed graph $G = (V, E)$; edge capacities $c(e)$
- two special nodes: source s; target t;
- no edges entering s or leaving t;
- at least for now: no parallel edges;



11 Introduction

Flow Network

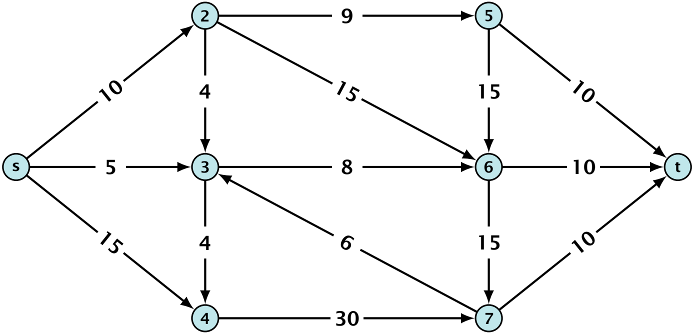
- directed graph $G = (V, E)$; edge capacities $c(e)$
- two special nodes: source s; target t;
- no edges entering s or leaving t;
- at least for now: no parallel edges;



11 Introduction

Flow Network

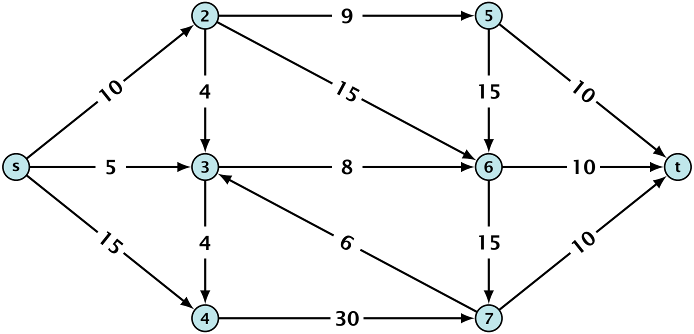
- directed graph $G = (V, E)$; edge capacities $c(e)$
- two special nodes: source s; target t;
- no edges entering s or leaving t;
- at least for now: no parallel edges;



11 Introduction

Flow Network

- directed graph $G = (V, E)$; edge capacities $c(e)$
- two special nodes: source s; target t;
- no edges entering s or leaving t;
- at least for now: no parallel edges;



Cuts

Definition 41

An (s,t)-cut in the graph G is given by a set $A \subset V$ with $s \in A$ and $t \in V \setminus A$.

Cuts

Definition 41

An (s,t)-cut in the graph G is given by a set $A \subset V$ with $s \in A$ and $t \in V \setminus A$.

Definition 42

The capacity of a cut A is defined as

$$\operatorname{cap}(A, V \setminus A) := \sum_{e \in \operatorname{out}(A)} c(e) \ ,$$

where $\operatorname{out}(A)$ denotes the set of edges of the form $A \times V \setminus A$ (i.e. edges leaving A).

Cuts

Definition 41

An (s,t)-cut in the graph G is given by a set $A \subset V$ with $s \in A$ and $t \in V \setminus A$.

Definition 42

The capacity of a cut A is defined as

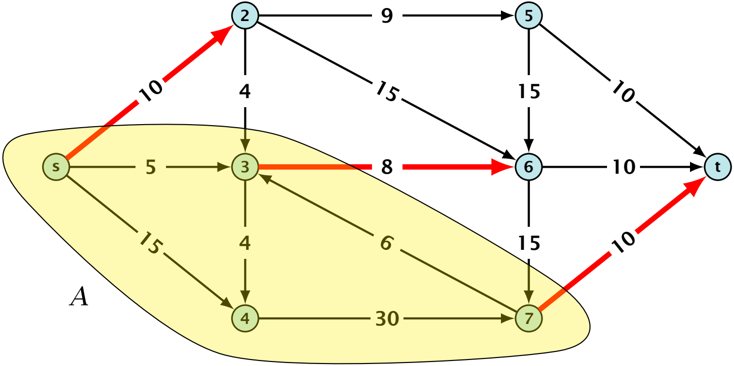
$$\operatorname{cap}(A, V \setminus A) := \sum_{e \in \operatorname{out}(A)} c(e) \ ,$$

where $\operatorname{out}(A)$ denotes the set of edges of the form $A \times V \setminus A$ (i.e. edges leaving A).

Minimum Cut Problem: Find an (s,t)-cut with minimum capacity.

Cuts

Example 43



The capacity of the cut is $\text{cap}(A, V \setminus A) = 28$.

Flows

Definition 44

An (s,t)-flow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e

$$0 \le f(e) \le c(e) \ .$$

   (capacity constraints)

2. For each $v \in V \setminus \{s,t\}$

$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{into}(v)} f(e) \ .$$

   (flow conservation constraints)

Flows

Definition 44

An (s,t)-flow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e

$$0 \leq f(e) \leq c(e) \enspace .$$

(capacity constraints)

2. For each $v \in V \setminus \{s,t\}$

$$\sum_{e \in \text{out}(v)} f(e) = \sum_{e \in \text{into}(v)} f(e) \enspace .$$

(flow conservation constraints)

Flows

Definition 45

The value of an (s, t)-flow f is defined as

$$\mathrm{val}(f) = \sum_{e \in \mathrm{out}(s)} f(e) \ .$$

Maximum Flow Problem: Find an (s, t)-flow with maximum value.

Flows

Definition 45

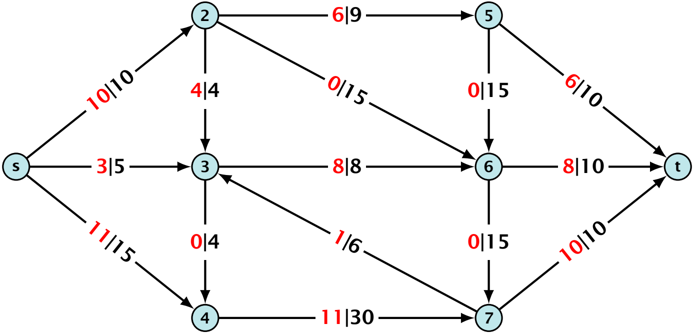
The value of an (s,t)-flow f is defined as

$$\mathrm{val}(f) = \sum_{e \in \mathrm{out}(s)} f(e) \enspace .$$

Maximum Flow Problem: Find an (s,t)-flow with maximum value.

Flows

Example 46



The value of the flow is $\mathrm{val}(f) = 24$.

Lemma 47 (Flow value lemma)

Let f a flow, and let $A \subseteq V$ be an (s,t)-cut. Then the net-flow across the cut is equal to the amount of flow leaving s, i.e.,

$$\operatorname{val}(f) = \sum_{e \in \operatorname{out}(A)} f(e) - \sum_{e \in \operatorname{into}(A)} f(e) \enspace .$$

Proof.

$$\mathrm{val}(f)$$

Proof.

$$\mathrm{val}(f) = \sum_{e \in \mathrm{out}(s)} f(e)$$

Proof.

$$\begin{aligned}\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(s)} f(e)\\ &= \sum_{e\in\mathrm{out}(s)} f(e) + \sum_{v\in A\setminus\{s\}}\left(\sum_{e\in\mathrm{out}(v)} f(e) - \sum_{e\in\mathrm{in}(v)} f(e)\right)\end{aligned}$$

Proof.

$$
\begin{aligned}
\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(s)} f(e) \\
&= \sum_{e\in\mathrm{out}(s)} f(e) + \sum_{v\in A\setminus\{s\}} \overset{=\ \mathbf{0}}{\boxed{\left(\sum_{e\in\mathrm{out}(v)} f(e) - \sum_{e\in\mathrm{in}(v)} f(e)\right)}}
\end{aligned}
$$

Proof.

$$
\begin{aligned}
\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(s)} f(e) \\
&= \sum_{e\in\mathrm{out}(s)} f(e) + \sum_{v\in A\setminus\{s\}} \left(\sum_{e\in\mathrm{out}(v)} f(e) - \sum_{e\in\mathrm{in}(v)} f(e) \right) \\
&= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e)
\end{aligned}
$$

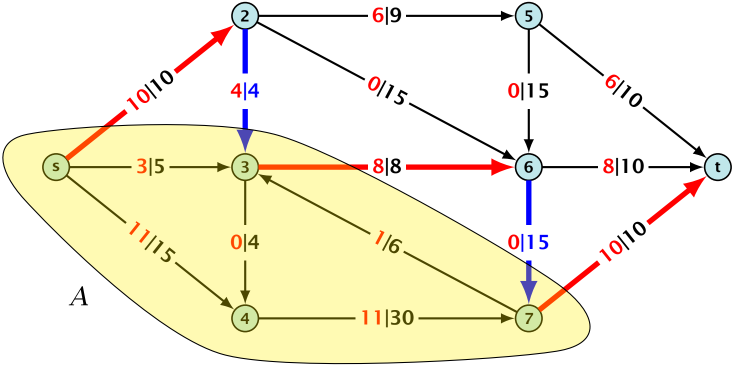
Proof.

$$
\begin{aligned}
\mathrm{val}(f) &= \sum_{e\in \mathrm{out}(s)} f(e) \\
&= \sum_{e\in \mathrm{out}(s)} f(e) + \sum_{v\in A\setminus\{s\}} \Bigg(\sum_{e\in \mathrm{out}(v)} f(e) - \sum_{e\in \mathrm{in}(v)} f(e)\Bigg) \\
&= \sum_{e\in \mathrm{out}(A)} f(e) - \sum_{e\in \mathrm{into}(A)} f(e)
\end{aligned}
$$

The last equality holds since every edge with both end-points in A contributes negatively as well as positively to the sum in line 2. The only edges whose contribution doesn't cancel out are edges leaving or entering A.

□

Example 48


2
5
s
3
6
t
4
7
6|9
10|10
4|4
0|15
0|15
6/10
3|5
8|8
8|10
11/15
0|4
1/6
0|15
10|10
11|30
A

Corollary 49

Let f be an (s, t)-flow and let A be an (s, t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\operatorname{val}(f) = \operatorname{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

□

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

$$\mathrm{cap}(A, V \setminus A) < \mathrm{val}(f')$$

☐

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\operatorname{val}(f) = \operatorname{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

$$\begin{aligned}\operatorname{cap}(A, V \setminus A) &< \operatorname{val}(f') \\ &= \sum_{e \in \operatorname{out}(A)} f'(e) - \sum_{e \in \operatorname{into}(A)} f'(e)\end{aligned}$$

☐

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

$$\begin{aligned}\mathrm{cap}(A, V \setminus A) &< \mathrm{val}(f') \\ &= \sum_{e \in \mathrm{out}(A)} f'(e) - \sum_{e \in \mathrm{into}(A)} f'(e) \\ &\leq \sum_{e \in \mathrm{out}(A)} f'(e)\end{aligned}$$

□

Corollary 49

Let f be an (s,t)-flow and let A be an (s,t)-cut, such that

$$\mathrm{val}(f) = \mathrm{cap}(A, V \setminus A).$$

Then f is a maximum flow.

Proof.

Suppose that there is a flow f' with larger value. Then

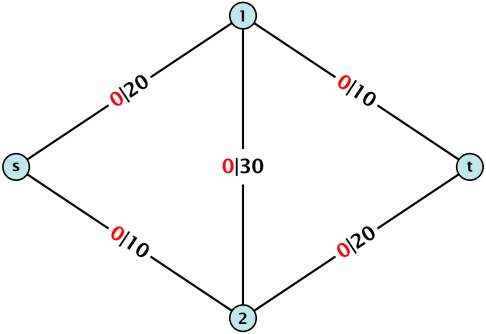
$$\begin{aligned}\mathrm{cap}(A, V \setminus A) &< \mathrm{val}(f') \\ &= \sum_{e\in\mathrm{out}(A)} f'(e) - \sum_{e\in\mathrm{into}(A)} f'(e) \\ &\le \sum_{e\in\mathrm{out}(A)} f'(e) \\ &\le \mathrm{cap}(A, V \setminus A)\end{aligned}$$

□

12 Augmenting Path Algorithms

Greedy-algorithm:

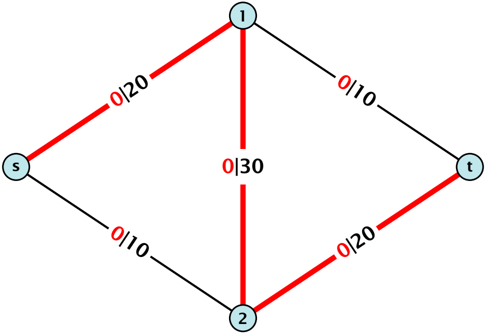
- start with $f(e) = 0$ everywhere
- find an s-t path with $f(e) < c(e)$ on every edge
- augment flow along the path
- repeat as long as possible



12 Augmenting Path Algorithms

Greedy-algorithm:

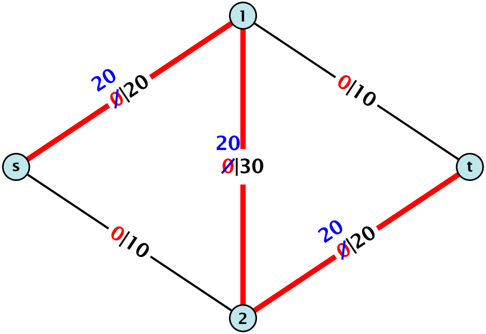
- start with $f(e) = 0$ everywhere
- find an s-t path with $f(e) < c(e)$ on every edge
- augment flow along the path
- repeat as long as possible



12 Augmenting Path Algorithms

Greedy-algorithm:

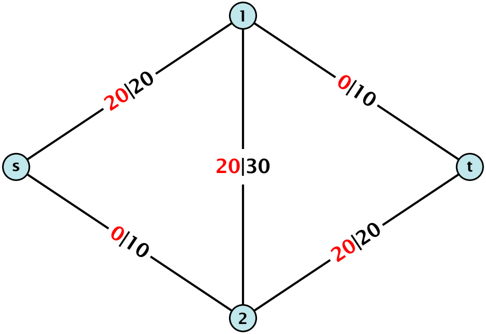
- start with $f(e) = 0$ everywhere
- find an s-t path with $f(e) < c(e)$ on every edge
- augment flow along the path
- repeat as long as possible



12 Augmenting Path Algorithms

Greedy-algorithm:

- start with $f(e) = 0$ everywhere
- find an s-t path with $f(e) < c(e)$ on every edge
- augment flow along the path
- repeat as long as possible



The Residual Graph

From the graph $G = (V, E, c)$ and the current flow f we construct an auxiliary graph $G_f = (V, E_f, c_f)$ (the residual graph):

The Residual Graph

From the graph $G = (V, E, c)$ and the current flow f we construct an auxiliary graph $G_f = (V, E_f, c_f)$ (the residual graph):

- Suppose the original graph has edges $e_1 = (u, v)$, and $e_2 = (v, u)$ between u and v.

The Residual Graph

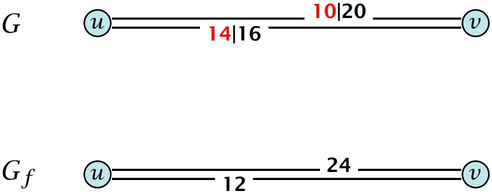
From the graph $G = (V, E, c)$ and the current flow f we construct an auxiliary graph $G_f = (V, E_f, c_f)$ (the residual graph):

- Suppose the original graph has edges $e_1 = (u, v)$, and $e_2 = (v, u)$ between u and v.
- G_f has edge e_1' with capacity $\max\{0, c(e_1) - f(e_1) + f(e_2)\}$ and e_2' with with capacity $\max\{0, c(e_2) - f(e_2) + f(e_1)\}$.

The Residual Graph

From the graph $G = (V, E, c)$ and the current flow f we construct an auxiliary graph $G_f = (V, E_f, c_f)$ (the residual graph):

- Suppose the original graph has edges $e_1 = (u, v)$, and $e_2 = (v, u)$ between u and v.
- G_f has edge e_1' with capacity $\max\{0, c(e_1) - f(e_1) + f(e_2)\}$ and e_2' with with capacity $\max\{0, c(e_2) - f(e_2) + f(e_1)\}$.



Augmenting Path Algorithm

Definition 50

An augmenting path with respect to flow f, is a path in the auxiliary graph G_f that contains only edges with non-zero capacity.

Algorithm 45 FordFulkerson($G = (V, E, c)$)

```
Initialize $f(e) \leftarrow 0$ for all edges.
while $\exists$ augmenting path $p$ in $G_f$ do
    augment as much flow along $p$ as possible.
```

Augmenting Path Algorithm

Definition 50

An augmenting path with respect to flow f, is a path in the auxiliary graph G_f that contains only edges with non-zero capacity.

Algorithm 45 FordFulkerson$(G = (V, E, c))$

```
Initialize $f(e) \leftarrow 0$ for all edges.
while $\exists$ augmenting path $p$ in $G_f$ do
    augment as much flow along $p$ as possible.
```

Augmenting Path Algorithm

Theorem 51

*A flow f is a maximum flow **iff** there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\text{val}(f) = \text{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

☐

Augmenting Path Algorithm

Theorem 51

A flow f is a maximum flow ***iff*** *there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\text{val}(f) = \text{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

☐

Augmenting Path Algorithm

Theorem 51

*A flow f is a maximum flow **iff** there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\text{val}(f) = \text{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

□

Augmenting Path Algorithm

Theorem 51

A flow f is a maximum flow ***iff*** *there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\text{val}(f) = \text{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

□

Augmenting Path Algorithm

Theorem 51

A flow f is a maximum flow ***iff*** *there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\mathrm{val}(f) = \mathrm{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

□

Augmenting Path Algorithm

Theorem 51

A flow f is a maximum flow ***iff*** *there are no augmenting paths.*

Theorem 52

The value of a maximum flow is equal to the value of a minimum cut.

Proof.

Let f be a flow. The following are equivalent:

1. There exists a cut A, B such that $\mathrm{val}(f) = \mathrm{cap}(A, B)$.
2. Flow f is a maximum flow.
3. There is no augmenting path w.r.t. f.

□

Augmenting Path Algorithm

1. $\Rightarrow$ 2.

This we already showed.

2. $\Rightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Rightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.

This we already showed.

2. $\Longrightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.

This we already showed.

2. $\Longrightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.

This we already showed.

2. $\Longrightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.

This we already showed.

2. $\Longrightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

1. $\Longrightarrow$ 2.

This we already showed.

2. $\Longrightarrow$ 3.

If there were an augmenting path, we could improve the flow. Contradiction.

3. $\Longrightarrow$ 1.

- Let f be a flow with no augmenting paths.
- Let A be the set of vertices reachable from s in the residual graph along non-zero capacity edges.
- Since there is no augmenting path we have $s \in A$ and $t \notin A$.

Augmenting Path Algorithm

$$\mathrm{val}(f)$$

Augmenting Path Algorithm

$$\mathrm{val}(f) = \sum_{e \in \mathrm{out}(A)} f(e) - \sum_{e \in \mathrm{into}(A)} f(e)$$

Augmenting Path Algorithm

$$\begin{aligned}\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e)\\ &= \sum_{e\in\mathrm{out}(A)} c(e)\end{aligned}$$

Augmenting Path Algorithm

$$
\begin{aligned}
\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e) \\
&= \sum_{e\in\mathrm{out}(A)} c(e) \\
&= \mathrm{cap}(A, V \setminus A)
\end{aligned}
$$

Augmenting Path Algorithm

$$\begin{aligned}\mathrm{val}(f) &= \sum_{e\in\mathrm{out}(A)} f(e) - \sum_{e\in\mathrm{into}(A)} f(e)\\ &= \sum_{e\in\mathrm{out}(A)} c(e)\\ &= \mathrm{cap}(A, V\setminus A)\end{aligned}$$

This finishes the proof.

Here the first equality uses the flow value lemma, and the second exploits the fact that the flow along incoming edges must be 0 as the residual graph does not have edges leaving A.

Analysis

Assumption:
All capacities are integers between 1 and C.

Invariant:
Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains integral troughout the algorithm.

Analysis

Assumption:
All capacities are integers between 1 and C.

Invariant:
Every flow value $f(e)$ and every residual capacity $c_f(e)$ remains integral troughout the algorithm.

Lemma 53

The algorithm terminates in at most $\mathrm{val}(f^*) \le nC$ *iterations, where* f^* *denotes the maximum flow. Each iteration can be implemented in time* $\mathcal{O}(m)$*. This gives a total running time of* $\mathcal{O}(nmC)$*.*

Theorem 54

If all capacities are integers, then there exists a maximum flow for which every flow value $f(e)$ *is integral.*

Lemma 53

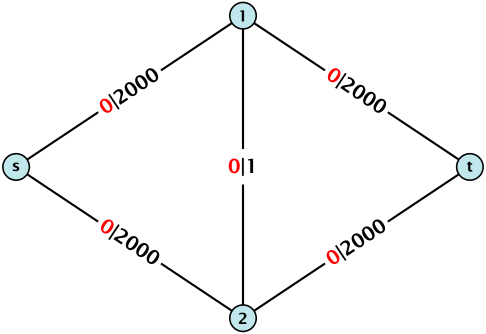
The algorithm terminates in at most $\mathrm{val}(f^*) \le nC$ *iterations, where* f^* *denotes the maximum flow. Each iteration can be implemented in time* $\mathcal{O}(m)$. *This gives a total running time of* $\mathcal{O}(nmC)$.

Theorem 54

If all capacities are integers, then there exists a maximum flow for which every flow value $f(e)$ *is integral.*

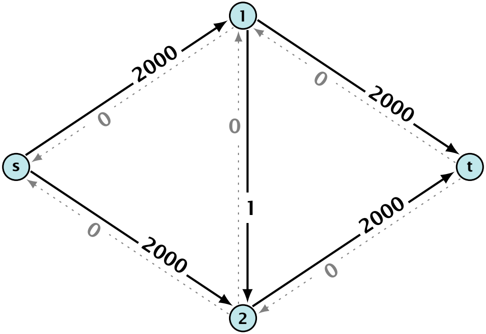
A bad input

Problem: The running time may not be polynomial.



A bad input

Problem: The running time may not be polynomial.

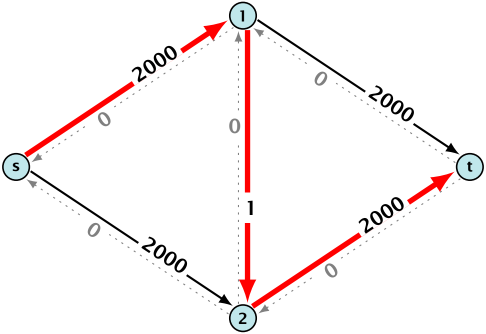


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

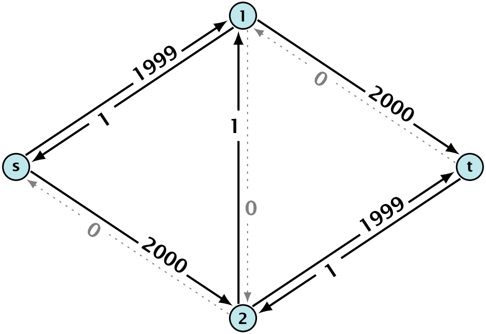


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

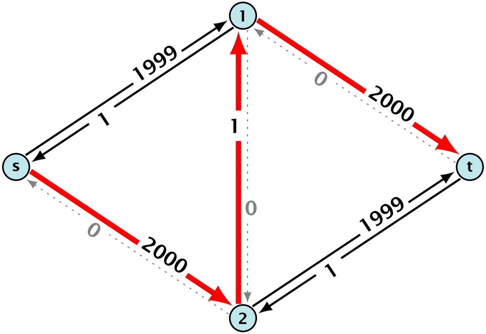


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

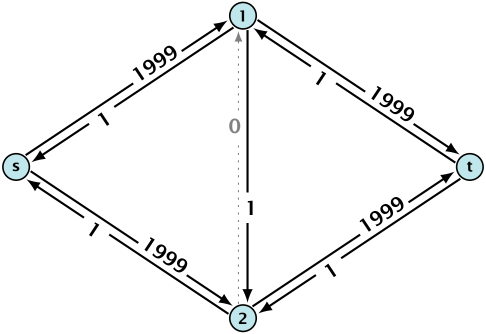


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

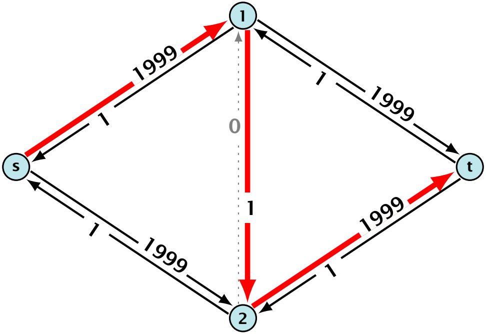


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

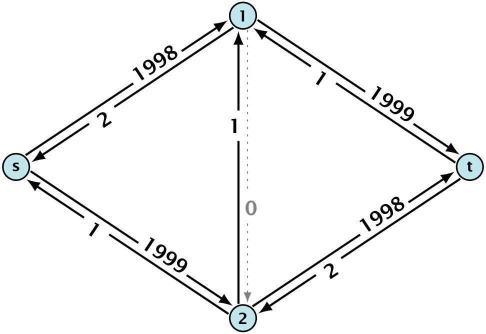


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

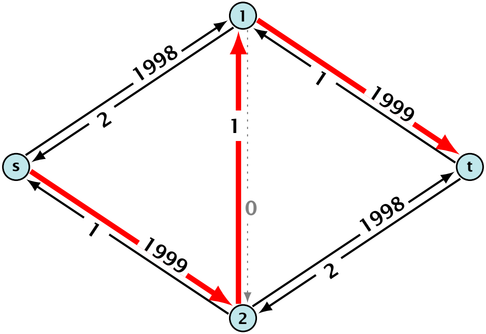


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

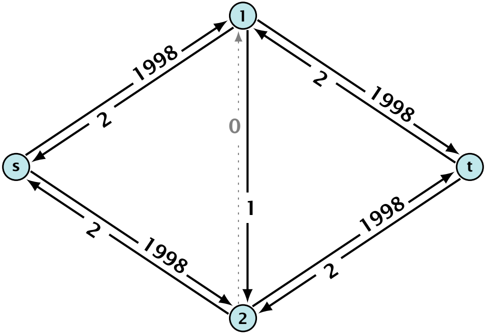


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

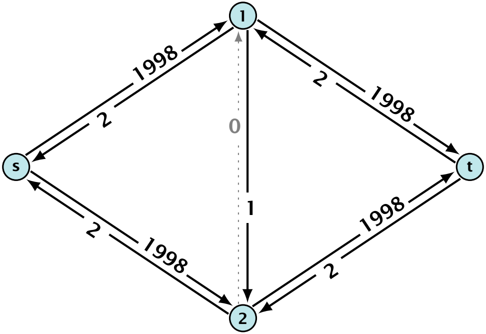


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

A bad input

Problem: The running time may not be polynomial.

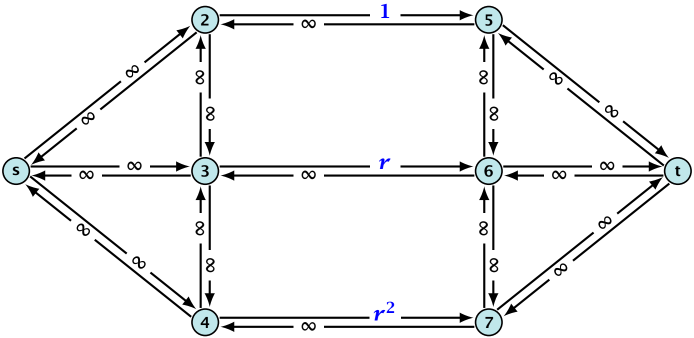


Question:

Can we tweak the algorithm so that the running time is polynomial in the input length?

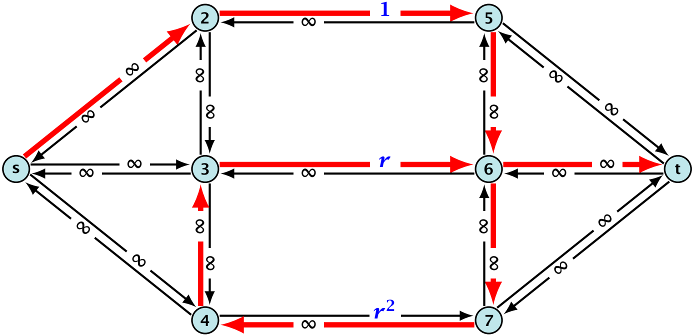
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



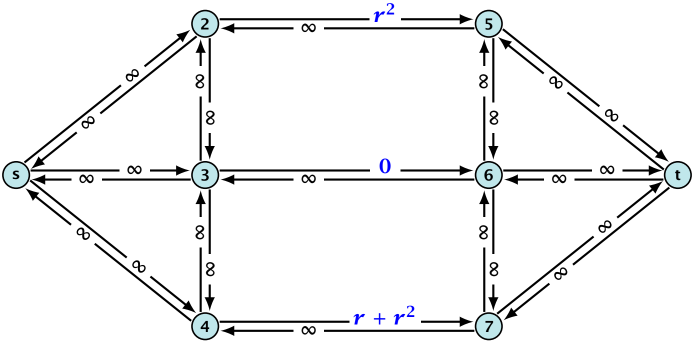
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



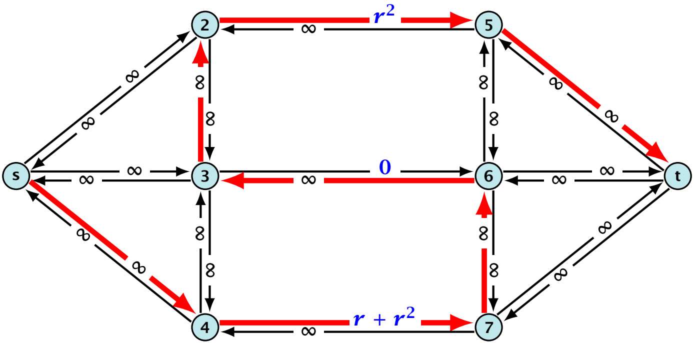
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



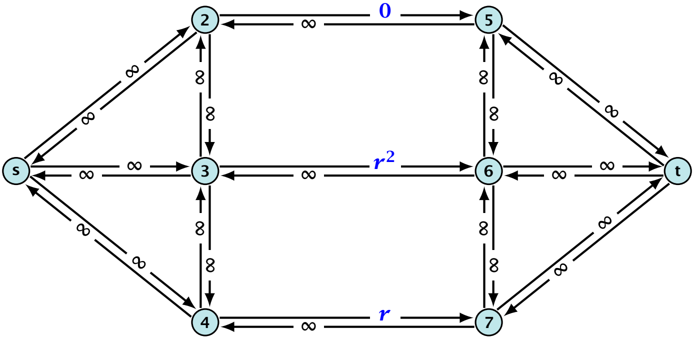
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



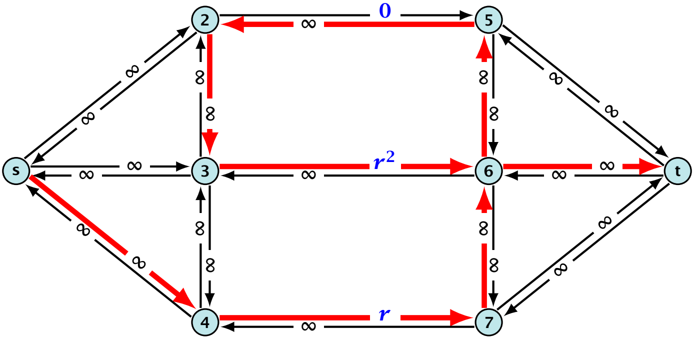
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



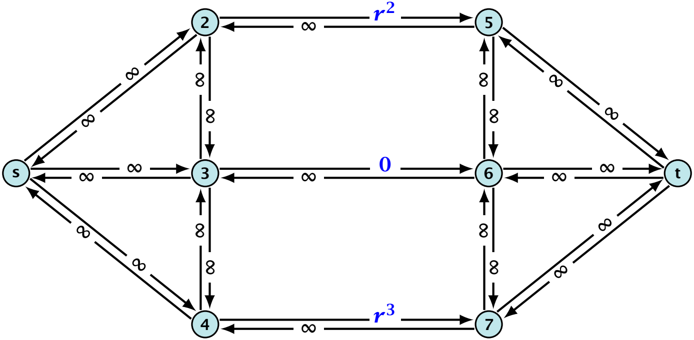
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



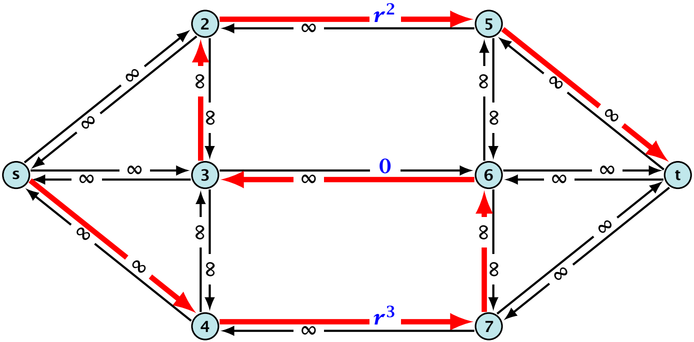
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



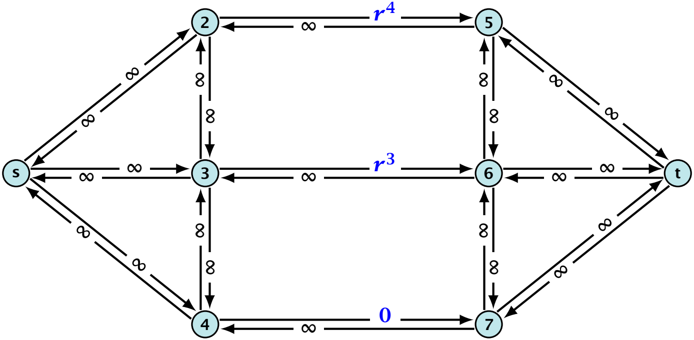
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



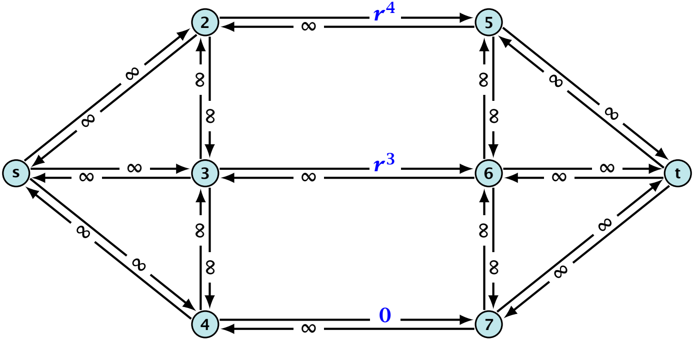
A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



A Pathological Input

Let $r = \frac{1}{2}(\sqrt{5} - 1)$. Then $r^{n+2} = r^n - r^{n+1}$.



Running time may be infinite!!!

How to choose augmenting paths?

How to choose augmenting paths?

- We need to find paths efficiently.

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

- Choose path with maximum bottleneck capacity.

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

- Choose path with maximum bottleneck capacity.
- Choose path with sufficiently large bottleneck capacity.

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

- Choose path with maximum bottleneck capacity.
- Choose path with sufficiently large bottleneck capacity.
- Choose the shortest augmenting path.

=[fill=DarkGreen,draw=DarkGreen]

Overview: Shortest Augmenting Paths

Lemma 55

The length of the shortest augmenting path never decreases.

Lemma 56

After at most $\mathcal{O}(m)$ augmentations, the length of the shortest augmenting path strictly increases.

Overview: Shortest Augmenting Paths

Lemma 55

The length of the shortest augmenting path never decreases.

Lemma 56

After at most $\mathcal{O}(m)$ augmentations, the length of the shortest augmenting path strictly increases.

Overview: Shortest Augmenting Paths

Lemma 55

The length of the shortest augmenting path never decreases.

Lemma 56

After at most $\mathcal{O}(m)$ *augmentations, the length of the shortest augmenting path strictly increases.*

Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

Theorem 57

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. This gives a running time of $\mathcal{O}(m^2n)$.

Proof.

- We can find the shortest augmenting paths in time $\mathcal{O}(m)$ via BFS.
- $\mathcal{O}(m)$ augmentations for paths of exactly $k < n$ edges.

□

Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

Theorem 57

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. This gives a running time of $\mathcal{O}(m^2n)$.

Proof.

- We can find the shortest augmenting paths in time $\mathcal{O}(m)$ via BFS.
- $\mathcal{O}(m)$ augmentations for paths of exactly $k < n$ edges.

□

Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

Theorem 57

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. This gives a running time of $\mathcal{O}(m^2n)$.

Proof.

- We can find the shortest augmenting paths in time $\mathcal{O}(m)$ via BFS.
- $\mathcal{O}(m)$ augmentations for paths of exactly $k < n$ edges.

□

Overview: Shortest Augmenting Paths

These two lemmas give the following theorem:

Theorem 57

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. This gives a running time of $\mathcal{O}(m^2 n)$.

Proof.

- We can find the shortest augmenting paths in time $\mathcal{O}(m)$ via BFS.
- $\mathcal{O}(m)$ augmentations for paths of exactly $k < n$ edges.

☐

Shortest Augmenting Paths

Define the level $\ell(v)$ of a node as the length of the shortest s-v path in G_f.

Shortest Augmenting Paths

Define the level $\ell(v)$ of a node as the length of the shortest s-v path in G_f.

Let L_G denote the subgraph of the residual graph G_f that contains only those edges (u, v) with $\ell(v) = \ell(u) + 1$.

Shortest Augmenting Paths

Define the level $\ell(v)$ of a node as the length of the shortest s-v path in G_f.

Let L_G denote the subgraph of the residual graph G_f that contains only those edges (u,v) with $\ell(v) = \ell(u) + 1$.

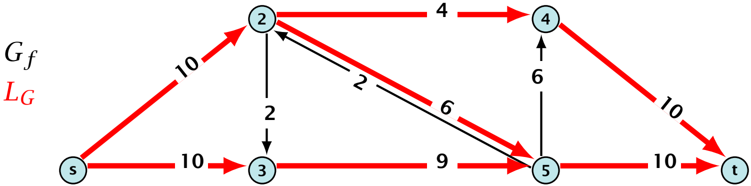
A path P is a shortest s-u path in G_f if it is a an s-u path in L_G.

Shortest Augmenting Paths

Define the level $\ell(v)$ of a node as the length of the shortest s-v path in G_f.

Let L_G denote the subgraph of the residual graph G_f that contains only those edges (u, v) with $\ell(v) = \ell(u) + 1$.

A path P is a shortest s-u path in G_f if it is a an s-u path in L_G.



Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

- After an augmentation the following changes are done in G_f.

Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

- After an augmentation the following changes are done in G_f.
- Some edges of the chosen path may be deleted (bottleneck edges).

Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

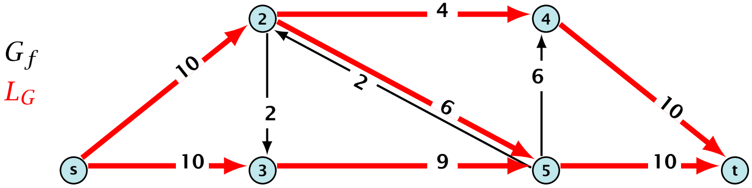
- After an augmentation the following changes are done in G_f.
- Some edges of the chosen path may be deleted (bottleneck edges).
- Back edges are added to all edges that don't have back edges so far.

Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

- After an augmentation the following changes are done in G_f.
- Some edges of the chosen path may be deleted (bottleneck edges).
- Back edges are added to all edges that don't have back edges so far.

These changes cannot decrease the distance between s and t.

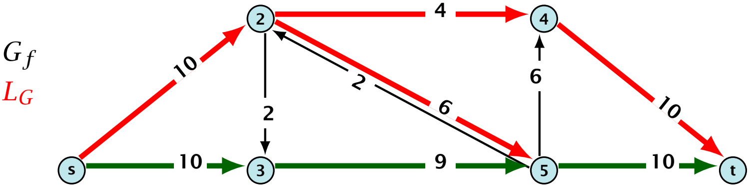


Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

- After an augmentation the following changes are done in G_f.
- Some edges of the chosen path may be deleted (bottleneck edges).
- Back edges are added to all edges that don't have back edges so far.

These changes cannot decrease the distance between s and t.

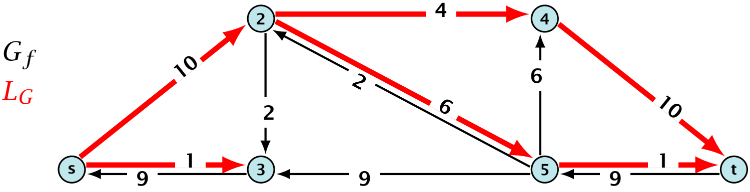


Shortest Augmenting Path

First Lemma: The length of the shortest augmenting path never decreases.

- After an augmentation the following changes are done in G_f.
- Some edges of the chosen path may be deleted (bottleneck edges).
- Back edges are added to all edges that don't have back edges so far.

These changes cannot decrease the distance between s and t.



Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let E_L denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let E_L denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

An s-t path in G_f that does use edges not in E_L has length larger than k, even when considering edges added to G_f during the round.

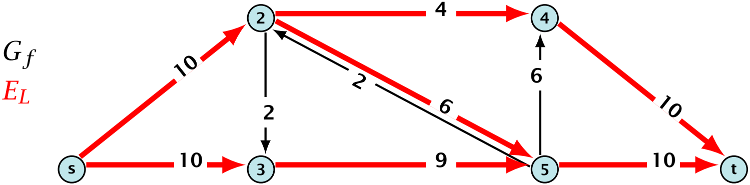
Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let E_L denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

An s-t path in G_f that does use edges not in E_L has length larger than k, even when considering edges added to G_f during the round.

In each augmentation one edge is deleted from E_L.



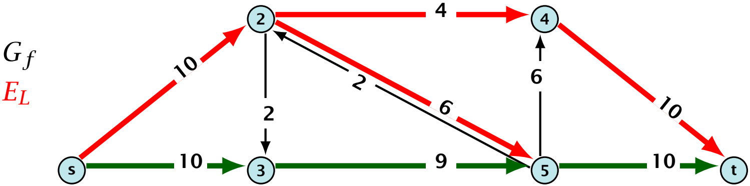
Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let E_L denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

An s-t path in G_f that does use edges not in E_L has length larger than k, even when considering edges added to G_f during the round.

In each augmentation one edge is deleted from E_L.



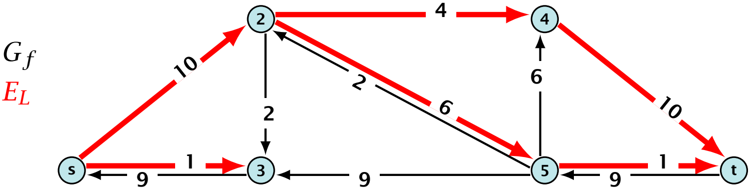
Shortest Augmenting Path

Second Lemma: After at most m augmentations the length of the shortest augmenting path strictly increases.

Let E_L denote the set of edges in graph L_G at the beginning of a round when the distance between s and t is k.

An s-t path in G_f that does use edges not in E_L has length larger than k, even when considering edges added to G_f during the round.

In each augmentation one edge is deleted from E_L.



Shortest Augmenting Paths

Theorem 58

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. Each augmentation can be performed in time $\mathcal{O}(m)$.

Theorem 59 (without proof)

There exist networks with $m = \Theta(n^2)$ that require $\mathcal{O}(mn)$ augmentations, when we restrict ourselves to only augment along shortest augmenting paths.

Note:

There always exists a set of m augmentations that gives a maximum flow.

Shortest Augmenting Paths

Theorem 58

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ *augmentations. Each augmentation can be performed in time* $\mathcal{O}(m)$*.*

Theorem 59 (without proof)

There exist networks with $m = \Theta(n^2)$ *that require* $\mathcal{O}(mn)$ *augmentations, when we restrict ourselves to only augment along shortest augmenting paths.*

Note:

There always exists a set of m augmentations that gives a maximum flow.

Shortest Augmenting Paths

Theorem 58

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. Each augmentation can be performed in time $\mathcal{O}(m)$.

Theorem 59 (without proof)

There exist networks with $m = \Theta(n^2)$ that require $\mathcal{O}(mn)$ augmentations, when we restrict ourselves to only augment along shortest augmenting paths.

Note:

There always exists a set of m augmentations that gives a maximum flow.

Shortest Augmenting Paths

Theorem 58

The shortest augmenting path algorithm performs at most $\mathcal{O}(mn)$ augmentations. Each augmentation can be performed in time $\mathcal{O}(m)$.

Theorem 59 (without proof)

There exist networks with $m = \Theta(n^2)$ that require $\mathcal{O}(mn)$ augmentations, when we restrict ourselves to only augment along shortest augmenting paths.

Note:

There always exists a set of m augmentations that gives a maximum flow.

Shortest Augmenting Paths

When sticking to shortest augmenting paths we cannot improve (asymptotically) on the number of augmentations.

However, we can improve the running time to $\mathcal{O}(mn^2)$ by improving the running time for finding an augmenting path (currently we assume $\mathcal{O}(m)$ per augmentation for this).

Shortest Augmenting Paths

When sticking to shortest augmenting paths we cannot improve (asymptotically) on the number of augmentations.

However, we can improve the running time to $\mathcal{O}(mn^2)$ by improving the running time for finding an augmenting path (currently we assume $\mathcal{O}(m)$ per augmentation for this).

Shortest Augmenting Paths

We maintain a subset E_L of the edges of G_f with the guarantee that a shortest s-t path using only edges from E_L is a shortest augmenting path.

With each augmentation some edges are deleted from E_L.

When E_L does not contain an s-t path anymore the distance between s and t strictly increases.

Note that E_L is not the set of edges of the level graph but a subset of level-graph edges.

Shortest Augmenting Paths

We maintain a subset E_L of the edges of G_f with the guarantee that a shortest s-t path using only edges from E_L is a shortest augmenting path.

With each augmentation some edges are deleted from E_L.

When E_L does not contain an s-t path anymore the distance between s and t strictly increases.

Note that E_L is not the set of edges of the level graph but a subset of level-graph edges.

Shortest Augmenting Paths

We maintain a subset E_L of the edges of G_f with the guarantee that a shortest s-t path using only edges from E_L is a shortest augmenting path.

With each augmentation some edges are deleted from E_L.

When E_L does not contain an s-t path anymore the distance between s and t strictly increases.

Note that E_L is not the set of edges of the level graph but a subset of level-graph edges.

Shortest Augmenting Paths

We maintain a subset E_L of the edges of G_f with the guarantee that a shortest s-t path using only edges from E_L is a shortest augmenting path.

With each augmentation some edges are deleted from E_L.

When E_L does not contain an s-t path anymore the distance between s and t strictly increases.

Note that E_L is not the set of edges of the level graph but a subset of level-graph edges.

Suppose that the initial distance between s and t in G_f is k.

E_L is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from E_L.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from E_L.

Suppose that the initial distance between s and t in G_f is k.

E_L is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from E_L.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from E_L.

Suppose that the initial distance between s and t in G_f is k.

E_L is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from E_L.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from E_L.

Suppose that the initial distance between s and t in G_f is k.

E_L is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from E_L.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from E_L.

Suppose that the initial distance between s and t in G_f is k.

E_L is initialized as the level graph L_G.

Perform a DFS search to find a path from s to t using edges from E_L.

Either you find t after at most n steps, or you end at a node v that does not have any outgoing edges.

You can delete incoming edges of v from E_L.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

Let a phase of the algorithm be defined by the time between two augmentations during which the distance between s and t strictly increases.

Initializing E_L for the phase takes time $\mathcal{O}(m)$.

The total cost for searching for augmenting paths during a phase is at most $\mathcal{O}(mn)$, since every search (successful (i.e., reaching t) or unsuccessful) decreases the number of edges in E_L and takes time $\mathcal{O}(n)$.

The total cost for performing an augmentation during a phase is only $\mathcal{O}(n)$. For every edge in the augmenting path one has to update the residual graph G_f and has to check whether the edge is still in E_L for the next search.

There are at most n phases. Hence, total cost is $\mathcal{O}(mn^2)$.

How to choose augmenting paths?

- We need to find paths efficiently.

Several possibilities:

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

How to choose augmenting paths?

- We need to find paths efficiently.
- We want to guarantee a small number of iterations.

Several possibilities:

- Choose path with maximum bottleneck capacity.
- Choose path with sufficiently large bottleneck capacity.
- Choose the shortest augmenting path.

Capacity Scaling

Capacity Scaling

Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.

Capacity Scaling

Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- Don’t worry about finding the exact bottleneck.

Capacity Scaling

Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- Don't worry about finding the exact bottleneck.
- Maintain scaling parameter Δ.

Capacity Scaling

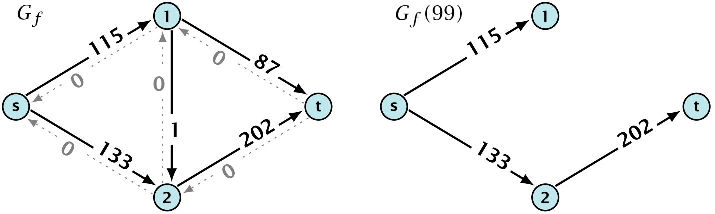
Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- Don't worry about finding the exact bottleneck.
- Maintain scaling parameter Δ.
- $G_f(\Delta)$ is a sub-graph of the residual graph G_f that contains only edges with capacity at least Δ.

Capacity Scaling

Intuition:

- Choosing a path with the highest bottleneck increases the flow as much as possible in a single step.
- Don't worry about finding the exact bottleneck.
- Maintain scaling parameter Δ.
- $G_f(\Delta)$ is a sub-graph of the residual graph G_f that contains only edges with capacity at least Δ.



Capacity Scaling

Algorithm 46 maxflow(G, s, t, c)

```
foreach $e \in E$ do $f_e \leftarrow 0$;
$\Delta \leftarrow 2^{\lceil \log_2 C \rceil}$
while $\Delta \geq 1$ do
    $G_f(\Delta) \leftarrow \Delta$-residual graph
    while there is augmenting path $P$ in $G_f(\Delta)$ do
        $f \leftarrow$ augment$(f, c, P)$
        update$(G_f(\Delta))$
    $\Delta \leftarrow \Delta/2$
return $f$
```

Capacity Scaling

Capacity Scaling

Assumption:

All capacities are integers between 1 and C.

Capacity Scaling

Assumption:
All capacities are integers between 1 and C.

Invariant:
All flows and capacities are/remain integral throughout the algorithm.

Capacity Scaling

Assumption:
All capacities are integers between 1 and C.

Invariant:
All flows and capacities are/remain integral throughout the algorithm.

Correctness:
The algorithm computes a maxflow:

- because of integrality we have $G_f(1) = G_f$

Capacity Scaling

Assumption:

All capacities are integers between 1 and C.

Invariant:

All flows and capacities are/remain integral throughout the algorithm.

Correctness:

The algorithm computes a maxflow:

- because of integrality we have $G_f(1) = G_f$
- therefore after the last phase there are no augmenting paths anymore

Capacity Scaling

Assumption:
All capacities are integers between 1 and C.

Invariant:
All flows and capacities are/remain integral throughout the algorithm.

Correctness:
The algorithm computes a maxflow:

- because of integrality we have $G_f(1) = G_f$
- therefore after the last phase there are no augmenting paths anymore
- this means we have a maximum flow.

Capacity Scaling

Capacity Scaling

Lemma 60

There are $\lceil \log C \rceil$ *iterations over* Δ*.*

Proof: obvious.

Capacity Scaling

Lemma 60

There are $\lceil \log C \rceil$ *iterations over* Δ*.*

Proof: obvious.

Lemma 61

Let f *be the flow at the end of a* Δ*-phase. Then the maximum flow is smaller than* $\mathrm{val}(f) + 2m\Delta$*.*

Proof: less obvious, but simple:

Capacity Scaling

Lemma 60

There are $\lceil \log C \rceil$ *iterations over* Δ.

Proof: obvious.

Lemma 61

Let f *be the flow at the end of a* Δ*-phase. Then the maximum flow is smaller than* $\mathrm{val}(f) + 2m\Delta$.

Proof: less obvious, but simple:

- An s-t cut in $G_f(\Delta)$ gives me an upper bound on the amount of flow that my algorithm can still add to f.

Capacity Scaling

Lemma 60

There are $\lceil \log C \rceil$ *iterations over* Δ.

Proof: obvious.

Lemma 61

Let f *be the flow at the end of a* Δ*-phase. Then the maximum flow is smaller than* $\mathrm{val}(f) + 2m\Delta$.

Proof: less obvious, but simple:

- An s-t cut in $G_f(\Delta)$ gives me an upper bound on the amount of flow that my algorithm can still add to f.
- The edges that currently have capacity at most Δ in G_f form an s-t cut with capacity at most $2m\Delta$.

Capacity Scaling

Capacity Scaling

Lemma 62

There are at most $2m$ augmentations per scaling-phase.

Capacity Scaling

Lemma 62

There are at most $2m$ augmentations per scaling-phase.

Proof:

- Let f be the flow at the end of the previous phase.

Capacity Scaling

Lemma 62

There are at most $2m$ augmentations per scaling-phase.

Proof:

- Let f be the flow at the end of the previous phase.
- $\mathrm{val}(f^*) \le \mathrm{val}(f) + 2m\Delta$

Capacity Scaling

Lemma 62

There are at most $2m$ augmentations per scaling-phase.

Proof:

- Let f be the flow at the end of the previous phase.
- $\text{val}(f^*) \le \text{val}(f) + 2m\Delta$
- each augmentation increases flow by Δ.

Capacity Scaling

Lemma 62

There are at most $2m$ augmentations per scaling-phase.

Proof:

- Let f be the flow at the end of the previous phase.
- $\mathrm{val}(f^*) \le \mathrm{val}(f) + 2m\Delta$
- each augmentation increases flow by Δ.

Theorem 63

We need $\mathcal{O}(m \log C)$ augmentations. The algorithm can be implemented in time $\mathcal{O}(m^2 \log C)$.

Preflows

Definition 64

An (s,t)-preflow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e
$$0 \le f(e) \le c(e) \;.$$
(capacity constraints)
2. For each $v \in V \setminus \{s,t\}$
$$\sum_{e \in \text{out}(v)} f(e) \le \sum_{e \in \text{into}(v)} f(e) \;.$$

Preflows

Definition 64

An (s,t)-preflow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e
$$0 \leq f(e) \leq c(e) \ .$$
(capacity constraints)
2. For each $v \in V \setminus \{s,t\}$
$$\sum_{e \in \text{out}(v)} f(e) \leq \sum_{e \in \text{into}(v)} f(e) \ .$$

Preflows

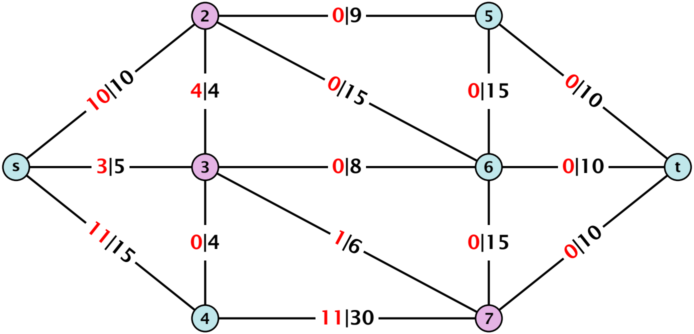
Definition 64

An (s,t)-preflow is a function $f : E \mapsto \mathbb{R}^+$ that satisfies

1. For each edge e
$$0 \leq f(e) \leq c(e) \enspace .$$
(capacity constraints)
2. For each $v \in V \setminus \{s,t\}$
$$\sum_{e \in \text{out}(v)} f(e) \leq \sum_{e \in \text{into}(v)} f(e) \enspace .$$

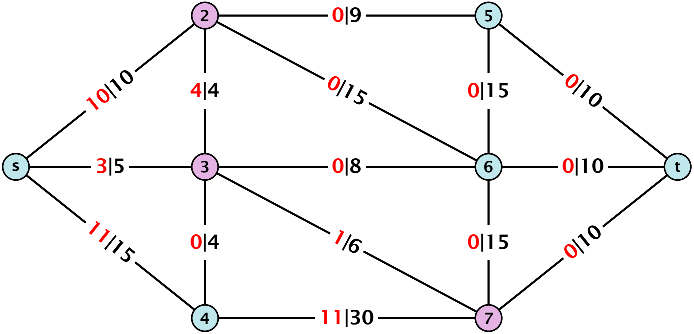
Preflows

Example 65



Preflows

Example 65



A node that has $\sum_{e\in\text{out}(v)} f(e) < \sum_{e\in\text{into}(v)} f(e)$ is called an active node.

Preflows

Preflows

Definition:

A labelling is a function $\ell : V \to \mathbb{N}$. It is valid for preflow f if

- $\ell(u) \le \ell(v) + 1$ for all edges in the residual graph G_f (only non-zero capacity edges!!!)

Preflows

Definition:

A labelling is a function $\ell : V \to \mathbb{N}$. It is valid for preflow f if

- $\ell(u) \leq \ell(v) + 1$ for all edges in the residual graph G_f (only non-zero capacity edges!!!)
- $\ell(s) = n$

Preflows

Definition:

A labelling is a function $\ell : V \to \mathbb{N}$. It is valid for preflow f if

- $\ell(u) \leq \ell(v) + 1$ for all edges in the residual graph G_f (only non-zero capacity edges!!!)
- $\ell(s) = n$
- $\ell(t) = 0$

Preflows

Definition:

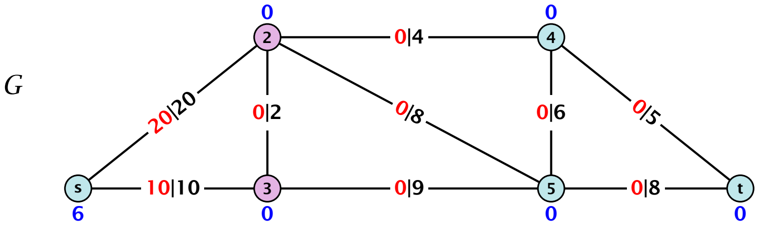
A labelling is a function $\ell : V \rightarrow \mathbb{N}$. It is valid for preflow f if

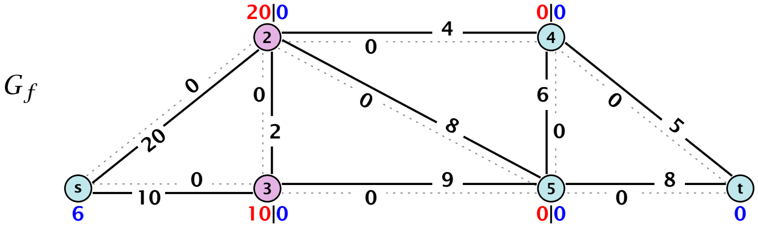
- $\ell(u) \leq \ell(v) + 1$ for all edges in the residual graph G_f (only non-zero capacity edges!!!)
- $\ell(s) = n$
- $\ell(t) = 0$

Intuition:

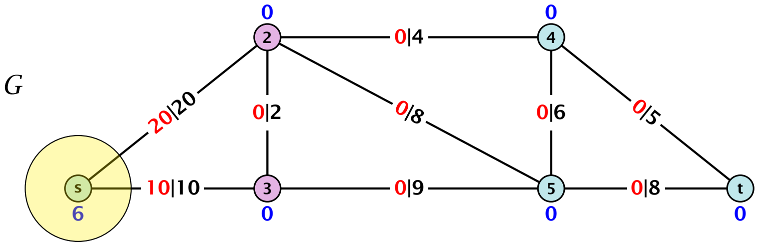
The labelling can be viewed as a height function. Whenever the height from node u to node v decreases by more than 1 (i.e., it goes very steep downhill from u to v), the corresponding edge must be saturated.

Preflows

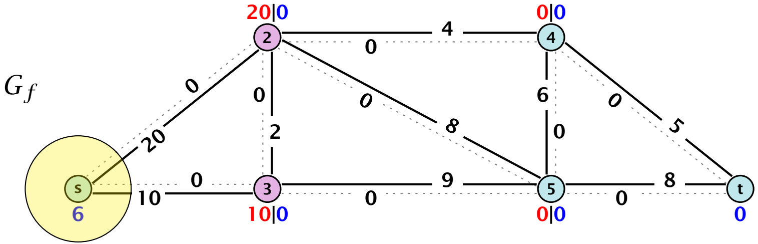

G
0
0
2
4
0|4
20|20
0|2
0|8
0|6
0|5
s
10|10
3
0|9
5
0|8
t
6
0
0
0


G_f
20|0
0|0
2
4
4
0
0
0
0
6
0
5
20
2
8
0
s
0
3
9
5
8
t
10
0
0
6
10|0
0|0
0

Preflows

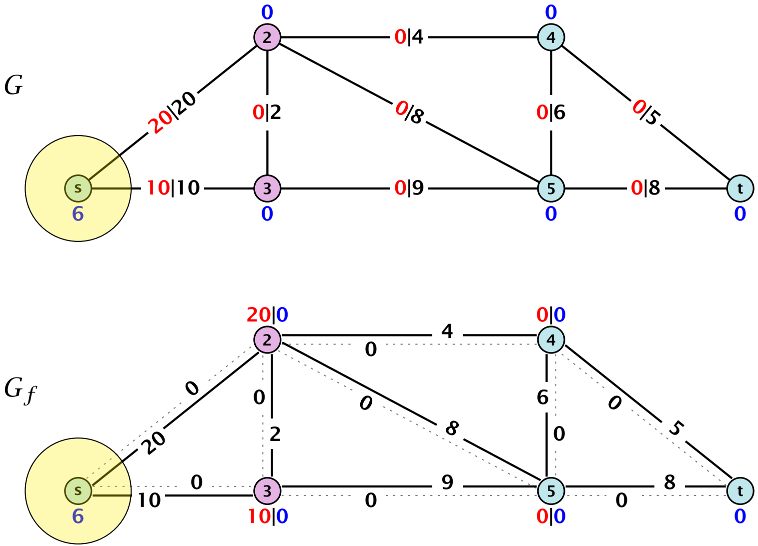


G
0
0
2
0|4
4
20|20
0|2
0|8
0|6
0|5
s
10|10
3
0|9
5
0|8
t
6
0
0
0



G_f
20|0
0|0
2
4
0
4
0
0
0
6
0
8
5
20
2
0
s
0
3
9
5
8
t
10
0
0
6
10|0
0|0
0

Preflows


G
0
0
2
4
0|4
20|20
0|2
0/8
0|6
0/5
s
6
10|10
3
0
0|9
5
0
0|8
t
0
G_f
20|0
0|0
2
4
4
0
0
0
0
0
20
2
8
6
0
5
s
6
0
10
3
9
0
5
8
0
t
10|0
0|0
0

Preflows

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n+1$ different labels from $0, \ldots, n$.

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n+1$ different labels from $0,\dots,n$.
- There must exist a label $d \in \{0,\dots,n\}$ such that none of the nodes carries this label.

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n+1$ different labels from $0, \ldots, n$.
- There must exist a label $d \in \{0, \ldots, n\}$ such that none of the nodes carries this label.
- Let $A = \{v \in V \mid \ell(v) > d\}$ and $B = \{v \in V \mid \ell(v) < d\}$.

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n+1$ different labels from $0, \dots, n$.
- There must exist a label $d \in \{0, \dots, n\}$ such that none of the nodes carries this label.
- Let $A = \{v \in V \mid \ell(v) > d\}$ and $B = \{v \in V \mid \ell(v) < d\}$.
- We have $s \in A$ and $t \in B$ and there is no edge from A to B in the residual graph G_f; this means that (A, B) is a saturated cut.

Preflows

Lemma 66

A preflow that has a valid labelling saturates a cut.

Proof:

- There are n nodes but $n+1$ different labels from $0,\ldots,n$.
- There must exist a label $d \in \{0,\ldots,n\}$ such that none of the nodes carries this label.
- Let $A = \{v \in V \mid \ell(v) > d\}$ and $B = \{v \in V \mid \ell(v) < d\}$.
- We have $s \in A$ and $t \in B$ and there is no edge from A to B in the residual graph G_f; this means that (A,B) is a saturated cut.

Lemma 67

A flow that has a valid labelling is a maximum flow.

Push Relabel Algorithms

Push Relabel Algorithms

Idea:

- start with some preflow and some valid labelling

Push Relabel Algorithms

Idea:

- start with some preflow and some valid labelling
- successively change the preflow while maintaining a valid labelling

Push Relabel Algorithms

Idea:

- start with some preflow and some valid labelling
- successively change the preflow while maintaining a valid labelling
- stop when you have a flow (i.e., no more active nodes)

Changing a Preflow

An arc (u,v) with $c_f(u,v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e\in\text{into}(u)} f(e) - \sum_{e\in\text{out}(u)} f(e)$ and suppose $e = (u,v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Changing a Preflow

An arc (u,v) with $c_f(u,v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e \in \mathrm{into}(u)} f(e) - \sum_{e \in \mathrm{out}(u)} f(e)$ and suppose $e = (u,v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Changing a Preflow

An arc (u, v) with $c_f(u, v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e \in \text{into}(u)} f(e) - \sum_{e \in \text{out}(u)} f(e)$ and suppose $e = (u, v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Changing a Preflow

An arc (u,v) with $c_f(u,v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e\in\text{into}(u)} f(e) - \sum_{e\in\text{out}(u)} f(e)$ and suppose $e = (u,v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Changing a Preflow

An arc (u, v) with $c_f(u, v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e \in \text{into}(u)} f(e) - \sum_{e \in \text{out}(u)} f(e)$ and suppose $e = (u, v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Changing a Preflow

An arc (u, v) with $c_f(u, v) > 0$ in the residual graph is admissable if $\ell(u) = \ell(v) + 1$ (i.e., it goes downwards w.r.t. labelling ℓ).

The push operation

Consider an active node u with excess flow $f(u) = \sum_{e \in \text{into}(u)} f(e) - \sum_{e \in \text{out}(u)} f(e)$ and suppose $e = (u, v)$ is an admissable arc with residual capacity $c_f(e)$.

We can send flow $\min\{c_f(e), f(u)\}$ along e and obtain a new preflow. The old labelling is still valid (!!!).

- saturating push: $\min\{f(u), c_f(e)\} = c_f(e)$
  the arc e is deleted from the residual graph
- non-saturating push: $\min\{f(u), c_f(e)\} = f(u)$
  the node u becomes inactive

Push Relabel Algorithms

Push Relabel Algorithms

The relabel operation

Consider an active node u that does not have an outgoing admissable arc.

Push Relabel Algorithms

The relabel operation

Consider an active node u that does not have an outgoing admissable arc.

Increasing the label of u by 1 results in a valid labelling.

Push Relabel Algorithms

The relabel operation

Consider an active node u that does not have an outgoing admissable arc.

Increasing the label of u by 1 results in a valid labelling.

- Edges (w,u) incoming to u still fulfill their constraint $\ell(w) \le \ell(u) + 1$.

Push Relabel Algorithms

The relabel operation

Consider an active node u that does not have an outgoing admissable arc.

Increasing the label of u by 1 results in a valid labelling.

- Edges (w,u) incoming to u still fulfill their constraint $\ell(w) \leq \ell(u)+1$.
- An outgoing edge (u,w) had $\ell(u) < \ell(w)+1$ before since it was not admissable. Now: $\ell(u) \leq \ell(w)+1$.

Push Relabel Algorithms

Intuition:
We want to send flow downwards, since the source has a height/label of n and the target a height/label of 0. If we see an active node u with an admissible arc we push the flow at u towards the other end-point that has a lower height/label. If we do not have an admissible arc but excess flow into u it should roughly mean that the level/height/label of u should rise. (If we consider the flow to be water than this would be natural).

Note that the above intuition is very incorrect as the labels are integral, i.e., they cannot really be seen as the height of a node.

Push Relabel Algorithms

Algorithm 47 maxflow(G, s, t, c)

```
find initial preflow $f$
while there is active node $u$ do
    if there is admiss. arc $e$ out of $u$ then
        push$(G, e, f, c)$
    else
        relabel$(u)$
return $f$
```

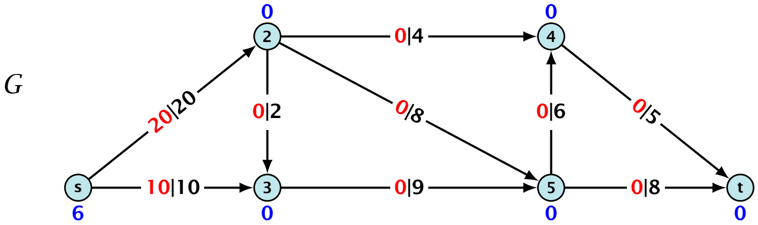
Push Relabel Algorithms

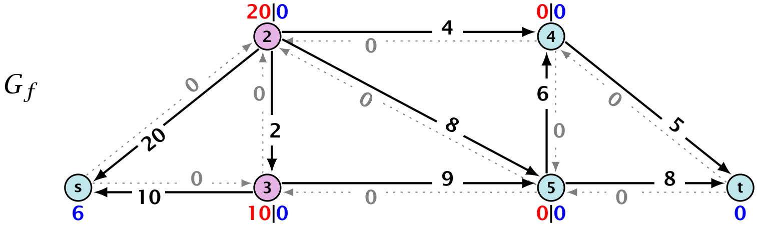
Algorithm 47 maxflow(G, s, t, c)

```
find initial preflow $f$
while there is active node $u$ do
    if there is admiss. arc $e$ out of $u$ then
        push$(G, e, f, c)$
    else
        relabel$(u)$
return $f$
```

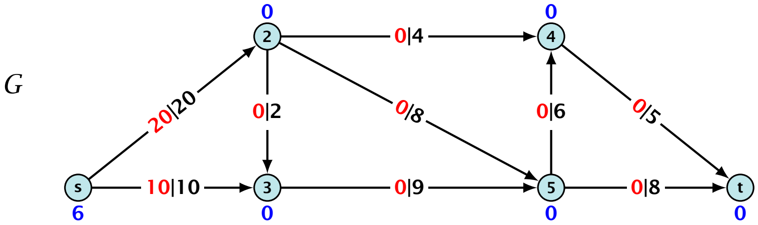
In the following example we always stick to the same active node u until it becomes inactive but this is not required.

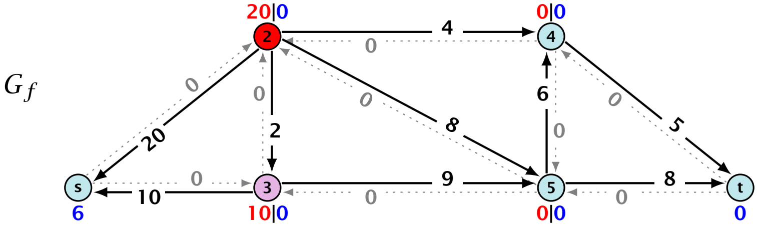
Preflow Push Algorithm





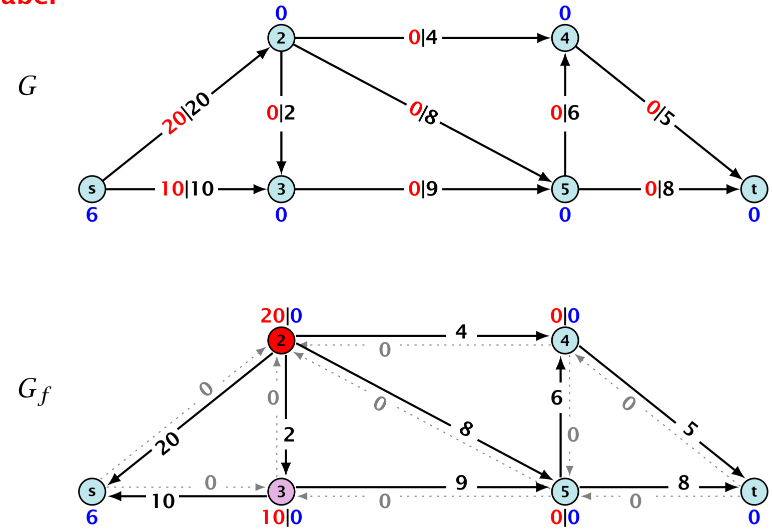
Preflow Push Algorithm



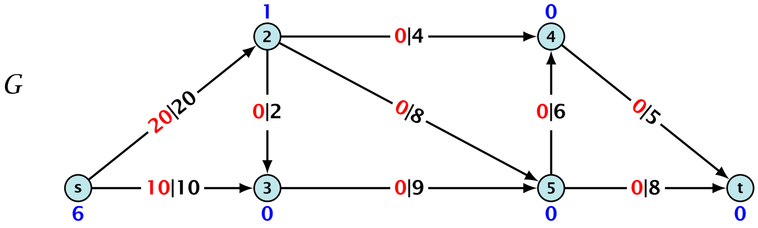


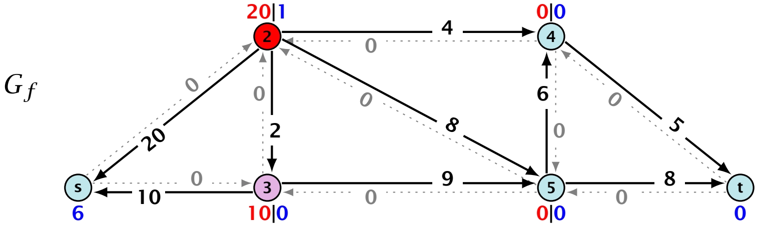
Preflow Push Algorithm

relabel



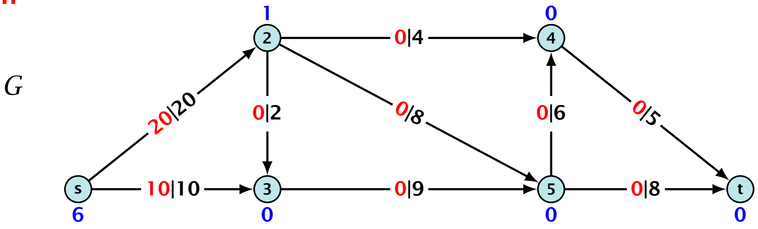
Preflow Push Algorithm

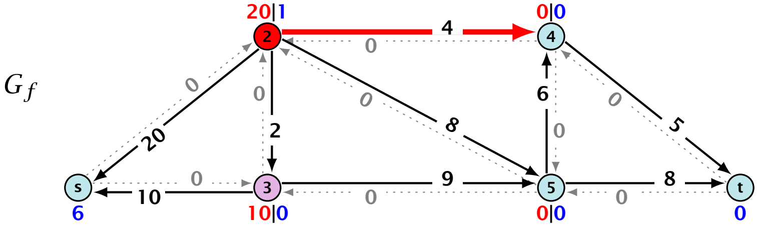




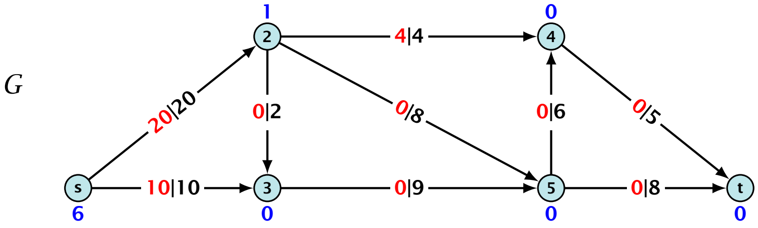
Preflow Push Algorithm

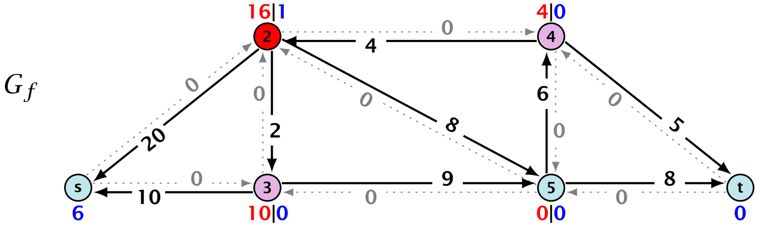
push





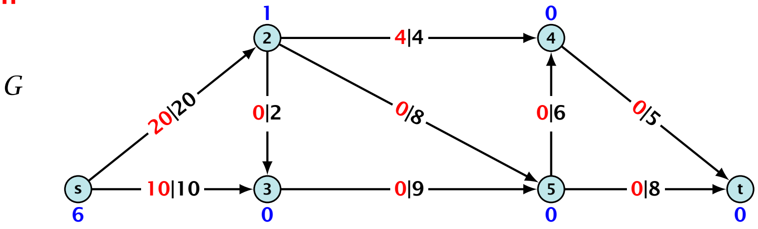
Preflow Push Algorithm

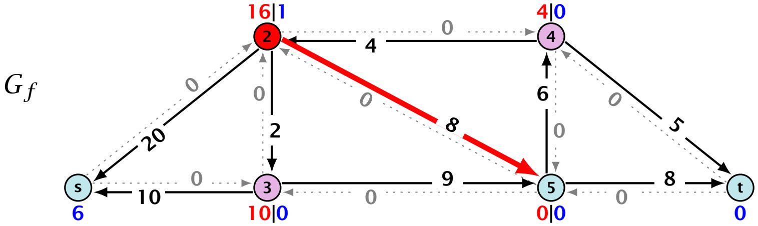




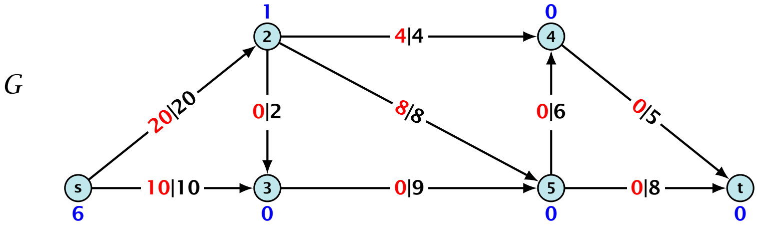
Preflow Push Algorithm

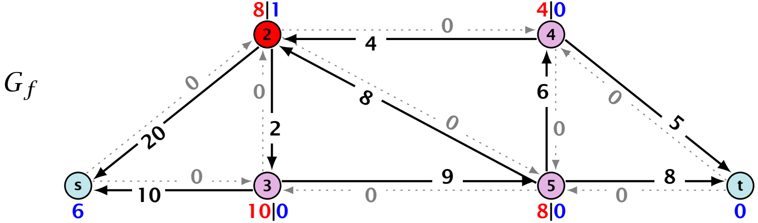
push





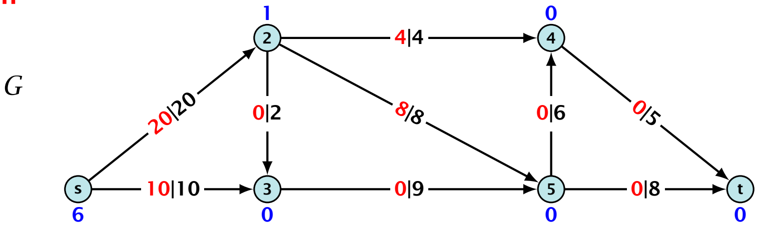
Preflow Push Algorithm

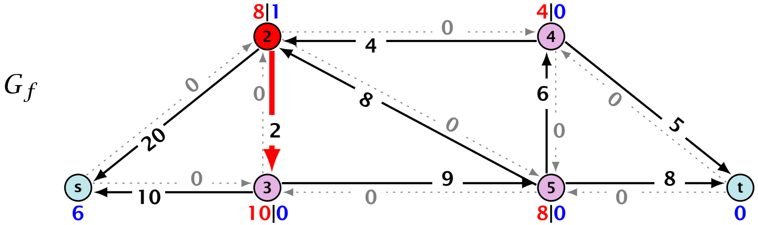




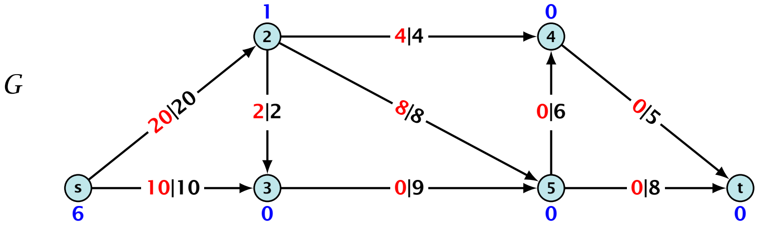
Preflow Push Algorithm

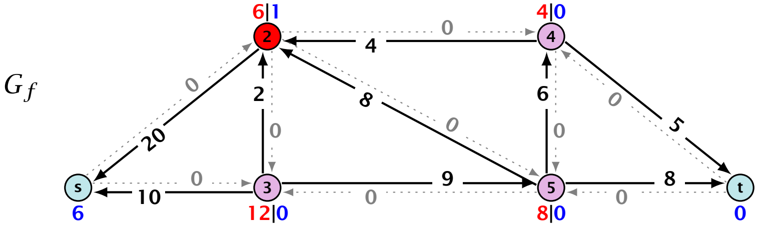
push





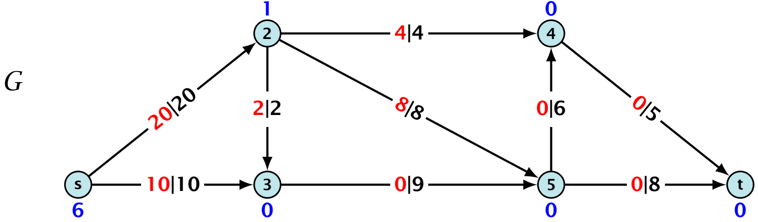
Preflow Push Algorithm

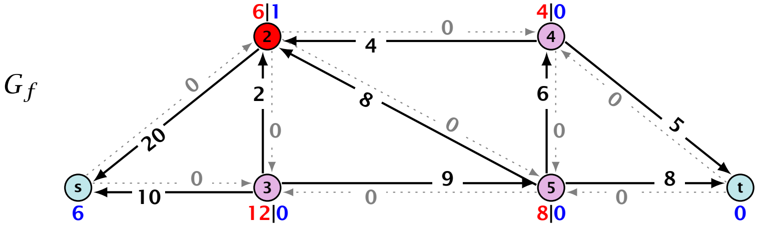




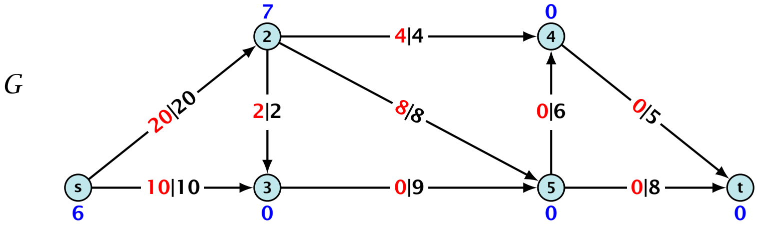
Preflow Push Algorithm

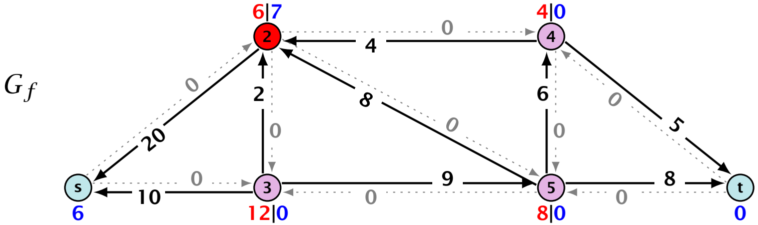
relabel 6 times





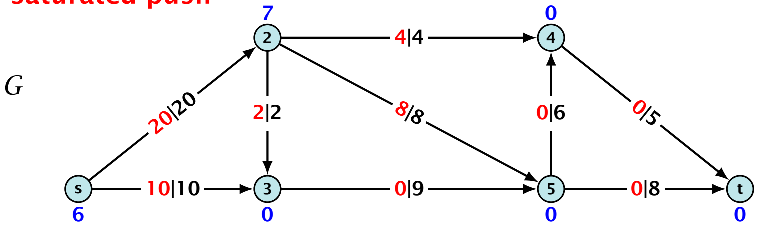
Preflow Push Algorithm

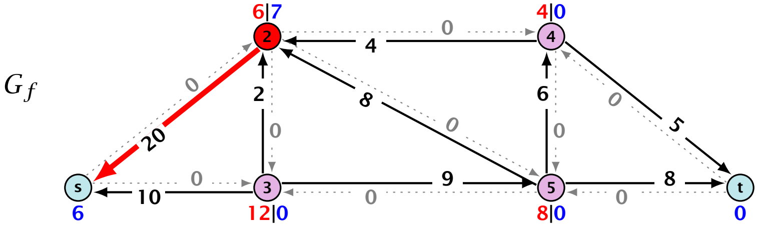




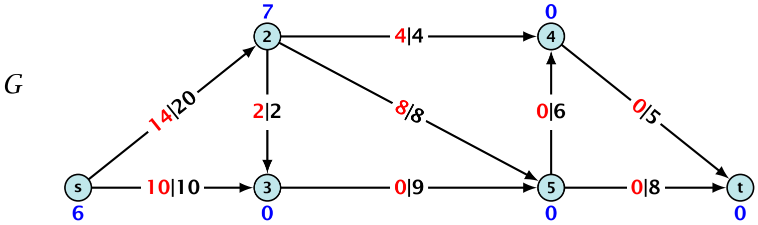
Preflow Push Algorithm

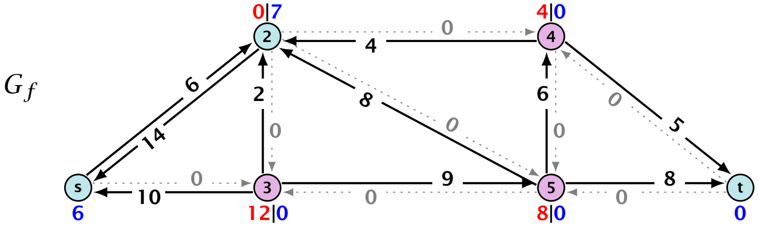
non-saturated push



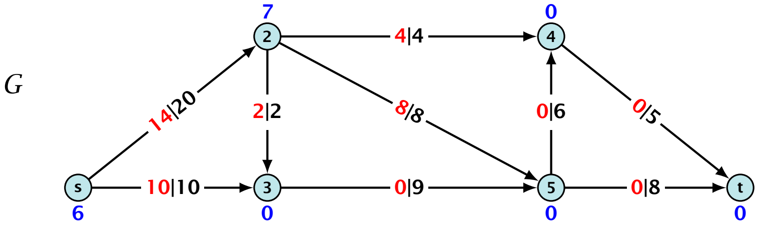


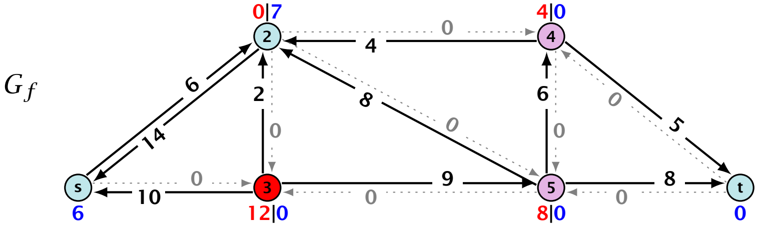
Preflow Push Algorithm





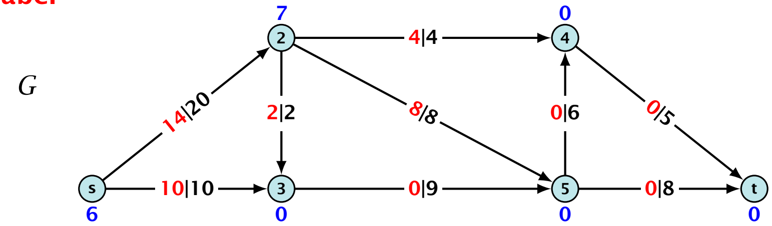
Preflow Push Algorithm

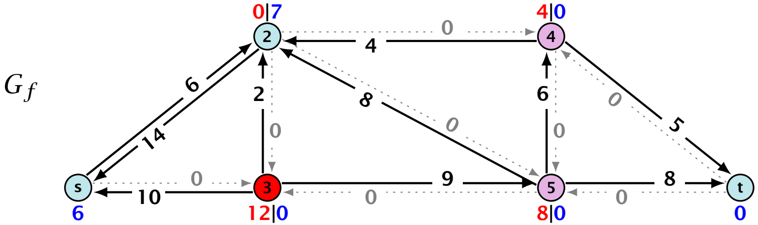




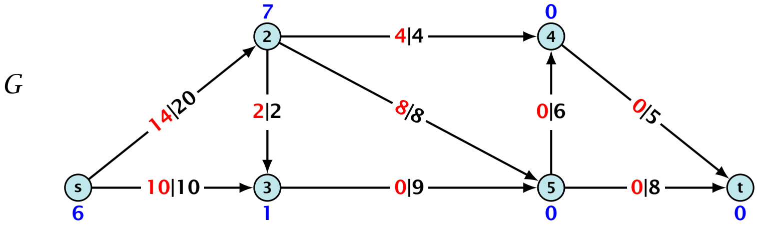
Preflow Push Algorithm

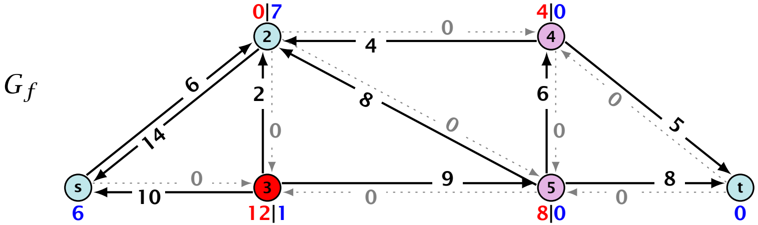
relabel





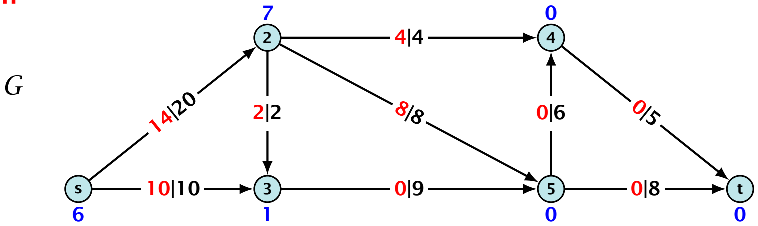
Preflow Push Algorithm

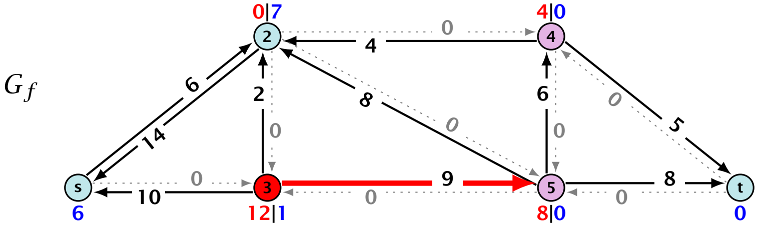




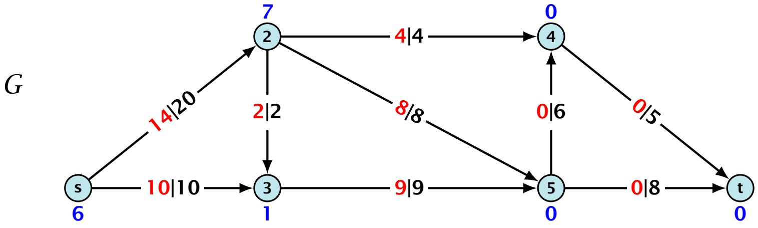
Preflow Push Algorithm

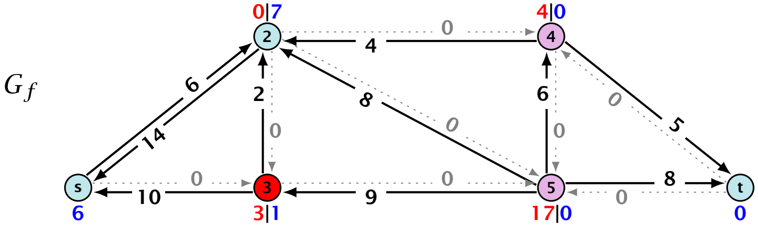
push





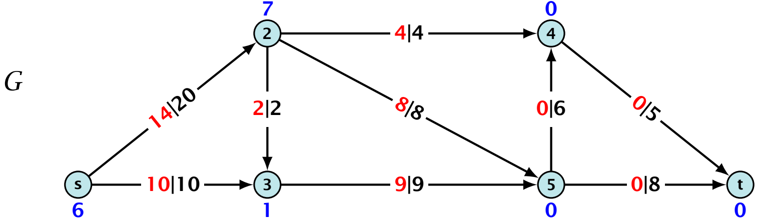
Preflow Push Algorithm

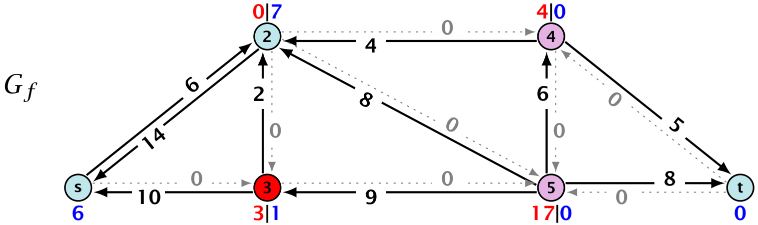




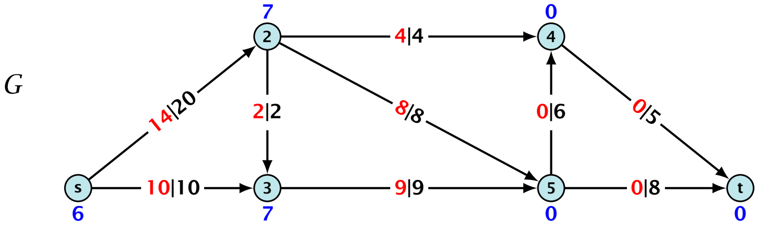
Preflow Push Algorithm

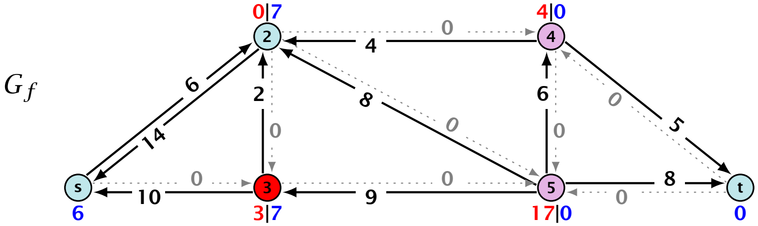
relabel 6 times





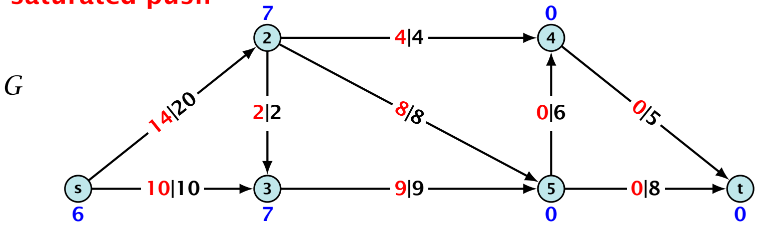
Preflow Push Algorithm

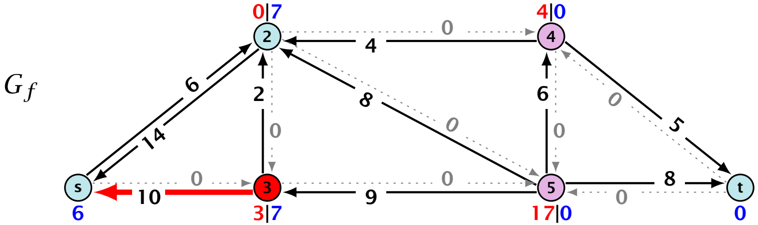




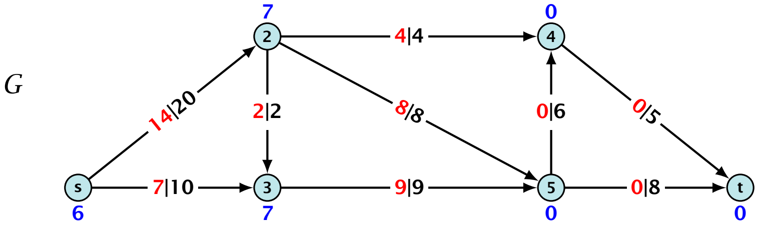
Preflow Push Algorithm

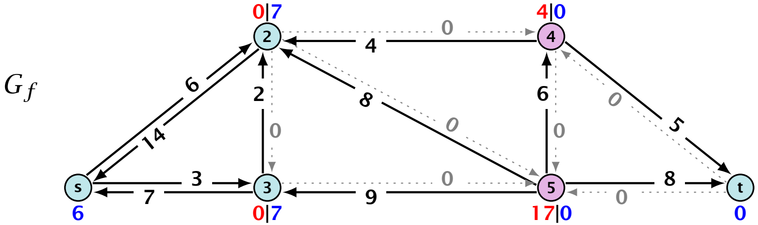
non-saturated push



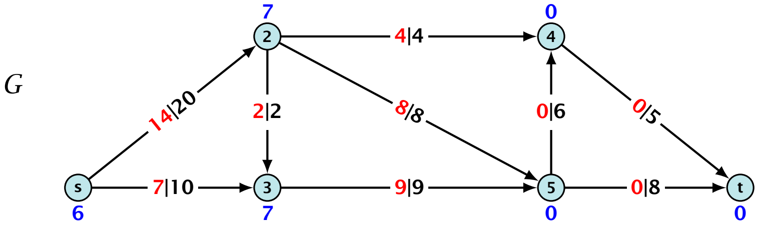


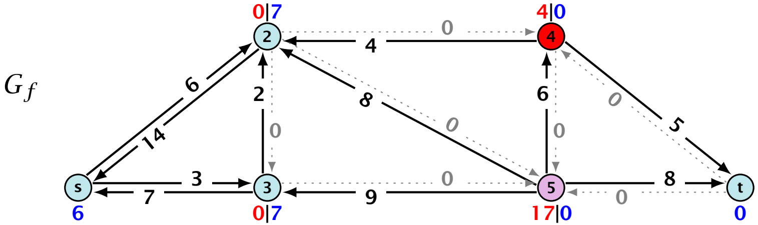
Preflow Push Algorithm





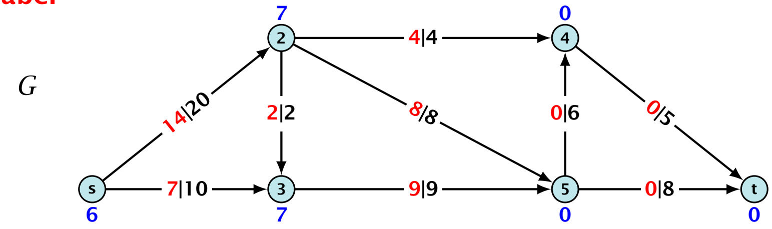
Preflow Push Algorithm

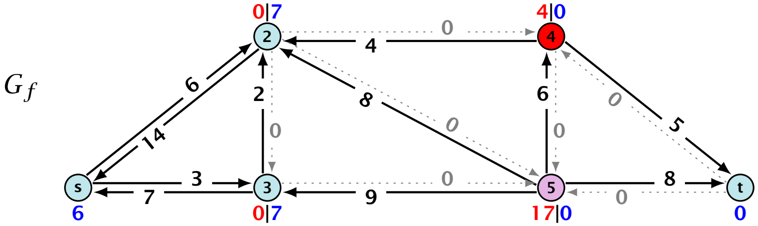




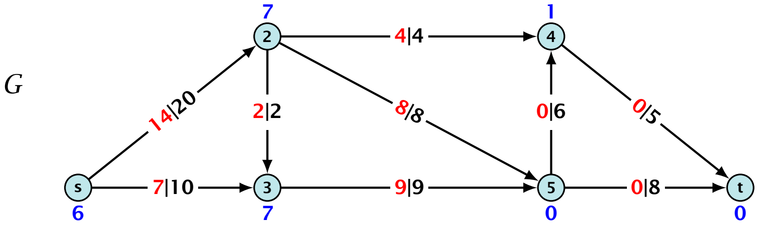
Preflow Push Algorithm

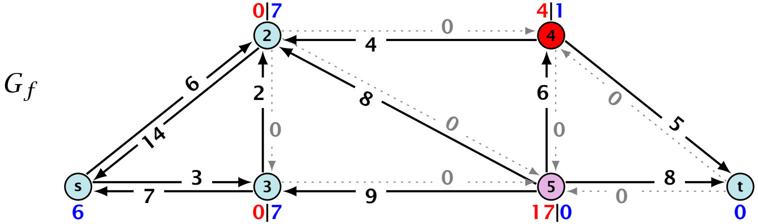
relabel





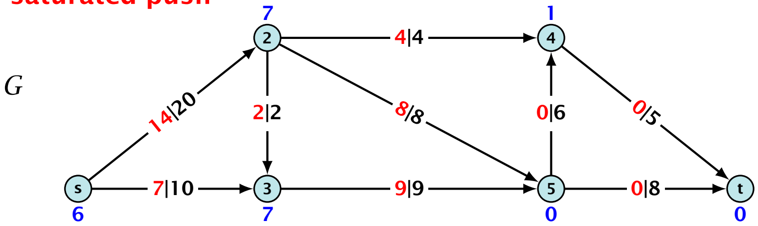
Preflow Push Algorithm

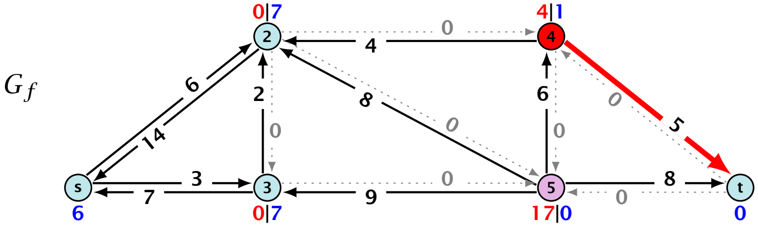




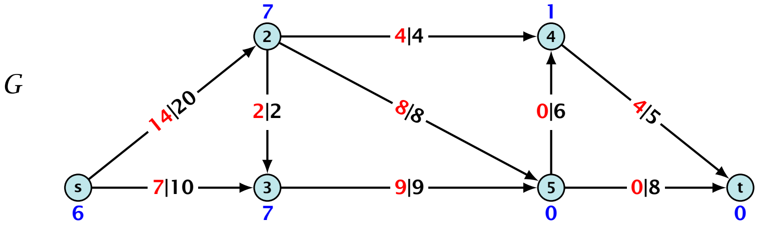
Preflow Push Algorithm

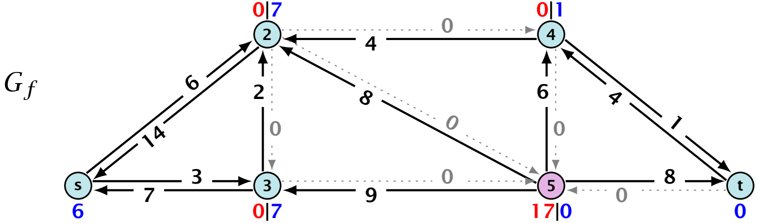
non-saturated push



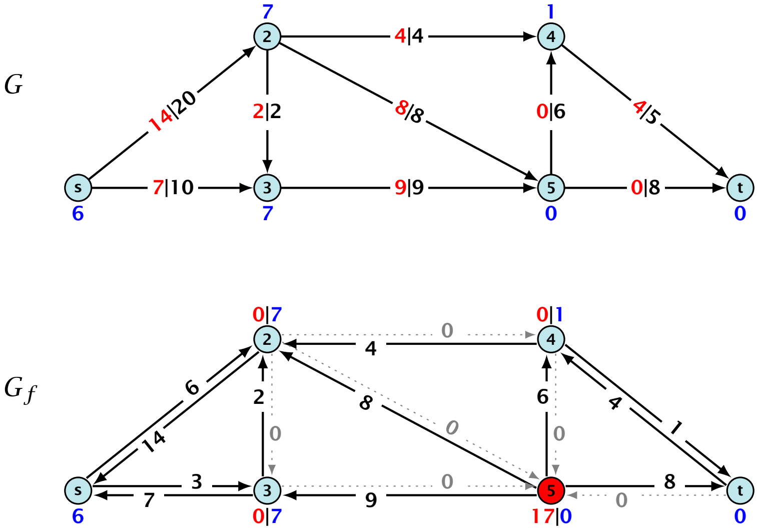


Preflow Push Algorithm



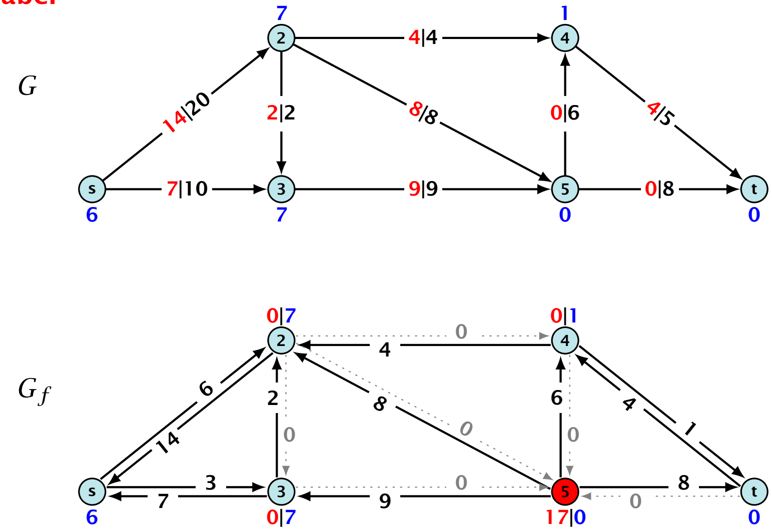


Preflow Push Algorithm

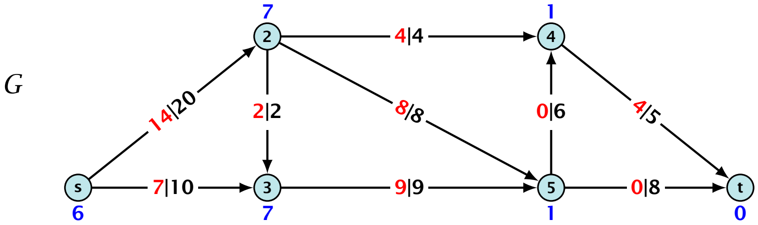


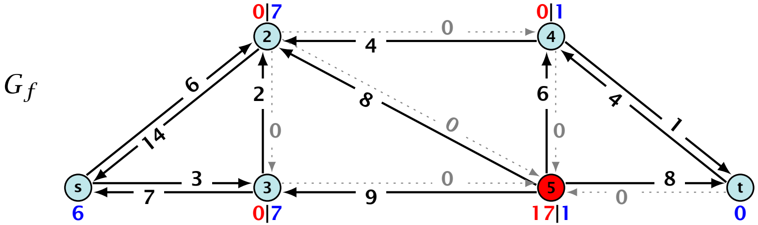
Preflow Push Algorithm

relabel



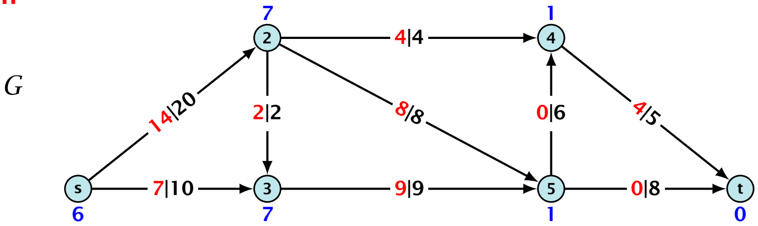
Preflow Push Algorithm

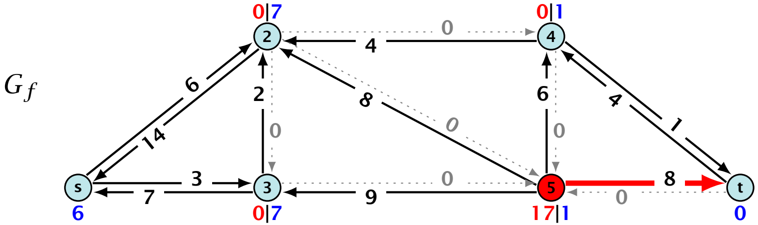




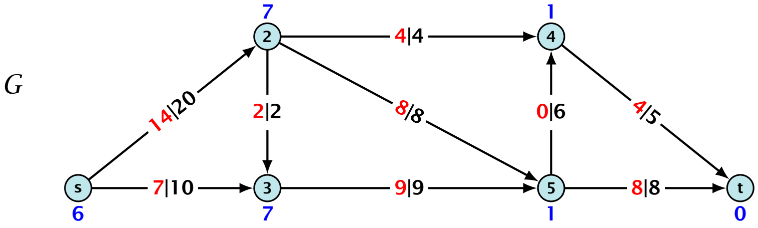
Preflow Push Algorithm

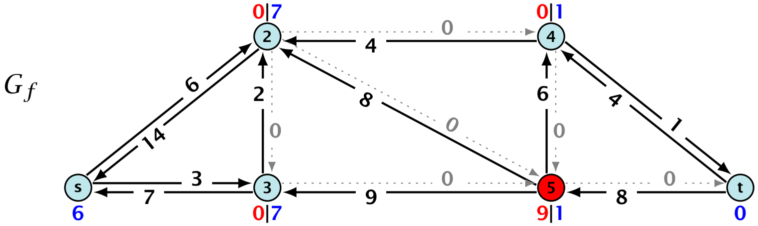
push





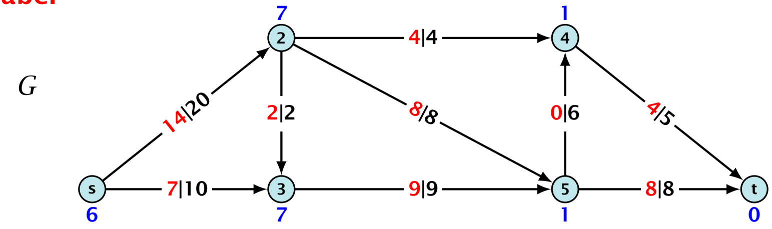
Preflow Push Algorithm

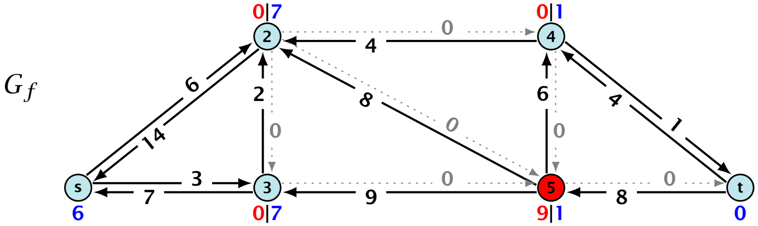




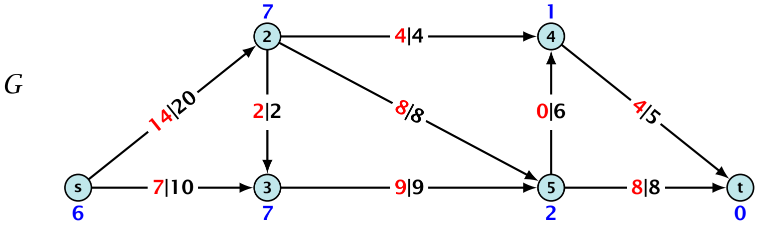
Preflow Push Algorithm

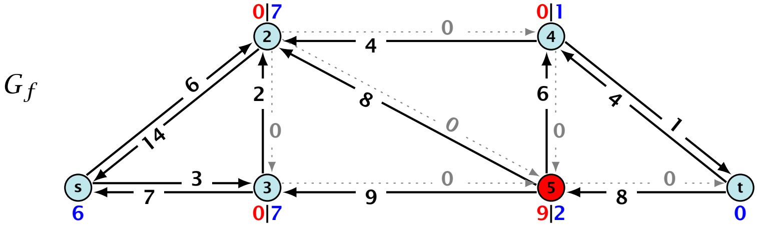
relabel





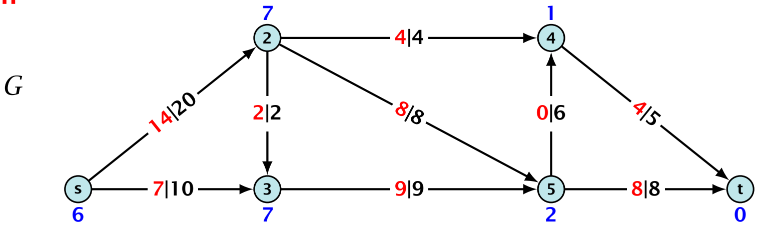
Preflow Push Algorithm

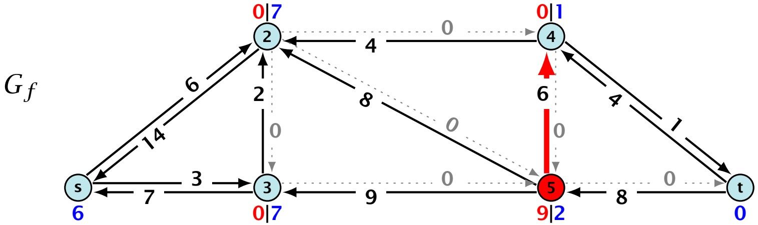




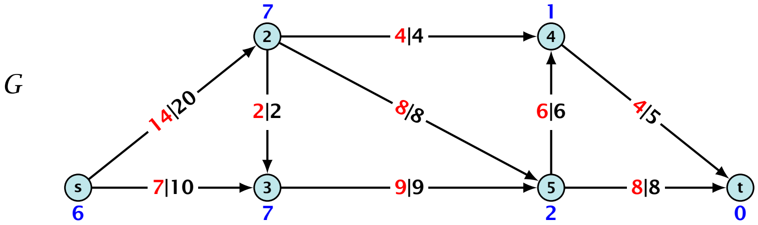
Preflow Push Algorithm

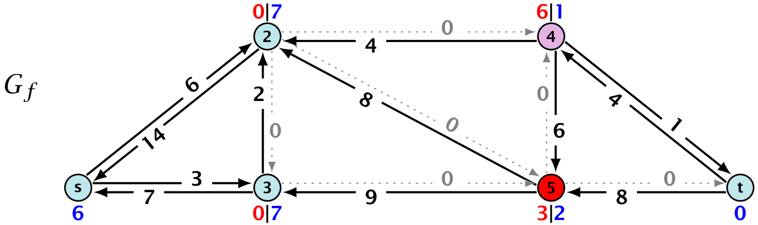
push





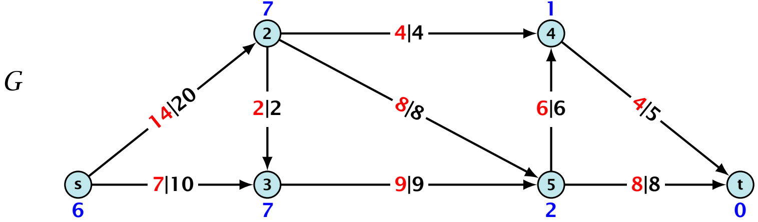
Preflow Push Algorithm

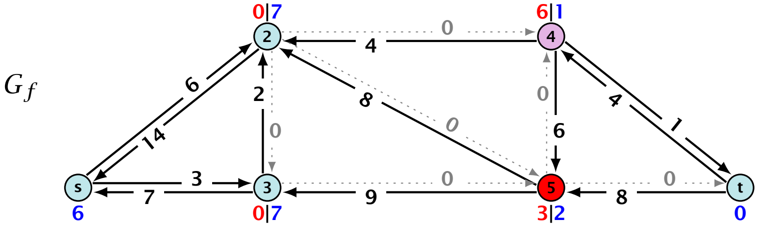




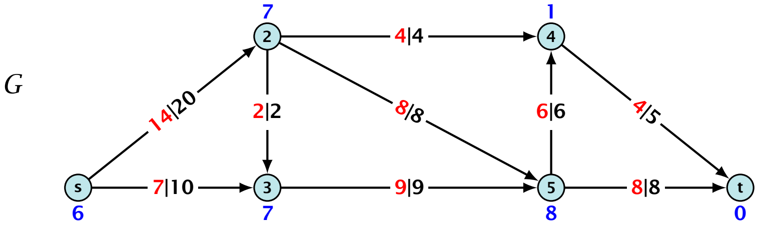
Preflow Push Algorithm

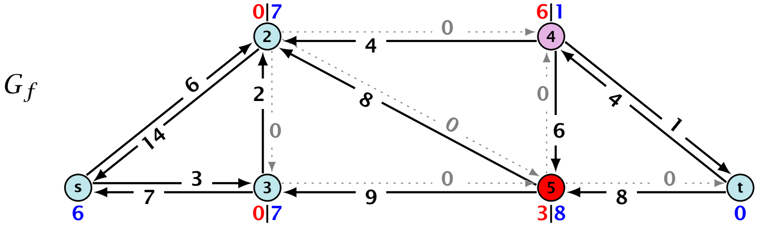
relabel 6 times





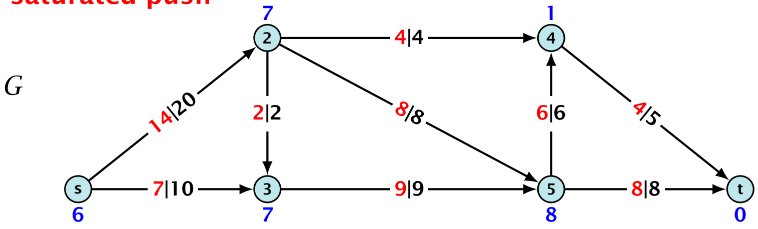
Preflow Push Algorithm

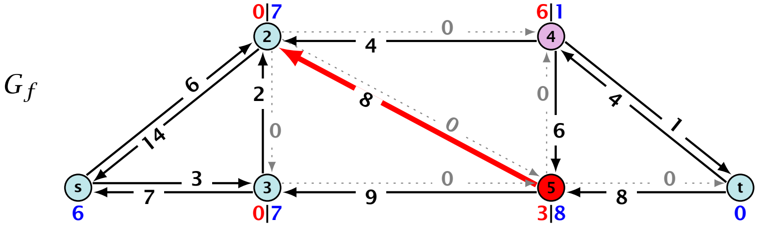




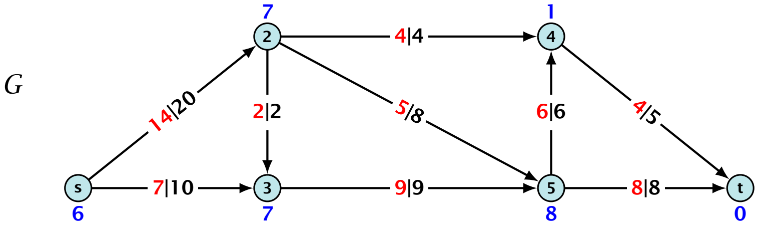
Preflow Push Algorithm

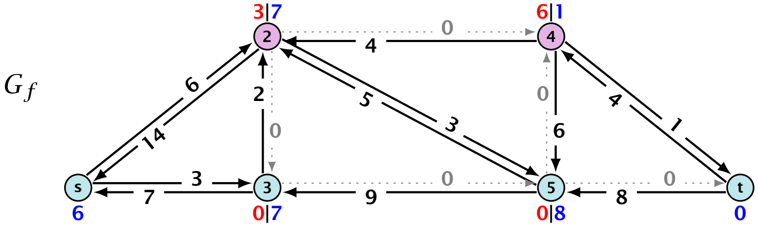
non-saturated push



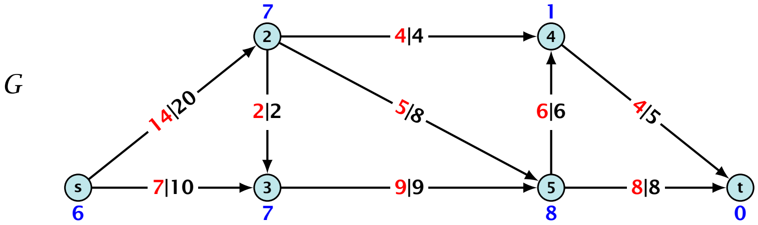


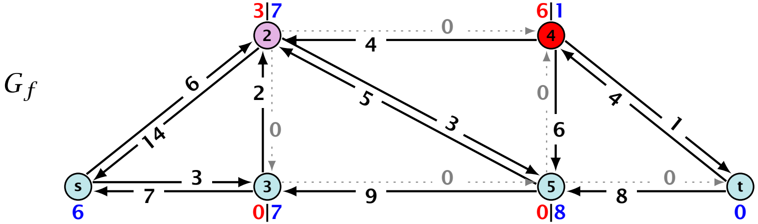
Preflow Push Algorithm





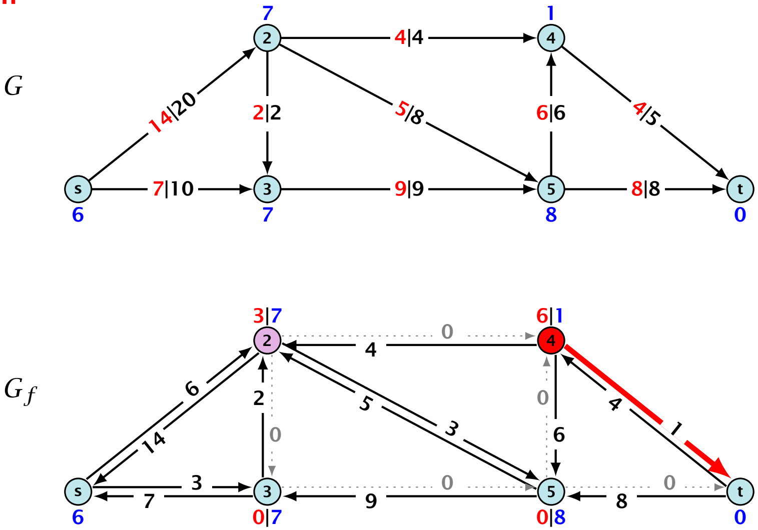
Preflow Push Algorithm



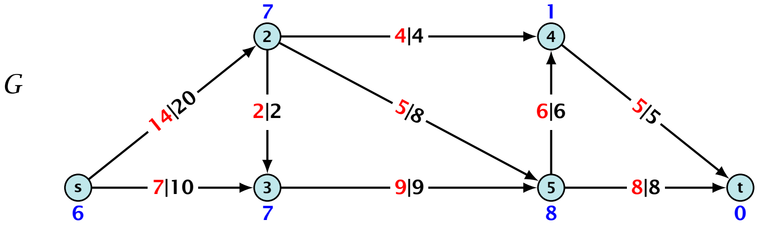


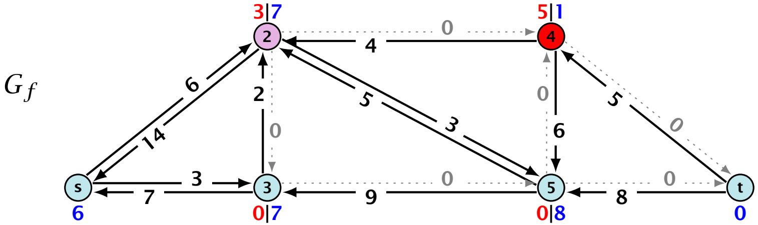
Preflow Push Algorithm

push



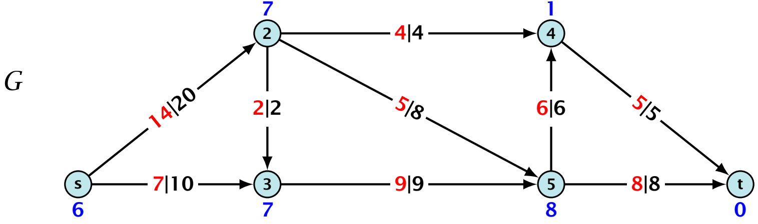
Preflow Push Algorithm

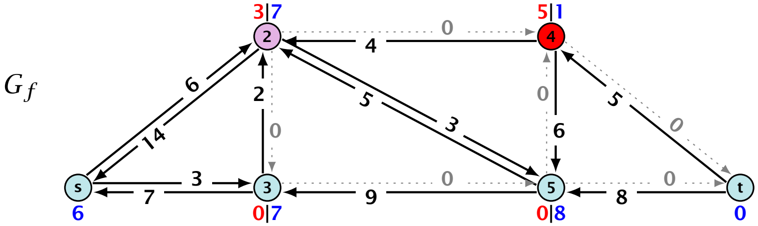




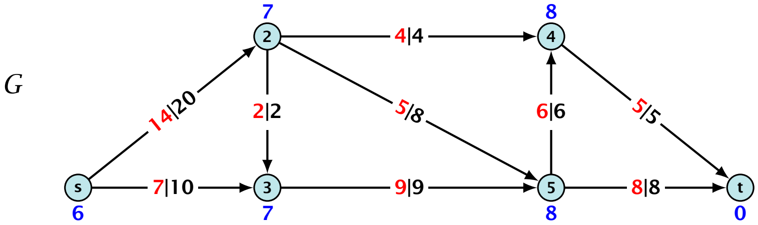
Preflow Push Algorithm

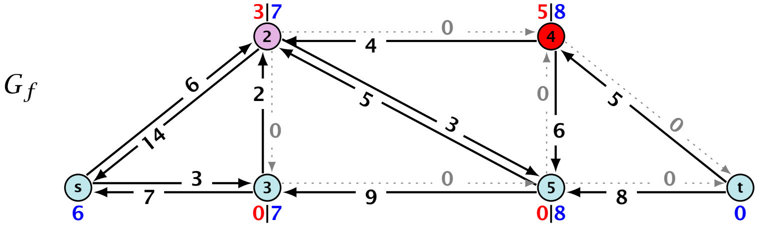
relabel 7 times





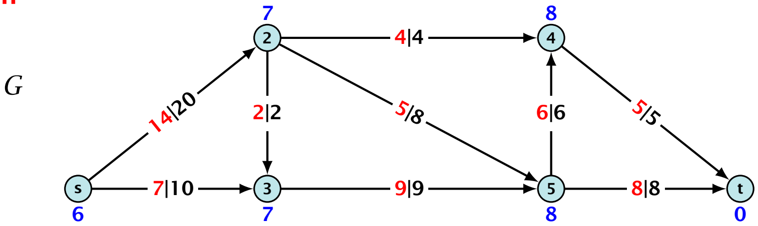
Preflow Push Algorithm

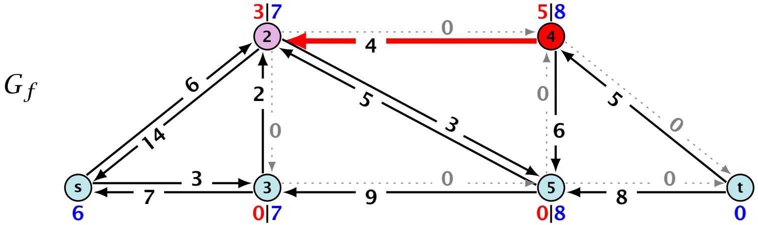




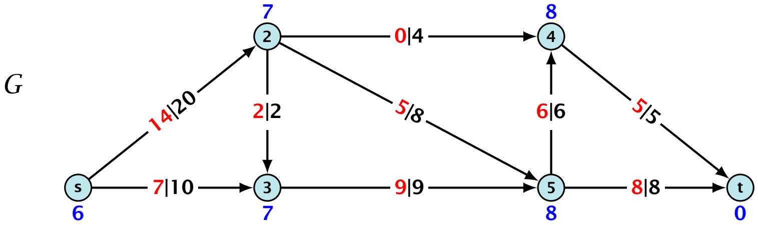
Preflow Push Algorithm

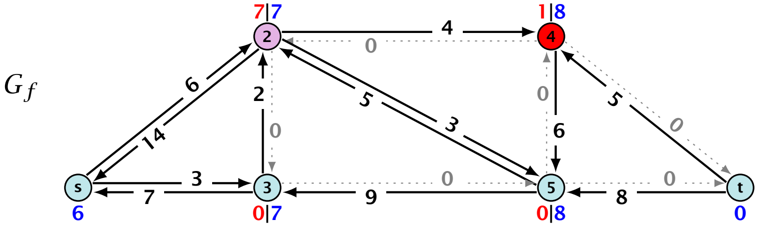
push





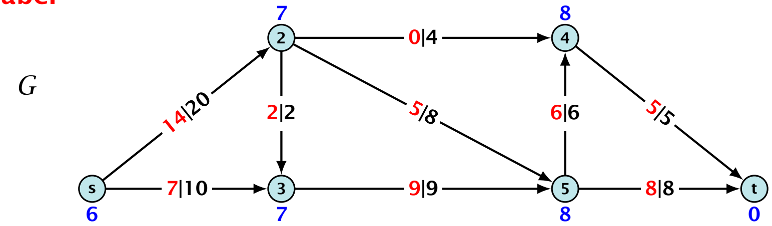
Preflow Push Algorithm

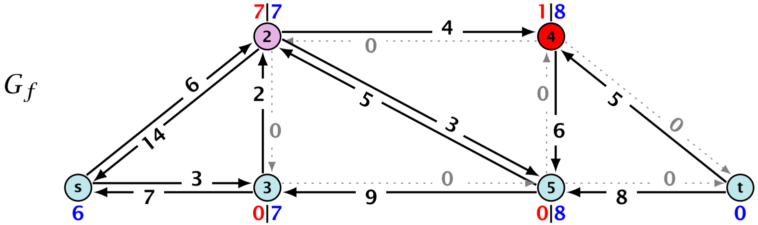




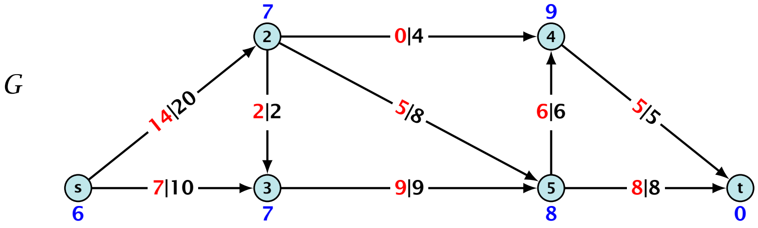
Preflow Push Algorithm

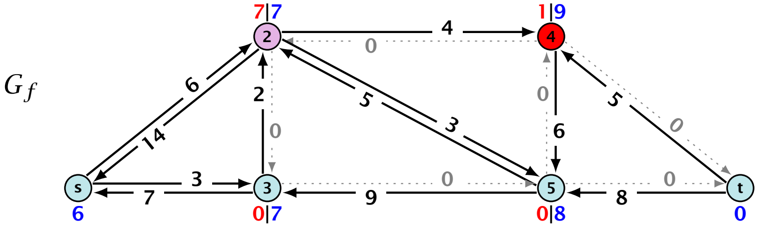
relabel





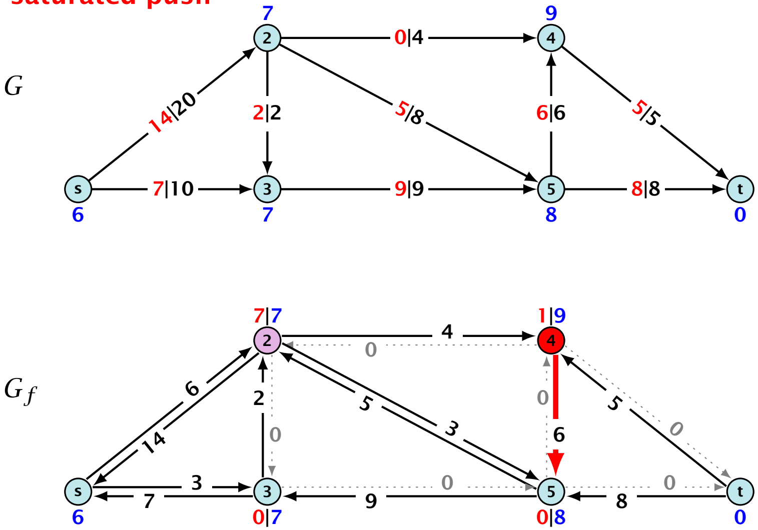
Preflow Push Algorithm



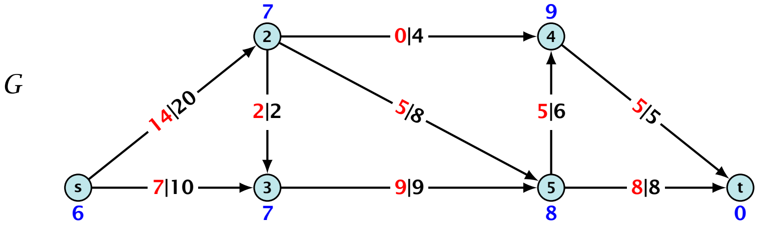


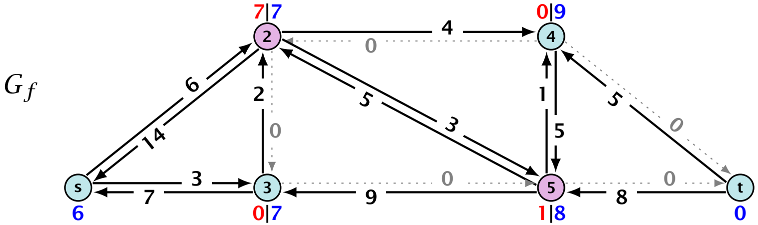
Preflow Push Algorithm

non-saturated push

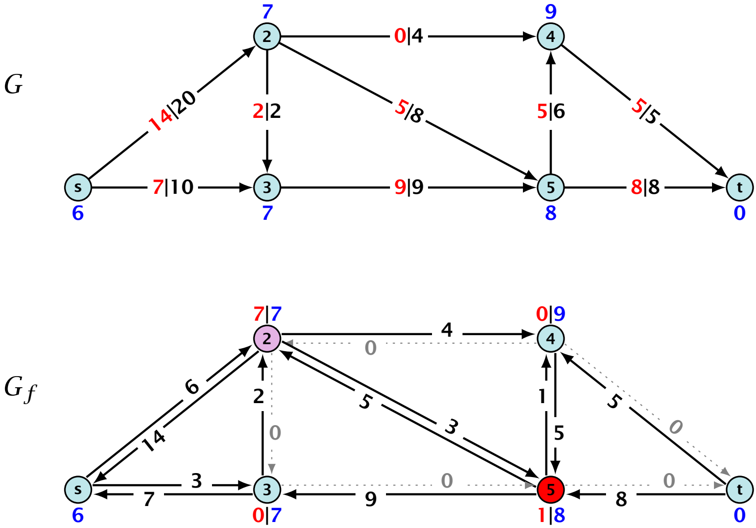


Preflow Push Algorithm



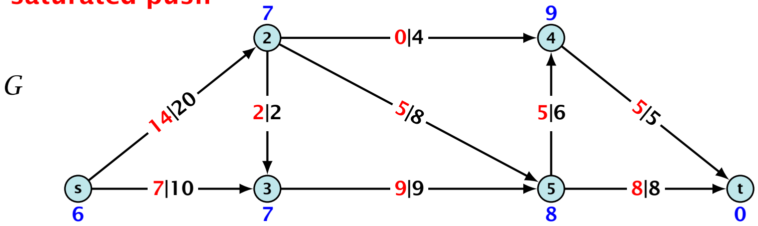


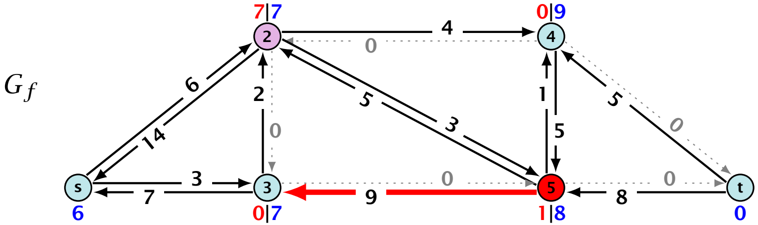
Preflow Push Algorithm



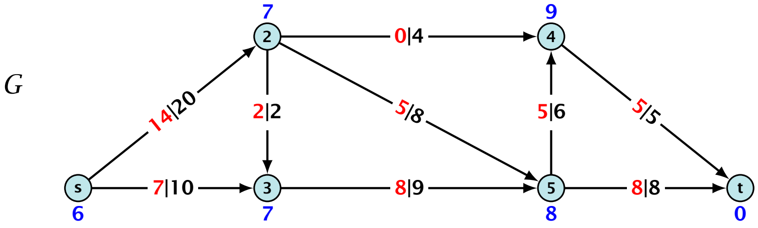
Preflow Push Algorithm

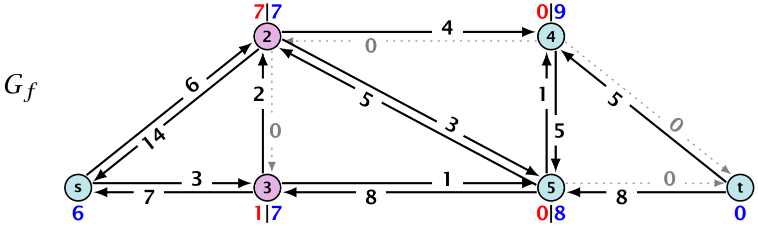
non-saturated push



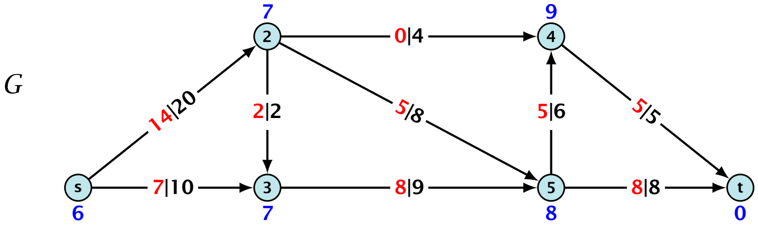


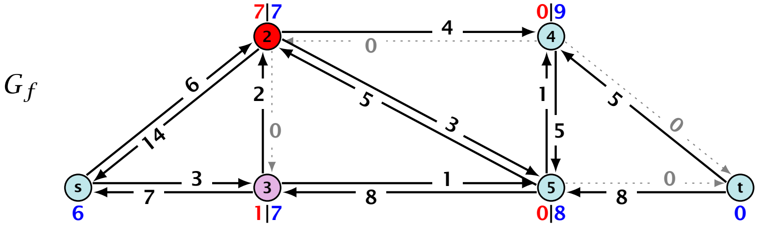
Preflow Push Algorithm





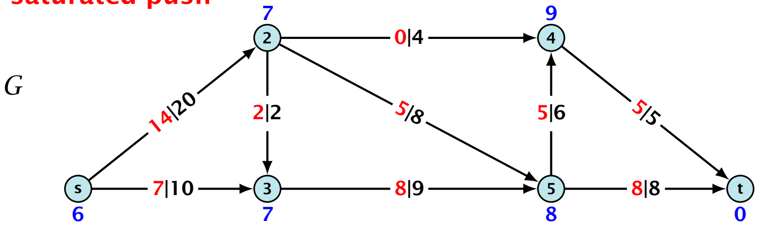
Preflow Push Algorithm

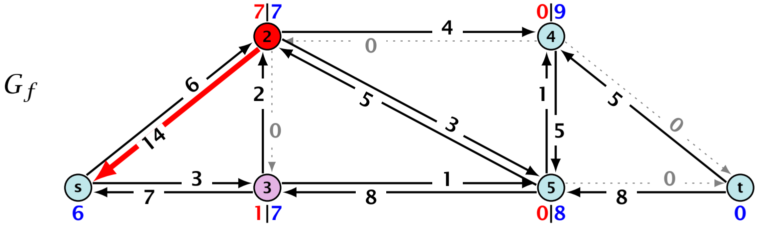




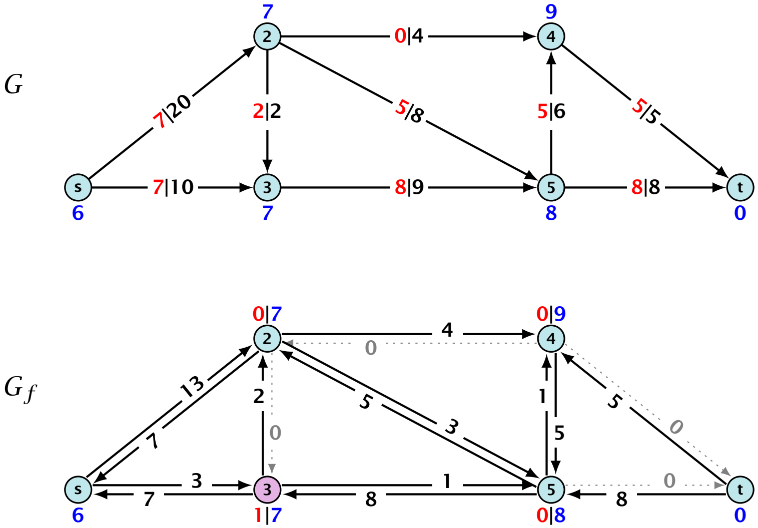
Preflow Push Algorithm

non-saturated push

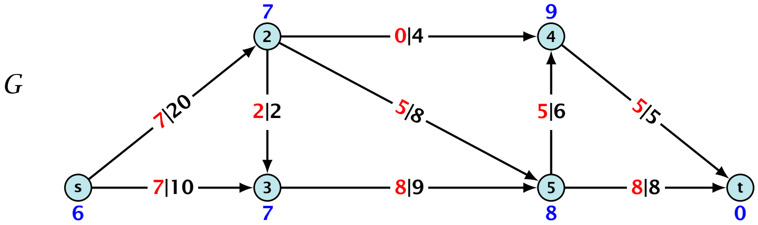


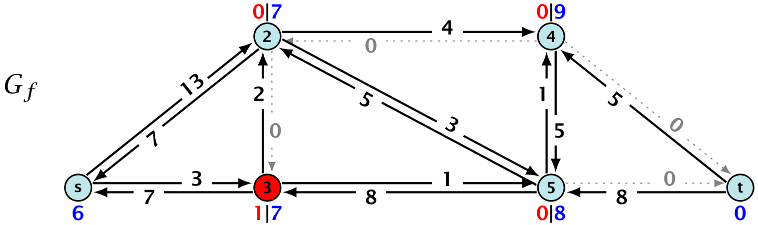


Preflow Push Algorithm



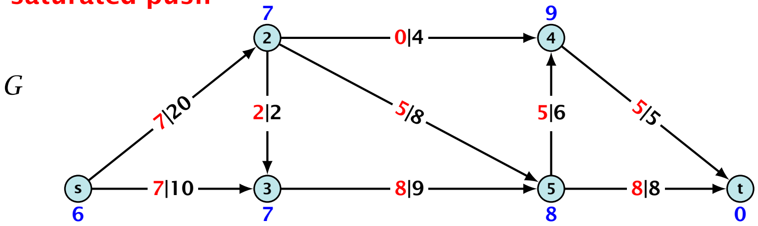
Preflow Push Algorithm

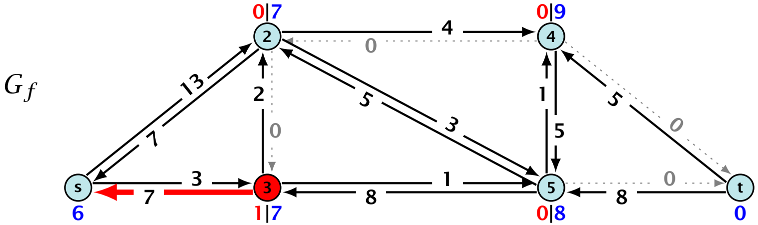




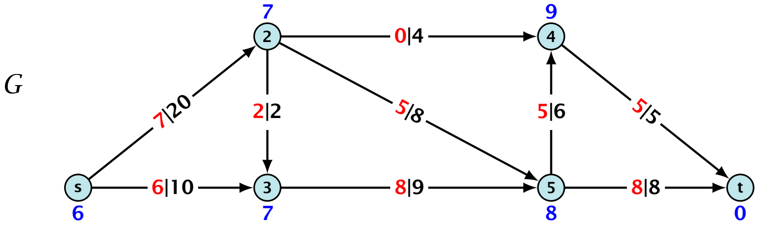
Preflow Push Algorithm

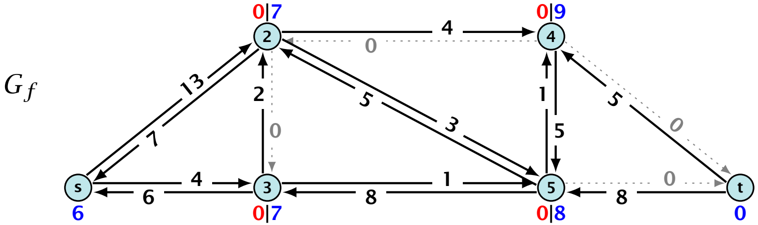
non-saturated push



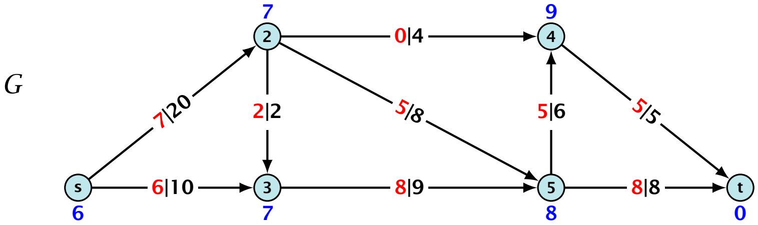


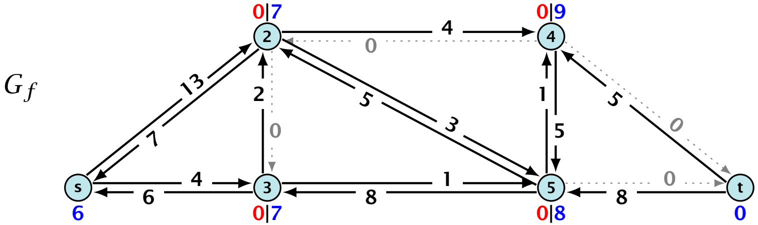
Preflow Push Algorithm





Preflow Push Algorithm





Analysis

Lemma 68

An active node has a path to s in the residual graph.

Analysis

Lemma 68

An active node has a path to s in the residual graph.

Proof.

- Let A denote the set of nodes that can reach s, and let B denote the remaining nodes. Note that $s \in A$.

Analysis

Lemma 68

An active node has a path to s in the residual graph.

Proof.

- Let A denote the set of nodes that can reach s, and let B denote the remaining nodes. Note that $s \in A$.
- In the following we show that a node $b \in B$ has excess flow $f(b) = 0$ which gives the lemma.

Analysis

Lemma 68

An active node has a path to s in the residual graph.

Proof.

- Let A denote the set of nodes that can reach s, and let B denote the remaining nodes. Note that $s \in A$.
- In the following we show that a node $b \in B$ has excess flow $f(b) = 0$ which gives the lemma.
- In the residual graph there are no edges into A, and, hence, no edges leaving A/entering B can carry any flow.

Analysis

Lemma 68

An active node has a path to s in the residual graph.

Proof.

- Let A denote the set of nodes that can reach s, and let B denote the remaining nodes. Note that $s \in A$.
- In the following we show that a node $b \in B$ has excess flow $f(b) = 0$ which gives the lemma.
- In the residual graph there are no edges into A, and, hence, no edges leaving A/entering B can carry any flow.
- Let $f(B) = \sum_{v \in B} f(v)$ be the excess flow of all nodes in B.

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$f(B)$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$f(B) = \sum_{b \in B} f(b)$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b \in B} f(b) \\ &= \sum_{b \in B} \left(\sum_{v \in V} f(v,b) - \sum_{v \in V} f(b,v) \right) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B} \left(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v) \right) \\ &= \sum_{b\in B} \left(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v) \right) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B}\left(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\right) \\ &= \sum_{b\in B}\left(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\right) \\ &= \sum_{b\in B}\sum_{v\in A} f(v,b) - \sum_{b\in B}\sum_{v\in A} f(b,v) + \sum_{b\in B}\sum_{v\in B} f(v,b) - \sum_{b\in B}\sum_{v\in B} f(b,v) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned}
f(B) &= \sum_{b\in B} f(b) \\
&= \sum_{b\in B}\left(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\right) \\
&= \sum_{b\in B}\left(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\right) \\
&= \sum_{b\in B}\sum_{v\in A} f(v,b) - \sum_{b\in B}\sum_{v\in A} f(b,v) + \underbrace{\boxed{\sum_{b\in B}\sum_{v\in B} f(v,b) - \sum_{b\in B}\sum_{v\in B} f(b,v)}}_{=\,\mathbf{0}}
\end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B}\Big(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\Big) \\ &= \sum_{b\in B}\Big(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\Big) \\ &= \sum_{b\in B}\sum_{v\in A} f(v,b) - \sum_{b\in B}\sum_{v\in A} f(b,v) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b \in B} f(b) \\ &= \sum_{b \in B} \Bigg(\sum_{v \in V} f(v,b) - \sum_{v \in V} f(b,v) \Bigg) \\ &= \sum_{b \in B} \Bigg(\sum_{v \in A} f(v,b) + \sum_{v \in B} f(v,b) - \sum_{v \in A} f(b,v) - \sum_{v \in B} f(b,v) \Bigg) \\ &= \sum_{b \in B} \sum_{v \in A} \underbrace{\boxed{f(v,b)}}_{=\,\mathbf{0}} - \sum_{b \in B} \sum_{v \in A} f(b,v) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B}\Bigg(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\Bigg) \\ &= \sum_{b\in B}\Bigg(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\Bigg) \\ &= \boxed{\sum_{b\in B}\sum_{v\in A} f(v,b)} - \sum_{b\in B}\sum_{v\in A} f(b,v) \end{aligned}$$

$$= \mathbf{0}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B} \left(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v) \right) \\ &= \sum_{b\in B} \left(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v) \right) \\ &= \qquad\qquad - \sum_{b\in B} \sum_{v\in A} f(b,v) \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b \in B} f(b) \\ &= \sum_{b \in B} \Big(\sum_{v \in V} f(v,b) - \sum_{v \in V} f(b,v) \Big) \\ &= \sum_{b \in B} \Big(\sum_{v \in A} f(v,b) + \sum_{v \in B} f(v,b) - \sum_{v \in A} f(b,v) - \sum_{v \in B} f(b,v) \Big) \\ &= \qquad - \sum_{b \in B} \sum_{v \in A} \underbrace{\boxed{f(b,v)}}_{\geq \mathbf{0}} \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B}\left(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\right) \\ &= \sum_{b\in B}\left(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\right) \\ &= \qquad\qquad\qquad -\sum_{b\in B}\sum_{v\in A} f(b,v) \\ &\le 0 \end{aligned}$$

Let $f : E \to \mathbb{R}_0^+$ be a preflow. We introduce the notation

$$f(x,y) = \begin{cases} 0 & (x,y) \notin E \\ f((x,y)) & (x,y) \in E \end{cases}$$

We have

$$\begin{aligned} f(B) &= \sum_{b\in B} f(b) \\ &= \sum_{b\in B}\Bigg(\sum_{v\in V} f(v,b) - \sum_{v\in V} f(b,v)\Bigg) \\ &= \sum_{b\in B}\Bigg(\sum_{v\in A} f(v,b) + \sum_{v\in B} f(v,b) - \sum_{v\in A} f(b,v) - \sum_{v\in B} f(b,v)\Bigg) \\ &= \qquad\qquad - \sum_{b\in B}\sum_{v\in A} f(b,v) \\ &\le 0 \end{aligned}$$

Hence, the excess flow $f(b)$ must be 0 for every node $b \in B$.

Analysis

Lemma 69

The label of a node cannot become larger than $2n-1$.

Proof.

- When increasing the label at a node u there exists a path from u to s of length at most $n-1$. Along each edge of the path the height/label can at most drop by 1, and the label of the source is n.

Analysis

Lemma 70

There are only $\mathcal{O}(n^3)$ calls to discharge when using the relabel-to-front heuristic.

Proof.

- When increasing the label at a node u there exists a path from u to s of length at most $n-1$. Along each edge of the path the height/label can at most drop by 1, and the label of the source is n.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u, v).

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u, v).
- Hence, the edge (u, v) is deleted from the residual graph.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u,v).
- Hence, the edge (u,v) is deleted from the residual graph.
- For the edge to appear again, a push from v to u is required.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u, v).
- Hence, the edge (u, v) is deleted from the residual graph.
- For the edge to appear again, a push from v to u is required.
- Currently, $\ell(u) = \ell(v) + 1$, as we only make pushes along admissable edges.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u,v).
- Hence, the edge (u,v) is deleted from the residual graph.
- For the edge to appear again, a push from v to u is required.
- Currently, $\ell(u) = \ell(v) + 1$, as we only make pushes along admissable edges.
- For a push from v to u the edge (v,u) must become admissable. The label of v must increase by at least 2.

Lemma 71

The number of saturating pushes performed is at most $\mathcal{O}(mn)$.

Proof.

- Suppose that we just made a saturating push along (u, v).
- Hence, the edge (u, v) is deleted from the residual graph.
- For the edge to appear again, a push from v to u is required.
- Currently, $\ell(u) = \ell(v) + 1$, as we only make pushes along admissable edges.
- For a push from v to u the edge (v, u) must become admissable. The label of v must increase by at least 2.
- Since the label of v is at most $2n - 1$, there are at most n pushes along (u, v).

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes} v} \ell(v)$

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes}v} \ell(v)$
- A saturating push increases Φ by at most $2n$.

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes} v} \ell(v)$
- A saturating push increases Φ by at most $2n$.
- A relabel increases Φ by at most 1.

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes} v} \ell(v)$
- A saturating push increases Φ by at most $2n$.
- A relabel increases Φ by at most 1.
- A non-saturating push decreases Φ by at least 1 as the node that is pushed from becomes inactive and has a label that is strictly larger than the target.

Lemma 72

The number of non-saturating pushes performed is at most $\mathcal{O}(n^2m)$.

Proof.

- Define a potential function $\Phi(f) = \sum_{\text{active nodes}v} \ell(v)$
- A saturating push increases Φ by at most $2n$.
- A relabel increases Φ by at most 1.
- A non-saturating push decreases Φ by at least 1 as the node that is pushed from becomes inactive and has a label that is strictly larger than the target.
- Hence,

$$\begin{aligned}\text{\#non-saturating\_pushes} &\leq \text{\#relabels} + 2n \cdot \text{\#saturating\_pushes}\\ &\leq \mathcal{O}(n^2m)\ .\end{aligned}$$

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissible

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissible

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u,v) can be performed in constant time

- check whether edge (v,u) needs to be added to G_f
- check whether (u,v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

Analysis

There is an implementation of the generic push relabel algorithm with running time $\mathcal{O}(n^2m)$.

For every node maintain a list of admissable edges starting at that node. Further maintain a list of active nodes.

A push along an edge (u, v) can be performed in constant time

- check whether edge (v, u) needs to be added to G_f
- check whether (u, v) needs to be deleted (saturating push)
- check whether u becomes inactive and has to be deleted from the set of active nodes

A relabel at a node u can be performed in time $\mathcal{O}(n)$

- check for all outgoing edges if they become admissable
- check for all incoming edges if they become non-admissable

13.2 Relabel to front

For special variants of push relabel algorithms we organize the neighbours of a node into a linked list (possible neighbours in the residual graph G_f). Then we use the discharge-operation:

Algorithm 48 discharge(u)

```
while $u$ is active do
    $v \leftarrow u.\textit{current-neighbour}$
    if $v = \text{null}$ then
        relabel($u$)
        $u.\textit{current-neighbour} \leftarrow u.\textit{neighbour-list-head}$
    else
        if $(u, v)$ admissable then push($u, v$)
        else $u.\textit{current-neighbour} \leftarrow v.\textit{next-in-list}$
```

13.2 Relabel to front

Lemma 73
If $v = \text{null}$ *in line 3, then there is no outgoing admissable edge from* u*.*

The lemma holds because push- and relabel-operations on nodes different from u cannot make edges outgoing from u admissable.

This shows that discharge(u) is correct, and that we can perform a relabel in line 4.

13.2 Relabel to front

Algorithm 49 relabel-to-front(G, s, t)

```
initialize preflow
initialize node list $L$ containing $V \setminus \{s,t\}$ in any order
foreach $u \in V \setminus \{s,t\}$ do
    $u.current\text{-}neighbour \leftarrow u.neighbour\text{-}list\text{-}head$
$u \leftarrow L.head$
while $u \neq$ null do
    $old\text{-}height \leftarrow \ell(u)$
    discharge($u$)
    if $\ell(u) > old\text{-}height$ then
        move $u$ to the front of $L$
    $u \leftarrow u.next$
```

13.2 Relabel to front

Lemma 74 (Invariant)

In Line 6 of the relabel-to-front algorithm the following invariant holds.

1. *The sequence L is topologically sorted w.r.t. the set of admissable edges; this means for an admissable edge (x, y) the node x appears before y in sequence L.*
2. *No node before u in the list L is active.*

Proof:

- Initialization:
  1. In the beginning s has label $n \geq 2$, and all other nodes have label 0. Hence, no edge is admissable, which means that any ordering L is permitted.
  2. We start with u being the head of the list; hence no node before u can be active
- Maintenance:
  1. - Pushes do no create any new admissable edges. Therefore, not relabeling u leaves L topologically sorted.
     - After relabeling, u cannot have admissable incoming edges as such an edge (x, u) would have had a difference $\ell(x) - \ell(u) \geq 2$ before the re-labeling (such edges do not exist in the residual graph).
       Hence, moving u to the front does not violate the sorting property for any edge; however it fixes this property for all admissable edges leaving u that were generated by the relabeling.

13.2 Relabel to front

Proof:

- Maintenance:

  2. If we do a relabel there is nothing to prove because the only node before u' (u in the next iteration) will be the current u; the discharge(u) operation only terminates when u is not active anymore.

     For the case that we do a relabel, observe that the only way a predecessor could be active is that we push flow to it via an admissable arc. However, all admissable arc point to successors of u.

Note that the invariant for $u = \text{null}$ means that we have a preflow with a valid labelling that does not have active nodes. This means we have a maximum flow.

13.2 Relabel to front

Lemma 75
There are at most $\mathcal{O}(n^3)$ calls to discharge(u).

Every discharge operation without a relabel advances u (the current node within list L). Hence, if we have n discharge operations without a relabel we have $u = \mathrm{null}$ and the algorithm terminates.

Therefore, the number of calls to discharge is at most $n(\#\mathit{relabels} + 1) = \mathcal{O}(n^3)$.

13.2 Relabel to front

Lemma 76
The cost for all relabel-operations is only $\mathcal{O}(n^2)$.

A relabel-operation at a node is constant time (increasing the label and resetting $u.current\text{-}neighbour$). In total we have $\mathcal{O}(n^2)$ relabel-operations.

13.2 Relabel to front

Note that by definition a saturing push operation $(\min\{c_f(e), f(u)\} = c_f(e))$ can at the same time be a non-saturating push operation $(\min\{c_f(e), f(u)\} = f(u))$.

Lemma 77

The cost for all saturating push-operations that are ***not*** *also non-saturating push-operations is only $\mathcal{O}(mn)$.*

Note that such a push-operation leaves the node u active but makes the edge e disappear from the residual graph. Therefore the push-operation is immediately followed by an increase of the pointer $u.\mathit{current\text{-}neighbour}$.

This pointer can traverse the neighbour-list at most $\mathcal{O}(n)$ times (upper bound on number of relabels) and the neighbour-list has only $\mathit{degree}(u) + 1$ many entries ($+1$ for null-entry).

13.2 Relabel to front

Lemma 78

The cost for all non-saturating push-operations is only $\mathcal{O}(n^3)$.

A non-saturating push-operation takes constant time and ends the current call to discharge(). Hence, there are only $\mathcal{O}(n^3)$ such operations.

Theorem 79

The push-relabel algorithm with the rule relabel-to-front takes time $\mathcal{O}(n^3)$.

13.3 Highest label

Algorithm 50 highest-label(G, s, t)

```
initialize preflow
foreach $u \in V \setminus \{s,t\}$ do
    $u.current\text{-}neighbour \leftarrow u.neighbour\text{-}list\text{-}head$
while $\exists$ active node $u$ do
    select active node $u$ with highest label
    discharge$(u)$
```

13.3 Highest label

Lemma 80

When using highest label the number of non-saturating pushes is only $\mathcal{O}(n^3)$.

After a non-saturating push from u a relabel is required to make a currently non-active node x, with $\ell(x) \geq \ell(u)$ active again (note that this includes u).

Hence, after n non-saturating pushes without an intermediate relabel there are no active nodes left.

Therefore, the number of non-saturating pushes is at most $n(\#relabels + 1) = \mathcal{O}(n^3)$.

13.3 Highest label

Since a discharge-operation is terminated by a non-saturating push this gives an upper bound of $\mathcal{O}(n^3)$ on the number of discharge-operations.

The cost for relabels and saturating pushes can be estimated in exactly the same way as in the case of relabel-to-front.

Question:

How do we find the next node for a discharge operation?

13.3 Highest label

Maintain lists L_i, $i \in \{0, \dots, 2n\}$, where list L_i contains active nodes with label i (maintaining these lists induces only constant additional cost for every push-operation and for every relabel-operation).

After a discharge operation terminated for a node u with label k, traverse the lists $k-1, \dots, 0$, (in that order) until you find a non-empty list.

Unless the last (non-saturating) push was to s or t the list $k-1$ must be non-empty (i.e., the search takes constant time).

13.3 Highest label

Hence, the total time required for searching for active nodes is at most

$$\mathcal{O}(n^3) + n(\#\textit{non-saturating-pushes-to-s-or-t})$$

Lemma 81

The number of non-saturating pushes to s or t is at most $\mathcal{O}(n^2)$.

With this lemma we get

Theorem 82

The push-relabel algorithm with the rule highest-label takes time $\mathcal{O}(n^3)$.

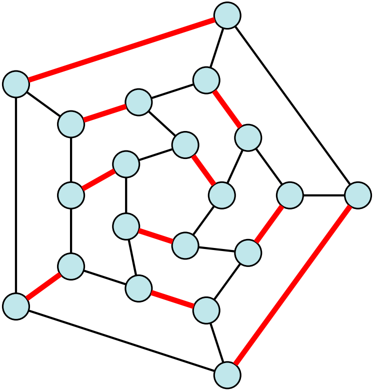
13.3 Highest label

Proof of the Lemma.

- We only show that the number of pushes to the source is at most $\mathcal{O}(n^2)$. A similar argument holds for the target.
- After a node v (which must have $\ell(v) = n + 1$) made a non-saturating push to the source there needs to be another node whose label is increased from $\leq n + 1$ to $n + 2$ before v can become active again.
- This happens for every push that v makes to the source. Since, every node can pass the threshold $n + 2$ at most once, v can make at most n pushes to the source.
- As this holds for every node the total number of pushes to the source is at most $\mathcal{O}(n^2)$.

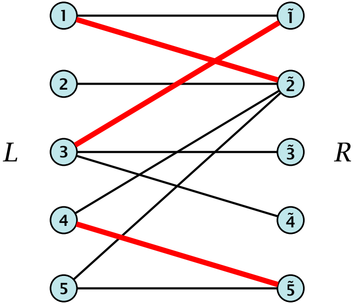
Matching

- Input: undirected graph $G = (V, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



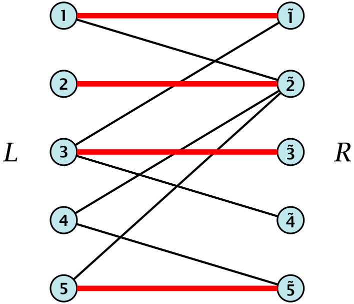
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



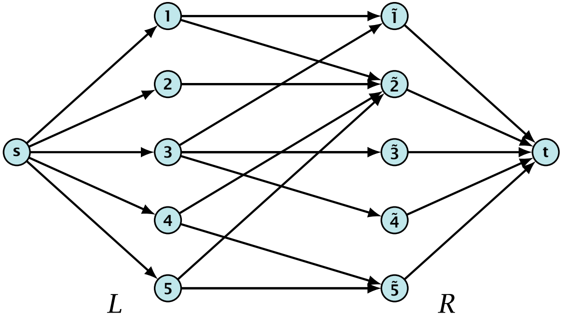
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



Maxflow Formulation

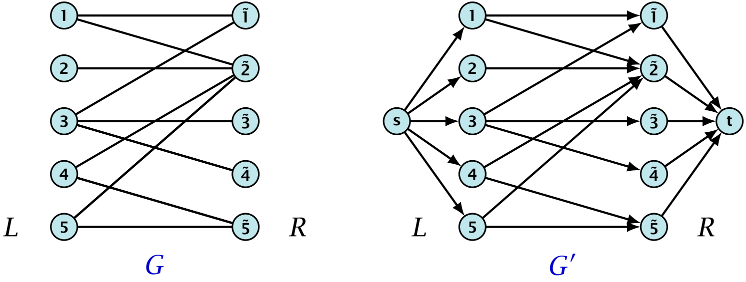
- Input: undirected, bipartite graph $G = (L \uplus R \uplus \{s,t\}, E')$.
- Direct all edges from L to R.
- Add source s and connect it to all nodes on the left.
- Add t and connect all nodes on the right to t.
- All edges have unit capacity.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

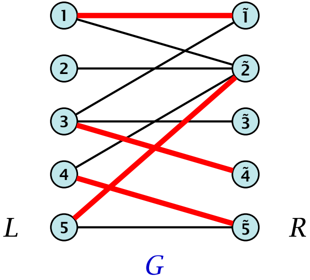
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.

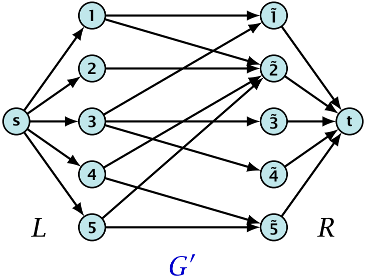


Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.

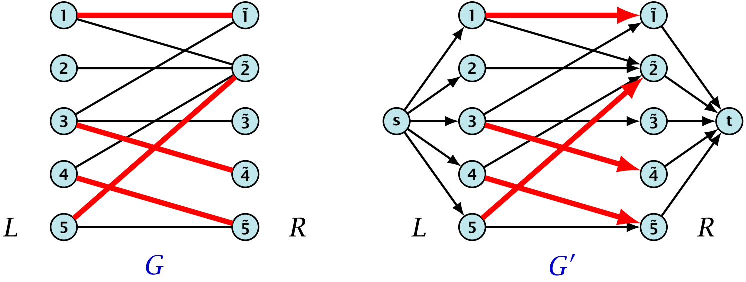




Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

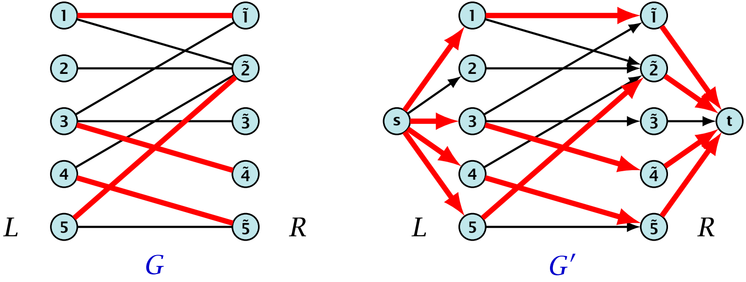
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

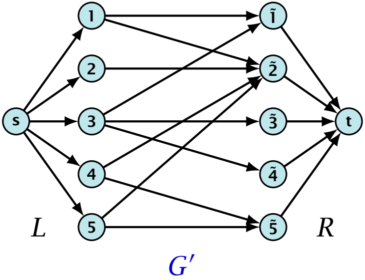
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.

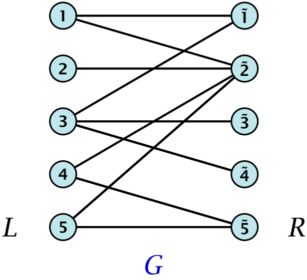


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

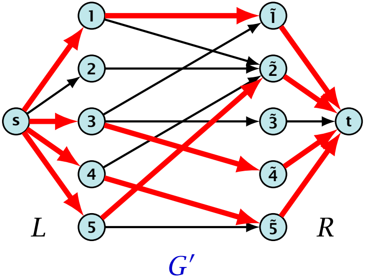


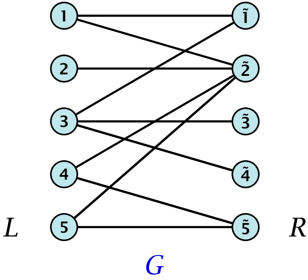


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

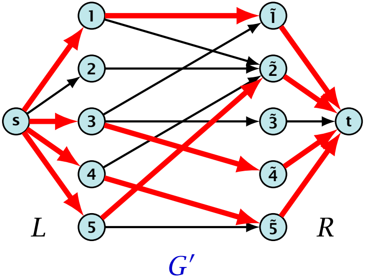


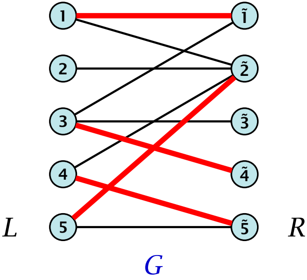


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

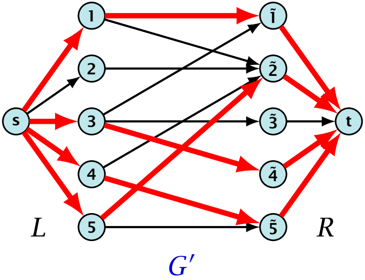


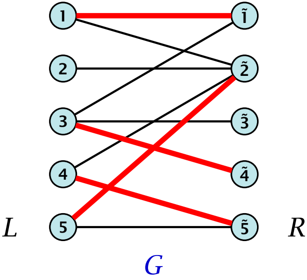


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.





14.1 Matching

Which flow algorithm to use?

- Generic augmenting path: $\mathcal{O}(m\,\mathrm{val}(f^*)) = \mathcal{O}(mn)$.
- Capacity scaling: $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$.

Baseball Elimination

team i	wins w_i	losses ℓ_i	remaining games Atl	 Phi	 NY	 Mon
Atlanta	83	71	-	1	6	1
Philadelphia	80	79	1	-	0	2
New York	78	78	6	0	-	0
Montreal	77	82	1	2	0	-

Which team can end the season with most wins?

- Montreal is eliminated, since even after winning all remaining games there are only 80 wins.
- But also Philadelphia is eliminated. Why?

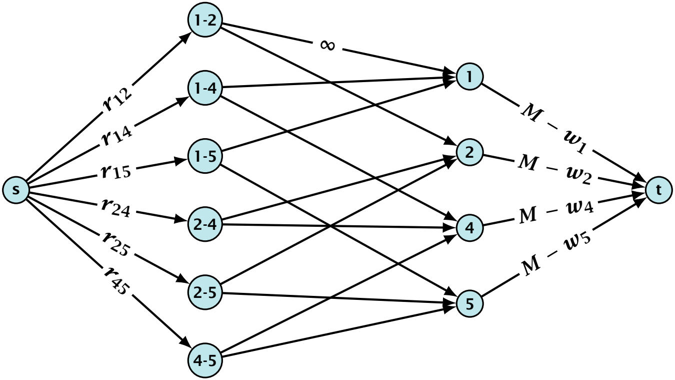
Baseball Elimination

Formal definition of the problem:

- Given a set S of teams, and one specific team $z \in S$.
- Team x has already won w_x games.
- Team x still has to play team y, r_{xy} times.
- Does team z still have a chance to finish with the most number of wins.

Baseball Elimination

Flow networks for $z = 3$. M is number of wins Team 3 can still obtain.



Idea. Distribute the results of remaining games in such a way that no team gets too many wins.

Certificate of Elimination

Let $T \subseteq S$ be a subset of teams. Define

$$w(T) := \sum_{i \in T} w_i, \qquad r(T) := \sum_{i,j \in T, i<j} r_{ij}$$

wins of teams in T

remaining games among teams in T

If $\frac{w(T)+r(T)}{|T|} > M$ then one of the teams in T will have more than M wins in the end. A team that can win at most M games is therefore eliminated.

Theorem 83

A team z is eliminated if and only if the flow network for z does not allow a flow of value $\sum_{ij\in S\setminus\{z\},i<j} r_{ij}$.

Proof (⇐)

- Consider the mincut A in the flow network. Let T be the set of team-nodes in A.
- If for a node x-y not both team nodes x and y are in T, then x-$y \notin A$ as otw. the cut would cut an infinite capacity edge.
- We don't find a flow that saturates all source edges:

$$\begin{aligned} r(S\setminus\{z\}) &> \mathrm{cap}(S, V\setminus S) \\ &\geq \sum_{i<j:\, i\notin T \vee j\notin T} r_{ij} + \sum_{i\in T}(M - w_i) \\ &\geq r(S\setminus\{z\}) - r(T) + |T|M - w(T) \end{aligned}$$

- This gives $M < (w(T) + r(T))/|T|$, i.e., z is eliminated.

Baseball Elimination

Proof (⇒)

- Suppose we have a flow that saturates all source edges.
- We can assume that this flow is integral.
- For every pairing x-y it defines how many games team x and team y should win.
- The flow leaving the team-node x can be interpreted as the additional number of wins that team x will obtain.
- This is less than $M - w_x$ because of capacity constraints.
- Hence, we found a set of results for the remaining games, such that no team obtains more than M wins in total.
- Hence, team z is not eliminated.

Project Selection

Project selection problem:

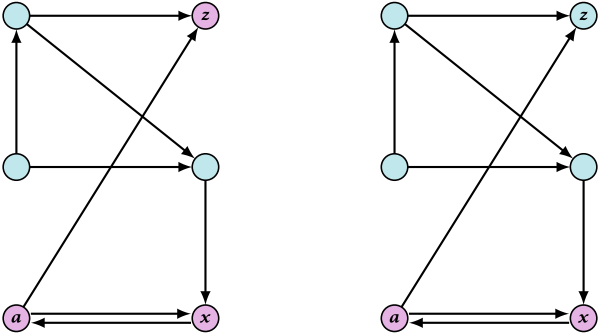
- Set P of possible projects. Project v has an associated profit p_v (can be positive or negative).
- Some projects have requirements (taking course EA2 requires course EA1).
- Dependencies are modelled in a graph. Edge (u,v) means "can't do project u without also doing project v."
- A subset A of projects is feasible if the prerequisites of every project in A also belong to A.

Goal: Find a feasible set of projects that maximizes the profit.

Project Selection

The prerequisite graph:

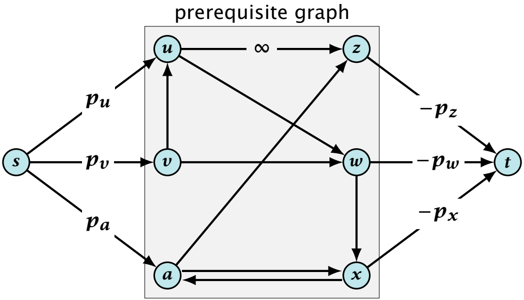
- $\{x, a, z\}$ is a feasible subset.
- $\{x, a\}$ is infeasible.



Project Selection

Mincut formulation:

- Edges in the prerequisite graph get infinite capacity.
- Add edge (s, v) with capacity p_v for nodes v with positive profit.
- Create edge (v, t) with capacity $-p_v$ for nodes v with negative profit.

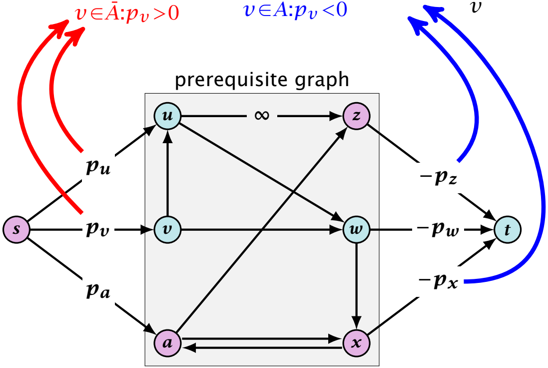


Theorem 84

A is a mincut if $A \setminus \{s\}$ is the optimal set of projects.

Proof.

- A is feasible because of capacity infinity edges.
- $\mathrm{cap}(A, V \setminus A) = \sum_{v \in \bar{A}: p_v > 0} p_v + \sum_{v \in A: p_v < 0} (-p_v) = \sum_{v} p_v - \sum_{v \in A} p_v$



Mincost Flow

Consider the following problem:

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \quad 0 \le f(e) \le u(e) \\ & \forall v \in V: \quad f(v) = b(v) \end{aligned}$$

- $G = (V, E)$ is an oriented graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

Mincost Flow

Consider the following problem:

$$\begin{array}{ll} \text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E:\ \ 0 \le f(e) \le u(e) \\ & \forall v \in V:\ \ f(v) = b(v) \end{array}$$

- $G = (V, E)$ is an oriented graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

Mincost Flow

Consider the following problem:

$$\begin{array}{lll} \text{min} & \sum_e c(e)f(e) & \\ \text{s.t.} & \forall e \in E: & 0 \le f(e) \le u(e) \\ & \forall v \in V: & f(v) = b(v) \end{array}$$

- $G = (V, E)$ is an oriented graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

Mincost Flow

Consider the following problem:

$$\begin{array}{lll} \text{min} & \sum_e c(e)f(e) & \\ \text{s.t.} & \forall e \in E: & 0 \le f(e) \le u(e) \\ & \forall v \in V: & f(v) = b(v) \end{array}$$

- $G = (V, E)$ is an oriented graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

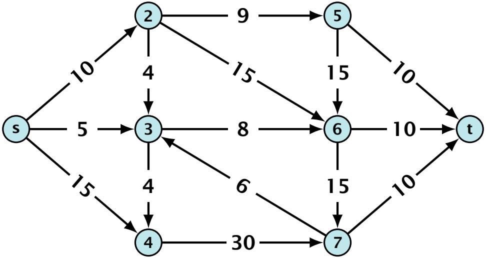
Mincost Flow

Consider the following problem:

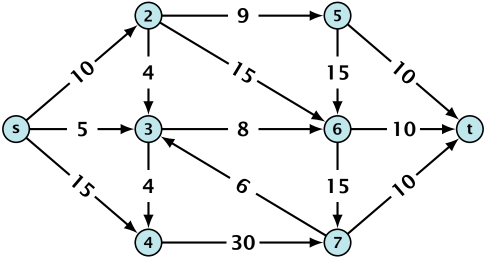
$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \quad 0 \le f(e) \le u(e) \\ & \forall v \in V: \quad f(v) = b(v) \end{aligned}$$

- $G = (V, E)$ is an oriented graph.
- $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$ is the capacity function.
- $c : E \to \mathbb{R}$ is the cost function (note that $c(e)$ may be negative).
- $b : V \to \mathbb{R}$, $\sum_{v \in V} b(v) = 0$ is a demand function.

Solve Maxflow Using Mincost Flow

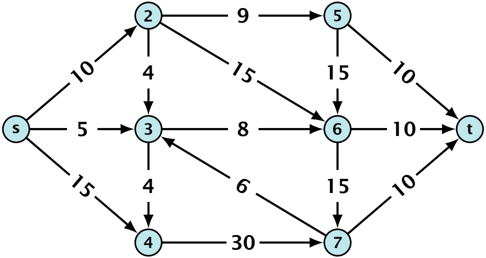


Solve Maxflow Using Mincost Flow



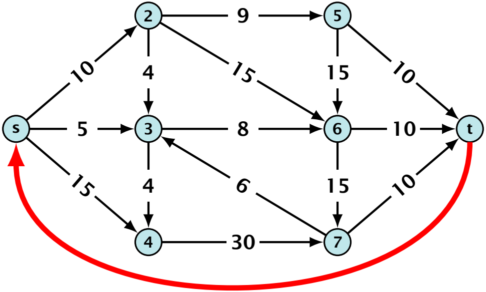
- Given a flow network for a standard maxflow problem.

Solve Maxflow Using Mincost Flow



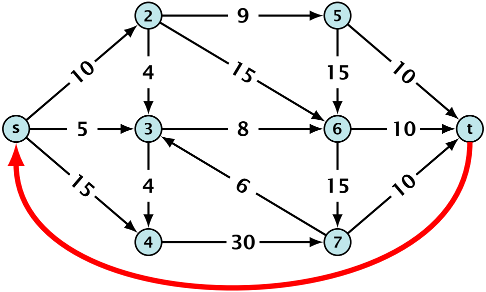
- Given a flow network for a standard maxflow problem.
- Set $b(v) = 0$ for every node. Keep the capacity function u for all edges. Set the cost $c(e)$ for every edge to 0.

Solve Maxflow Using Mincost Flow



- Given a flow network for a standard maxflow problem.
- Set $b(v) = 0$ for every node. Keep the capacity function u for all edges. Set the cost $c(e)$ for every edge to 0.
- Add an edge from t to s with infinite capacity and cost -1.

Solve Maxflow Using Mincost Flow



- Given a flow network for a standard maxflow problem.
- Set $b(v) = 0$ for every node. Keep the capacity function u for all edges. Set the cost $c(e)$ for every edge to 0.
- Add an edge from t to s with infinite capacity and cost -1.
- Then, $\mathrm{val}(f^*) = -\mathrm{cost}(f_{\min})$, where f^* is a maxflow, and $f_{\min}$ is a mincost-flow.

Solve Maxflow Using Mincost Flow

Solve decision version of maxflow:

- Given a flow network for a standard maxflow problem, and a value k.
- Set $b(v) = 0$ for every node apart from s or t. Set $b(s) = -k$ and $b(t) = k$.
- Set edge-costs to zero, and keep the capacities.
- There exists a maxflow of value k if and only if the mincost-flow problem is feasible.

Solve Maxflow Using Mincost Flow

Solve decision version of maxflow:

- Given a flow network for a standard maxflow problem, and a value k.
- Set $b(v) = 0$ for every node apart from s or t. Set $b(s) = -k$ and $b(t) = k$.
- Set edge-costs to zero, and keep the capacities.
- There exists a maxflow of value k if and only if the mincost-flow problem is feasible.

Solve Maxflow Using Mincost Flow

Solve decision version of maxflow:

- Given a flow network for a standard maxflow problem, and a value k.
- Set $b(v) = 0$ for every node apart from s or t. Set $b(s) = -k$ and $b(t) = k$.
- Set edge-costs to zero, and keep the capacities.
- There exists a maxflow of value k if and only if the mincost-flow problem is feasible.

Solve Maxflow Using Mincost Flow

Solve decision version of maxflow:

- Given a flow network for a standard maxflow problem, and a value k.
- Set $b(v) = 0$ for every node apart from s or t. Set $b(s) = -k$ and $b(t) = k$.
- Set edge-costs to zero, and keep the capacities.
- There exists a maxflow of value k if and only if the mincost-flow problem is feasible.

Generalization

Our model:

$$\begin{array}{ll}\text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E: \;\; 0 \le f(e) \le u(e) \\ & \forall v \in V: \;\; f(v) = b(v)\end{array}$$

where $b : V \to \mathbb{R}$, $\sum_v b(v) = 0$; $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$; $c : E \to \mathbb{R}$;

A more general model?

$$\begin{array}{ll}\text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v)\end{array}$$

where $a : V \to \mathbb{R}$, $b : V \to \mathbb{R}$; $\ell : E \to \mathbb{R} \cup \{-\infty\}$, $u : E \to \mathbb{R} \cup \{\infty\}$
$c : E \to \mathbb{R}$;

Generalization

Our model:

$$\begin{array}{ll} \text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E:\ \ 0 \leq f(e) \leq u(e) \\ & \forall v \in V:\ \ f(v) = b(v) \end{array}$$

where $b : V \to \mathbb{R}$, $\sum_v b(v) = 0$; $u : E \to \mathbb{R}_0^+ \cup \{\infty\}$; $c : E \to \mathbb{R}$;

A more general model?

$$\begin{array}{ll} \text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E:\ \ \ell(e) \leq f(e) \leq u(e) \\ & \forall v \in V:\ \ a(v) \leq f(v) \leq b(v) \end{array}$$

where $a : V \to \mathbb{R}$, $b : V \to \mathbb{R}$; $\ell : E \to \mathbb{R} \cup \{-\infty\}$, $u : E \to \mathbb{R} \cup \{\infty\}$ $c : E \to \mathbb{R}$;

Reduction I

$$\begin{array}{ll} \text{min} & \sum_e c(e)f(e) \\ \text{s.t.} & \forall e \in E: \ \ \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \ \ a(v) \le f(v) \le b(v) \end{array}$$

We can assume that $a(v) = b(v)$:

Add new node r.

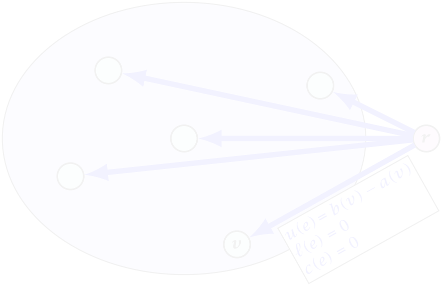
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v\in V} b(v)$.



Reduction I

$$\begin{aligned} \min \quad & \textstyle\sum_e c(e) f(e) \\ \text{s.t.} \quad & \forall e \in E:\ \ \ell(e) \le f(e) \le u(e) \\ & \forall v \in V:\ \ a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

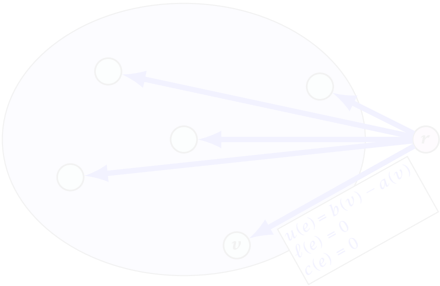
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.



Reduction I

$$\begin{aligned}\text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v)\end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

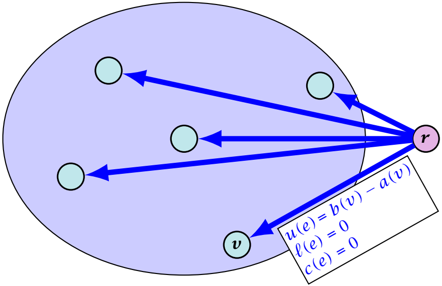
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.



Reduction I

$$\begin{aligned} \min \quad & \textstyle\sum_e c(e) f(e) \\ \text{s.t.} \quad & \forall e \in E: \; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

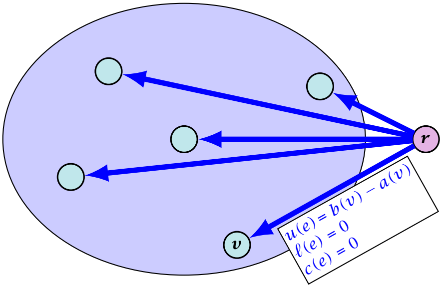
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.



Reduction I

$$\begin{aligned} \min \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

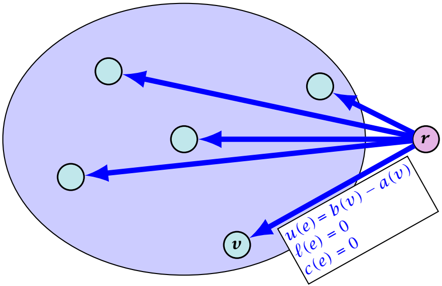
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.



Reduction I

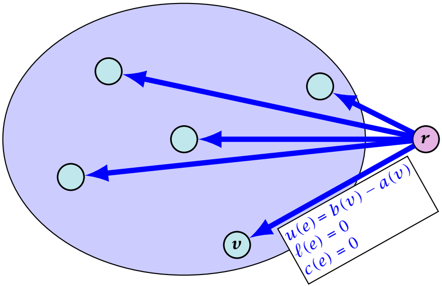
$$\begin{aligned} \min \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.



Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.

Reduction I

$$\begin{array}{ll} \text{min} & \sum_e c(e) f(e) \\ \text{s.t.} & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{array}$$

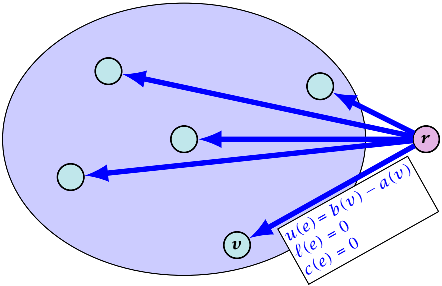
We can assume that $a(v) = b(v)$:

Add new node r.

Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).



Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.

Reduction I

$$\begin{aligned} \min \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

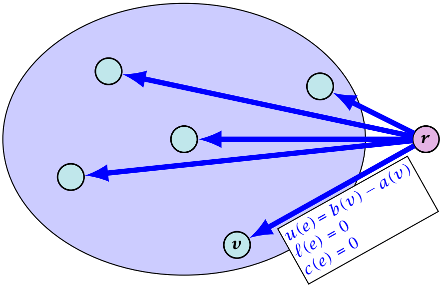
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

Set $b(r) = \sum_{v \in V} b(v)$.



Reduction I

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e)f(e) \\ \text{s.t.} \quad & \forall e \in E: \;\; \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \;\; a(v) \le f(v) \le b(v) \end{aligned}$$

We can assume that $a(v) = b(v)$:

Add new node r.

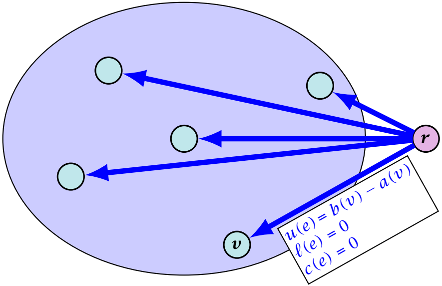
Add edge (r, v) for all $v \in V$.

Set $\ell(e) = c(e) = 0$ for these edges.

Set $u(e) = b(v) - a(v)$ for edge (r, v).

Set $a(v) = b(v)$ for all $v \in V$.

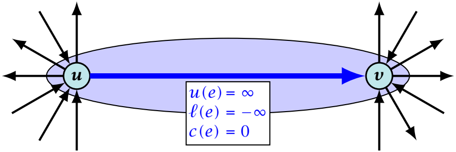
Set $b(r) = \sum_{v \in V} b(v)$.



Reduction II

$$\begin{array}{lll} \text{min} & \sum_e c(e)f(e) & \\ \text{s.t.} & \forall e \in E: & \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: & f(v) = b(v) \end{array}$$

We can assume that either $\ell(e) \neq -\infty$ or $u(e) \neq \infty$:

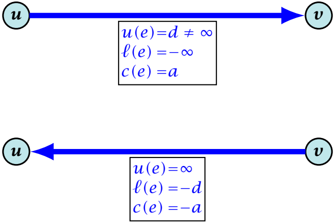


If $c(e) = 0$ we can simply contract the edge/identify nodes u and v

Reduction III

$$\begin{array}{lll} \text{min} & \sum_e c(e)f(e) & \\ \text{s.t.} & \forall e \in E: & \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: & f(v) = b(v) \end{array}$$

We can assume that $\ell(e) \neq -\infty$:

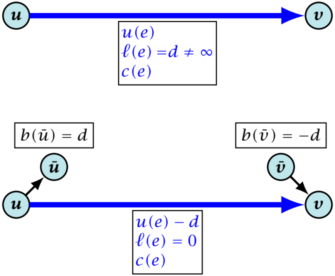


Replace the edge by an edge in opposite direction.

Reduction IV

$$\begin{aligned} \text{min} \quad & \textstyle\sum_e c(e) f(e) \\ \text{s.t.} \quad & \forall e \in E: \quad \ell(e) \le f(e) \le u(e) \\ & \forall v \in V: \quad f(v) = b(v) \end{aligned}$$

We can assume that $\ell(e) = 0$:



The added edges have infinite capacity and cost $c(e)/2$.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Applications

Caterer Problem

- She needs to supply r_i napkins on N successive days.
- She can buy new napkins at p cents each.
- She can launder them at a fast laundry that takes m days and cost f cents a napkin.
- She can use a slow laundry that takes $k > m$ days and costs s cents each.
- At the end of each day she should determine how many to send to each laundry and how many to buy in order to fulfill demand.
- Minimize cost.

Residual Graph

The residual graph for a mincost flow is exactly defined as the residual graph for standard flows, with the only exception that one needs to define a cost for the residual edge.

For a flow of z from u to v the residual edge (v,u) has capacity z and a cost of $-c((u,v))$.

Residual Graph

The residual graph for a mincost flow is exactly defined as the residual graph for standard flows, with the only exception that one needs to define a cost for the residual edge.

For a flow of z from u to v the residual edge (v, u) has capacity z and a cost of $-c((u, v))$.

15 Mincost Flow

A circulation in a graph $G = (V, E)$ is a function $f : E \to \mathbb{R}^+$ that has an excess flow $f(v) = 0$ for every node $v \in V$ (G may be a directed graph instead of just an oriented graph).

A circulation is feasible if it fulfills capacity constraints, i.e., $f(e) \le u(e)$ for every edge of G.

15 Mincost Flow

A circulation in a graph $G = (V, E)$ is a function $f : E \to \mathbb{R}^+$ that has an excess flow $f(v) = 0$ for every node $v \in V$ (G may be a directed graph instead of just an oriented graph).

A circulation is feasible if it fulfills capacity constraints, i.e., $f(e) \leq u(e)$ for every edge of G.

15 Mincost Flow

Lemma 85

A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

$\Rightarrow$ Suppose that g is a feasible circulation of negative cost in the residual graph.

$\Leftarrow$ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

15 Mincost Flow

Lemma 85

A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

⇒ Suppose that g is a feasible circulation of negative cost in the residual graph.

Then $f + g$ is a feasible flow with cost $\mathrm{cost}(f) + \mathrm{cost}(g) < \mathrm{cost}(f)$. Hence, f is not minimum cost.

⇐ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

Clearly $f^* - f$ is a circulation of negative cost. One can also easily see that it is feasible for the residual graph.

15 Mincost Flow

Lemma 85

A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

- ⇒ Suppose that g is a feasible circulation of negative cost in the residual graph.

  Then $f + g$ is a feasible flow with cost $\mathrm{cost}(f) + \mathrm{cost}(g) < \mathrm{cost}(f)$. Hence, f is not minimum cost.

- ⇐ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

  Clearly $f^* - f$ is a circulation of negative cost. One can also easily see that it is feasible for the residual graph.

15 Mincost Flow

Lemma 85

A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

⇒ Suppose that g is a feasible circulation of negative cost in the residual graph.

Then $f + g$ is a feasible flow with cost $\text{cost}(f) + \text{cost}(g) < \text{cost}(f)$. Hence, f is not minimum cost.

⇐ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

Clearly $f^* - f$ is a circulation of negative cost. One can also easily see that it is feasible for the residual graph.

15 Mincost Flow

Lemma 85

A given flow is a mincost-flow if and only if the corresponding residual graph G_f does not have a feasible circulation of negative cost.

- ⇒ Suppose that g is a feasible circulation of negative cost in the residual graph.

  Then $f + g$ is a feasible flow with cost $\text{cost}(f) + \text{cost}(g) < \text{cost}(f)$. Hence, f is not minimum cost.

- ⇐ Let f be a non-mincost flow, and let f^* be a min-cost flow. We need to show that the residual graph has a feasible circulation with negative cost.

  Clearly $f^* - f$ is a circulation of negative cost. One can also easily see that it is feasible for the residual graph.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$*.*

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

Lemma 86

A graph (without zero-capacity edges) has a feasible circulation of negative cost if and only if it has a negative cycle w.r.t. edge-weights $c : E \to \mathbb{R}$.

Proof.

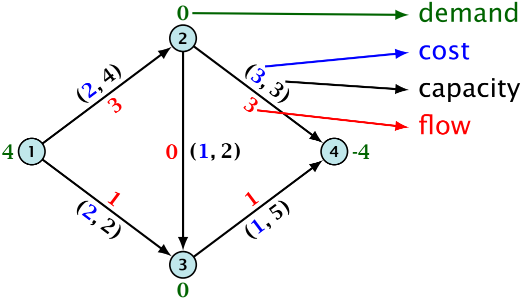
- Suppose that we have a negative cost circulation.
- Find directed path only using edges that have non-zero flow.
- If this path has negative cost you are done.
- Otherwise send flow in opposite direction along the cycle until the bottleneck edge(s) does not carry any flow.
- You still have a circulation with negative cost.
- Repeat.

15 Mincost Flow

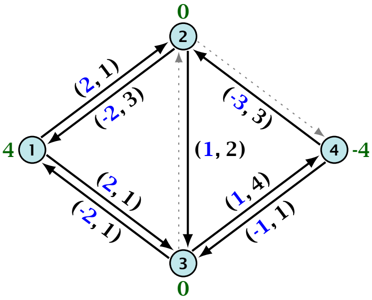
Algorithm 51 CycleCanceling$(G = (V, E), c, u, b)$

```
establish a feasible flow $f$ in $G$
while $G_f$ contains negative cycle do
    use Bellman-Ford to find a negative circuit $Z$
    $\delta \leftarrow \min\{u_f(e) \mid e \in Z\}$
    augment $\delta$ units along $Z$ and update $G_f$
```

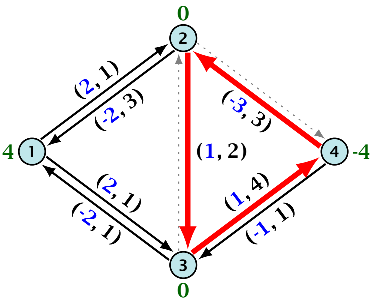
15 Mincost Flow



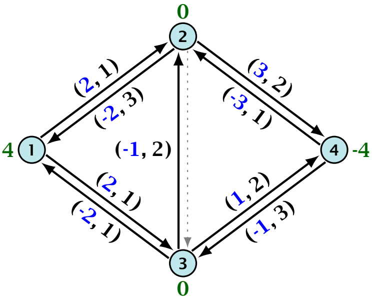
15 Mincost Flow



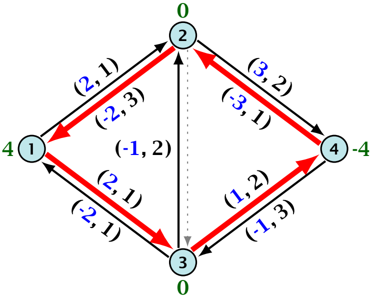
15 Mincost Flow



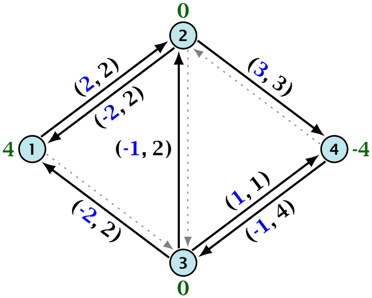
15 Mincost Flow



15 Mincost Flow



15 Mincost Flow



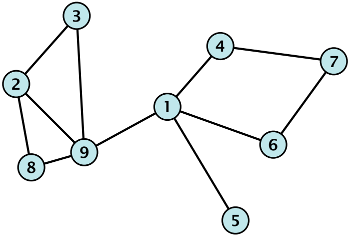
15 Mincost Flow

Lemma 87

The improving cycle algorithm runs in time $\mathcal{O}(nm^2CU)$, for integer capacities and costs, when for all edges e, $|c(e)| \le C$ and $|u(e)| \le U$.

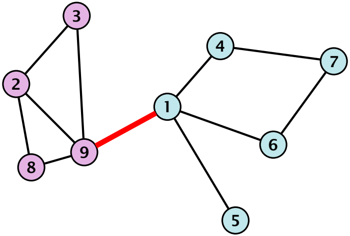
16 Global Mincut

Given an undirected, capacitated graph $G = (V, E, c)$ find a partition of V into two non-empty sets $S, V \setminus S$ s.t. the capacity of edges between both sets is minimized.



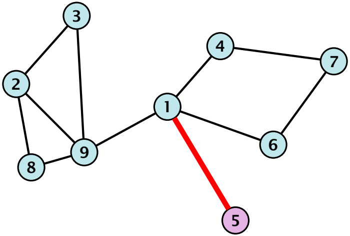
16 Global Mincut

Given an undirected, capacitated graph $G = (V, E, c)$ find a partition of V into two non-empty sets $S, V \setminus S$ s.t. the capacity of edges between both sets is minimized.



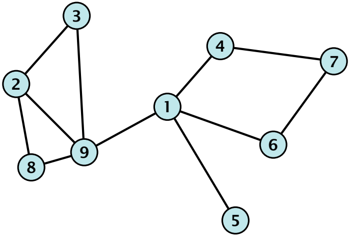
16 Global Mincut

Given an undirected, capacitated graph $G = (V, E, c)$ find a partition of V into two non-empty sets $S, V \setminus S$ s.t. the capacity of edges between both sets is minimized.



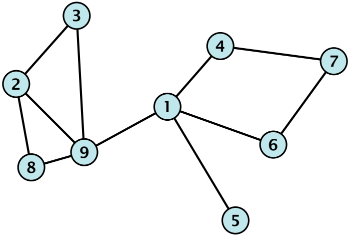
16 Global Mincut

Given an undirected, capacitated graph $G = (V, E, c)$ find a partition of V into two non-empty sets $S, V \setminus S$ s.t. the capacity of edges between both sets is minimized.



16 Global Mincut

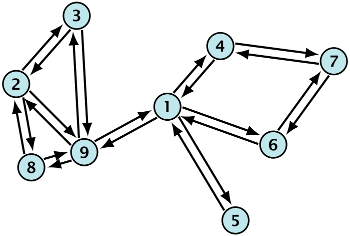
We can solve this problem using standard maxflow/mincut.



16 Global Mincut

We can solve this problem using standard maxflow/mincut.

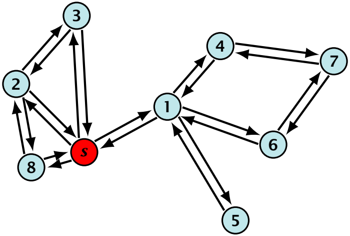
- Construct a directed graph $G' = (V, E')$ that has edges (u, v) and (v, u) for every edge $\{u, v\} \in E$.



16 Global Mincut

We can solve this problem using standard maxflow/mincut.

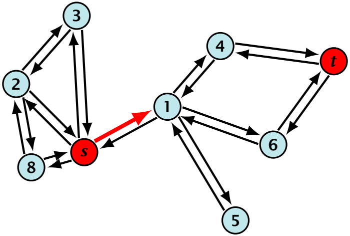
- Construct a directed graph $G' = (V, E')$ that has edges (u, v) and (v, u) for every edge $\{u, v\} \in E$.
- Fix an arbitrary node $s \in V$ as source. Compute a minimum s-t cut for all possible choices $t \in V, t \neq s$. (Time: $\mathcal{O}(n^4)$)



16 Global Mincut

We can solve this problem using standard maxflow/mincut.

- Construct a directed graph $G' = (V, E')$ that has edges (u, v) and (v, u) for every edge $\{u, v\} \in E$.
- Fix an arbitrary node $s \in V$ as source. Compute a minimum s-t cut for all possible choices $t \in V, t \neq s$. (Time: $\mathcal{O}(n^4)$)
- Let $(S, V \setminus S)$ be a minimum global mincut. The above algorithm will output a cut of capacity $\text{cap}(S, V \setminus S)$ whenever $|\{s, t\} \cap S| = 1$.



Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by “identifying” u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

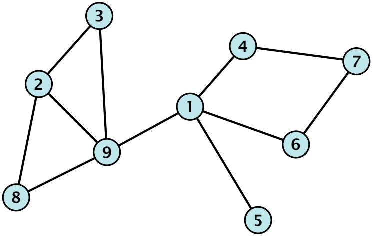
Example 88

- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

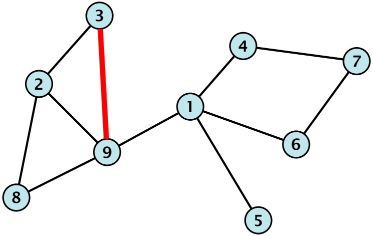


- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

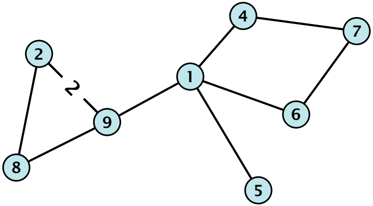


- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88

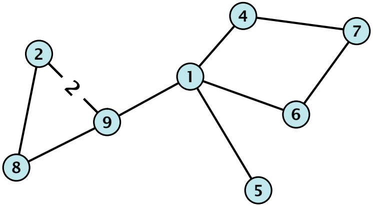


- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

- Given a graph $G = (V, E)$ and an edge $e = \{u, v\}$.
- The graph G/e is obtained by "identifying" u and v to form a new node.
- Resulting parallel edges are replaced by a single edge, whose capacity equals the sum of capacities of the parallel edges.

Example 88



- Edge-contractions do no decrease the size of the mincut.

Edge Contractions

We can perform an edge-contraction in time $\mathcal{O}(n)$.

Randomized Mincut Algorithm

Algorithm 52 KargerMincut($G = (V, E, c)$)

```
for $i = 1 \to n - 2$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
return only cut in $G$
```

Randomized Mincut Algorithm

Algorithm 52 KargerMincut$(G = (V, E, c))$

```
for $i = 1 \to n - 2$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
return only cut in $G$
```

- Let G_t denote the graph after the $(n - t)$-th iteration, when t nodes are left.

Randomized Mincut Algorithm

Algorithm 52 KargerMincut($G = (V, E, c)$)

```
for $i = 1 \to n - 2$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
return only cut in $G$
```

- Let G_t denote the graph after the $(n-t)$-th iteration, when t nodes are left.
- Note that the final graph G_2 only contains a single edge.

Randomized Mincut Algorithm

```
Algorithm 52 KargerMincut($G = (V, E, c)$)
for $i = 1 \to n - 2$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
return only cut in $G$
```

- Let G_t denote the graph after the $(n - t)$-th iteration, when t nodes are left.
- Note that the final graph G_2 only contains a single edge.
- The cut in G_2 corresponds to a cut in the original graph G with the same capacity.

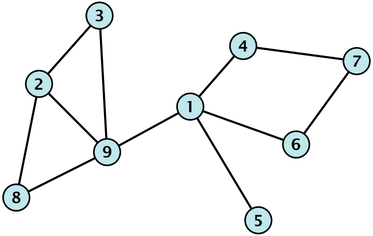
Randomized Mincut Algorithm

Algorithm 52 KargerMincut($G = (V, E, c)$)

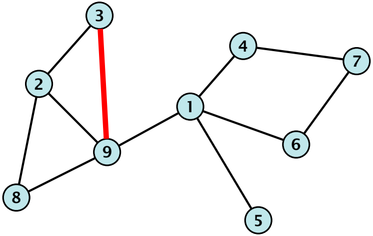
```
for $i = 1 \to n - 2$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
return only cut in $G$
```

- Let G_t denote the graph after the $(n - t)$-th iteration, when t nodes are left.
- Note that the final graph G_2 only contains a single edge.
- The cut in G_2 corresponds to a cut in the original graph G with the same capacity.
- What is the probability that this algorithm returns a mincut?

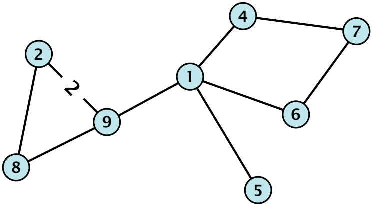
Example: Randomized Mincut Algorithm



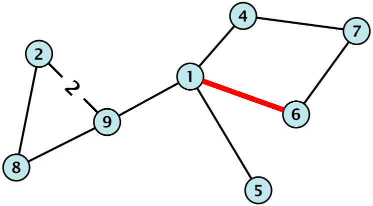
Example: Randomized Mincut Algorithm



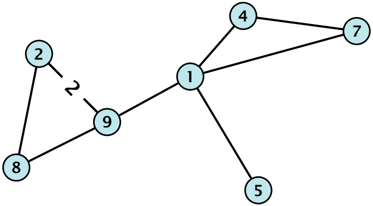
Example: Randomized Mincut Algorithm



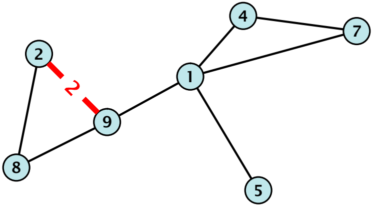
Example: Randomized Mincut Algorithm



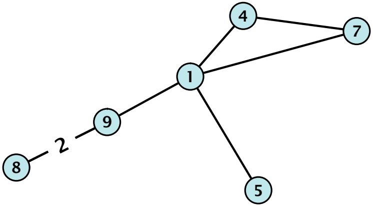
Example: Randomized Mincut Algorithm



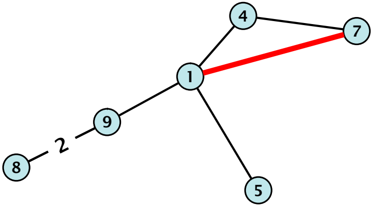
Example: Randomized Mincut Algorithm



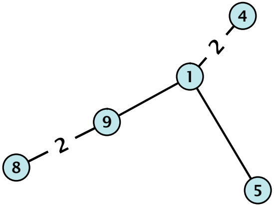
Example: Randomized Mincut Algorithm



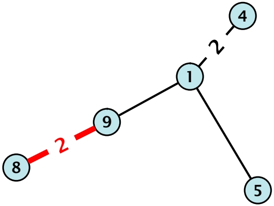
Example: Randomized Mincut Algorithm



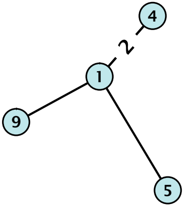
Example: Randomized Mincut Algorithm



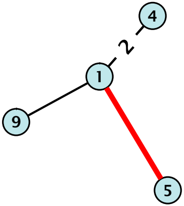
Example: Randomized Mincut Algorithm



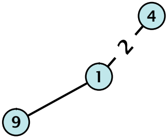
Example: Randomized Mincut Algorithm



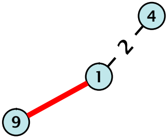
Example: Randomized Mincut Algorithm



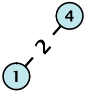
Example: Randomized Mincut Algorithm



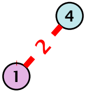
Example: Randomized Mincut Algorithm



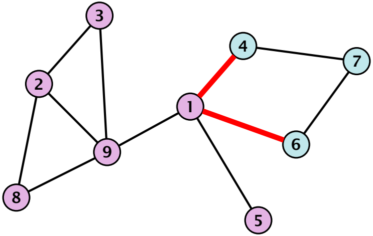
Example: Randomized Mincut Algorithm



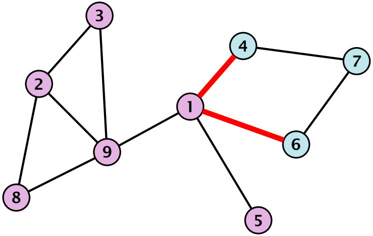
Example: Randomized Mincut Algorithm



Example: Randomized Mincut Algorithm



Example: Randomized Mincut Algorithm



What is the probability that this algorithm returns a mincut?

Analysis

What is the probability that a given mincut A is still possible after round i?

- It is still possible to obtain cut A in the end if so far no edge in $(A, V \setminus A)$ has been contracted.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \text{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\text{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\text{cap}(v) \geq \min$.
- Summing $\text{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e \in E} c(e) = \sum_{v \in V} \text{cap}(v) \geq (n-i+1) \cdot \min$$

- Hence, the probability of choosing an edge from the cut is at most $\min / c(E) \leq 2/(n-i+1)$.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \text{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\text{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\text{cap}(v) \geq \min$.
- Summing $\text{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e\in E} c(e) = \sum_{v\in V} \text{cap}(v) \geq (n-i+1)\cdot\min$$

- Hence, the probability of choosing an edge from the cut is at most $\min/c(E) \leq 2/(n-i+1)$.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \mathrm{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\mathrm{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\mathrm{cap}(v) \geq \min$.
- Summing $\mathrm{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e \in E} c(e) = \sum_{v \in V} \mathrm{cap}(v) \geq (n-i+1) \cdot \min$$

- Hence, the probability of choosing an edge from the cut is at most $\min / c(E) \leq 2/(n-i+1)$.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \text{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\text{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\text{cap}(v) \geq \min$.
- Summing $\text{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e \in E} c(e) = \sum_{v \in V} \text{cap}(v) \geq (n - i + 1) \cdot \min$$

- Hence, the probability of choosing an edge from the cut is at most $\min / c(E) \leq 2/(n - i + 1)$.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \text{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\text{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\text{cap}(v) \geq \min$.
- Summing $\text{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e \in E} c(e) = \sum_{v \in V} \text{cap}(v) \geq (n-i+1) \cdot \min$$

- Hence, the probability of choosing an edge from the cut is at most $\min / c(E) \leq 2/(n-i+1)$.

Analysis

What is the probability that we select an edge from A in iteration i?

- Let $\min = \text{cap}(A, V \setminus A)$ denote the capacity of a mincut.
- Let $\text{cap}(v)$ be capacity of edges incident to vertex $v \in V_{n-i+1}$.
- Clearly, $\text{cap}(v) \geq \min$.
- Summing $\text{cap}(v)$ over all edges gives

$$2c(E) = 2\sum_{e\in E} c(e) = \sum_{v\in V} \text{cap}(v) \geq (n-i+1)\cdot \min$$

- Hence, the probability of choosing an edge from the cut is at most $\min / c(E) \leq 2/(n-i+1)$.

Analysis

The probability that we do not choose an edge from the cut in iteration i is

$$1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1} \ .$$

The probability that the cut is alive after iteration $n-t$ (after which t nodes are left) is

$$\prod_{i=1}^{n-t} \frac{n-i-1}{n-i+1} = \frac{t(t-1)}{n(n-1)} \ .$$

Choosing $t = 2$ gives that with probability $1/\binom{n}{2}$ the algorithm computes a mincut.

Analysis

The probability that we do not choose an edge from the cut in iteration i is

$$1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1} \enspace .$$

The probability that the cut is alive after iteration $n-t$ (after which t nodes are left) is

$$\prod_{i=1}^{n-t} \frac{n-i-1}{n-i+1} = \frac{t(t-1)}{n(n-1)} \enspace .$$

Choosing $t = 2$ gives that with probability $1/\binom{n}{2}$ the algorithm computes a mincut.

Analysis

The probability that we do not choose an edge from the cut in iteration i is

$$1 - \frac{2}{n-i+1} = \frac{n-i-1}{n-i+1} \ .$$

The probability that the cut is alive after iteration $n-t$ (after which t nodes are left) is

$$\prod_{i=1}^{n-t} \frac{n-i-1}{n-i+1} = \frac{t(t-1)}{n(n-1)} \ .$$

Choosing $t = 2$ gives that with probability $1/\binom{n}{2}$ the algorithm computes a mincut.

Analysis

Repeating the algorithm $c \ln n \binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le \left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le n^{-c} \enspace ,$$

where we used $1 - x \le e^{-x}$.

Theorem 89
The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4 \log n)$.

Analysis

Repeating the algorithm $c \ln n \binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1-\frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}c\ln n} \le \left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2}c\ln n} \le n^{-c} ,$$

where we used $1-x \le e^{-x}$.

Theorem 89

The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4 \log n)$.

Analysis

Repeating the algorithm $c\ln n\binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1-\frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}c\ln n}\le\left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2}c\ln n}\le n^{-c}\enspace,$$

where we used $1-x\le e^{-x}$.

Theorem 89

The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4\log n)$.

Analysis

Repeating the algorithm $c \ln n \binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le \left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le n^{-c} \ ,$$

where we used $1 - x \le e^{-x}$.

Theorem 89

The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4 \log n)$.

Analysis

Repeating the algorithm $c\ln n\binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1-\frac{1}{\binom{n}{2}}\right)^{\binom{n}{2}c\ln n} \le \left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2}c\ln n} \le n^{-c} \ ,$$

where we used $1-x \le e^{-x}$.

Theorem 89

The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4 \log n)$.

Analysis

Repeating the algorithm $c \ln n \binom{n}{2}$ times gives that the probability that we are never successful is

$$\left(1 - \frac{1}{\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le \left(e^{-1/\binom{n}{2}}\right)^{\binom{n}{2} c \ln n} \le n^{-c} \ ,$$

where we used $1 - x \le e^{-x}$.

Theorem 89

The randomized mincut algorithm computes an optimal cut with high probability. The total running time is $\mathcal{O}(n^4 \log n)$.

Improved Algorithm

Algorithm 53 RecursiveMincut($G = (V, E, c)$)

```
for $i = 1 \to n - n/\sqrt{2}$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
if $|V| = 2$ return cut-value;
$cuta \leftarrow$ RecursiveMincut(G);
$cutb \leftarrow$ RecursiveMincut(G);
return $\min\{cuta, cutb\}$
```

Running time:

- $T(n) = 2T(\frac{n}{\sqrt{2}}) + \mathcal{O}(n^2)$
- This gives $T(n) = \mathcal{O}(n^2 \log n)$.

Improved Algorithm

Algorithm 53 RecursiveMincut$(G = (V, E, c))$

```
for $i = 1 \to n - n/\sqrt{2}$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
if $|V| = 2$ return cut-value;
$cuta \leftarrow$ RecursiveMincut(G);
$cutb \leftarrow$ RecursiveMincut(G);
return $\min\{cuta, cutb\}$
```

Running time:

- $T(n) = 2T(\frac{n}{\sqrt{2}}) + \mathcal{O}(n^2)$
- This gives $T(n) = \mathcal{O}(n^2 \log n)$.

Improved Algorithm

Algorithm 53 RecursiveMincut($G = (V, E, c)$)

```
for $i = 1 \to n - n/\sqrt{2}$ do
    choose $e \in E$ randomly with probability $c(e)/C(E)$
    $G \leftarrow G/e$
if $|V| = 2$ return cut-value;
$cuta \leftarrow$ RecursiveMincut(G);
$cutb \leftarrow$ RecursiveMincut(G);
return $\min\{cuta, cutb\}$
```

Running time:

- $T(n) = 2T(\frac{n}{\sqrt{2}}) + \mathcal{O}(n^2)$
- This gives $T(n) = \mathcal{O}(n^2 \log n)$.

Probability of Success

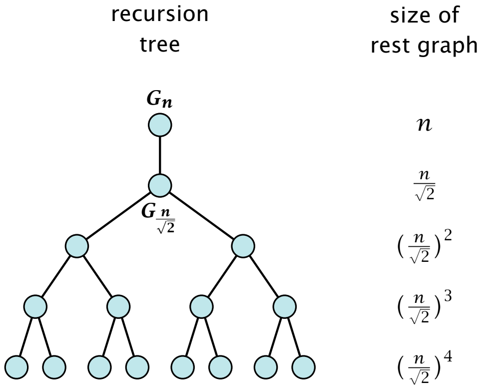
The probability of contracting an edge from the mincut during one iteration through the for-loop is only

$$\frac{t(t-1)}{n(n-1)} \approx \frac{t^2}{n^2} = \frac{1}{2} \ ,$$

as $t = \frac{n}{\sqrt{2}}$.

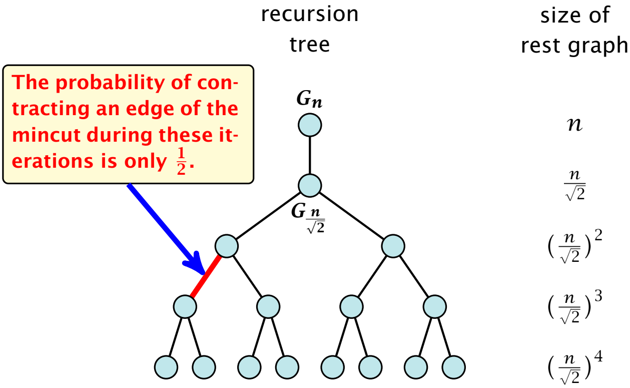
For the following analysis we ignore the slight error and assume that this probability is at most $\frac{1}{2}$.

Probability of Success



We can estimate the success probability by using the following game on the recursion tree. Delete every edge with probability $\frac{1}{2}$. If in the end you have a path from the root to at least one leaf node you are successful.

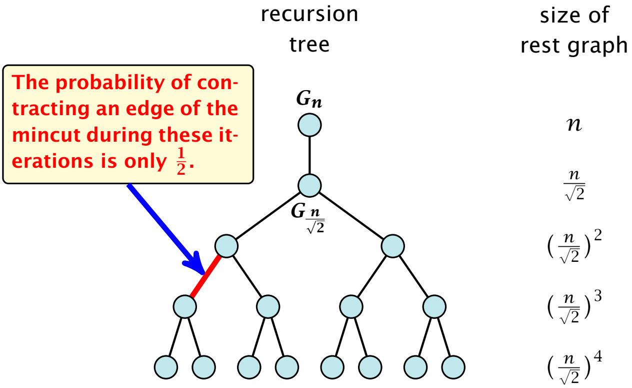
Probability of Success



The probability of contracting an edge of the mincut during these iterations is only $\frac{1}{2}$.

We can estimate the success probability by using the following game on the recursion tree. Delete every edge with probability $\frac{1}{2}$. If in the end you have a path from the root to at least one leaf node you are successful.

Probability of Success



The probability of contracting an edge of the mincut during these iterations is only $\frac{1}{2}$.

We can estimate the success probability by using the following game on the recursion tree. Delete every edge with probability $\frac{1}{2}$. If in the end you have a path from the root to at least one leaf node you are successful.

Probability of Success

Let for an edge e in the recursion tree, $h(e)$ denote the height (distance to leaf level) of the parent-node of e (end-point that is higher up in the tree). Let h denote the height of the root node.

Call an edge e alive if there exists a path from the parent-node of e to a descendant leaf, after we randomly deleted edges. Note that an edge can only be alive if it hasn't been deleted.

Lemma 90

The probability that an edge e is alive is at least $\frac{1}{h(e)+1}$.

Probability of Success

Let for an edge e in the recursion tree, $h(e)$ denote the height (distance to leaf level) of the parent-node of e (end-point that is higher up in the tree). Let h denote the height of the root node.

Call an edge e alive if there exists a path from the parent-node of e to a descendant leaf, after we randomly deleted edges. Note that an edge can only be alive if it hasn't been deleted.

Lemma 90

The probability that an edge e is alive is at least $\frac{1}{h(e)+1}$.

Probability of Success

Let for an edge e in the recursion tree, $h(e)$ denote the height (distance to leaf level) of the parent-node of e (end-point that is higher up in the tree). Let h denote the height of the root node.

Call an edge e alive if there exists a path from the parent-node of e to a descendant leaf, after we randomly deleted edges. Note that an edge can only be alive if it hasn't been deleted.

Lemma 90

The probability that an edge e is alive is at least $\frac{1}{h(e)+1}$.

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$p_d$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$p_d = \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right)$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$p_d = \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right) \quad \boxed{\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]}$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$\begin{aligned} p_d &= \frac{1}{2}\Big(2p_{d-1} - p_{d-1}^2\Big) \quad \boxed{\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]} \\ &= p_{d-1} - \frac{p_{d-1}^2}{2} \end{aligned}$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$\begin{aligned} p_d &= \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right) \\ &= p_{d-1} - \frac{p_{d-1}^2}{2} \end{aligned}$$

$\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$

$x - x^2/2$ is monotonically increasing for $x \in [0, 1]$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$p_d = \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right) \quad \boxed{\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]}$$

$$= p_{d-1} - \frac{p_{d-1}^2}{2}$$

$$\boxed{\text{$x - x^2/2$ is monotonically increasing for $x \in [0,1]$}} \quad \geq \frac{1}{d} - \frac{1}{2d^2}$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$$p_d = \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right)$$

$\Pr[A \vee B] = \Pr[A] + \Pr[B] - \Pr[A \wedge B]$

$$= p_{d-1} - \frac{p_{d-1}^2}{2}$$

$x - x^2/2$ is monotonically increasing for $x \in [0, 1]$

$$\geq \frac{1}{d} - \frac{1}{2d^2} \geq \frac{1}{d} - \frac{1}{d(d+1)}$$

Probability of Success

Proof.

- An edge e with $h(e) = 1$ is alive if and only if it is not deleted. Hence, it is alive with proability at least $\frac{1}{2}$.
- Let p_d be the probability that an edge e with $h(e) = d$ is alive. For $d > 1$ this happens for edge $e = \{c, p\}$ if it is not deleted **and** if one of the child-edges connecting to c is alive.
- This happens with probability

$\mathrm{Pr}[A \vee B] = \mathrm{Pr}[A] + \mathrm{Pr}[B] - \mathrm{Pr}[A \wedge B]$

$x - x^2/2$ is monotonically increasing for $x \in [0, 1]$

$$\begin{aligned} p_d &= \frac{1}{2}\left(2p_{d-1} - p_{d-1}^2\right) \\ &= p_{d-1} - \frac{p_{d-1}^2}{2} \\ &\geq \frac{1}{d} - \frac{1}{2d^2} \geq \frac{1}{d} - \frac{1}{d(d+1)} = \frac{1}{d+1} \enspace . \end{aligned}$$

16 Global Mincut

Lemma 91

One run of the algorithm can be performed in time $\mathcal{O}(n^2 \log n)$ *and has a success probability of* $\Omega(\frac{1}{\log n})$.

Doing $\Theta(\log^2 n)$ *runs gives that the algorithm succeeds with high probability. The total running time is* $\mathcal{O}(n^2 \log^3 n)$.

16 Global Mincut

Lemma 91

One run of the algorithm can be performed in time $\mathcal{O}(n^2 \log n)$ *and has a success probability of* $\Omega(\frac{1}{\log n})$*.*

Doing $\Theta(\log^2 n)$ *runs gives that the algorithm succeeds with high probability. The total running time is* $\mathcal{O}(n^2 \log^3 n)$*.*

17 Gomory Hu Trees

Given an undirected, weighted graph $G = (V, E, c)$ a cut-tree $T = (V, F, w)$ is a tree with edge-set F and capacities w that fulfills the following properties.

1. **Equivalent Flow Tree:** For any pair of vertices $s, t \in V$, $f(s,t)$ in G is equal to $f_T(s,t)$.
2. **Cut Property:** A minimum s-t cut in T is also a minimum cut in G.

Here, $f(s,t)$ is the value of a maximum s-t flow in G, and $f_T(s,t)$ is the corresponding value in T.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \dots, S_t$), and a spanning tree T on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \dots, S_t$), and a spanning tree T on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \dots, S_t$), and a spanning tree T on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \ldots, S_t$), and a spanning tree T on the vertex set $\{S_1, \ldots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \ldots, S_t$), and a spanning tree T on the vertex set $\{S_1, \ldots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \dots, S_t$), and a spanning tree T on the vertex set $\{S_1, \dots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Overview of the Algorithm

The algorithm maintains a partition of V, (sets $S_1, \ldots, S_t$), and a spanning tree T on the vertex set $\{S_1, \ldots, S_t\}$.

Initially, there exists only the set $S_1 = V$.

Then the algorithm performs $n - 1$ split-operations:

- In each such split-operation it chooses a set S_i with $|S_i| \geq 2$ and splits this set into two non-empty parts X and Y.
- S_i is then removed from T and replaced by X and Y.
- X and Y are connected by an edge, and the edges that before the split were incident to S_i are attached to either X or Y.

In the end this gives a tree on the vertex set V.

Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a := S_i \cap A$ and $S_i^b := S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

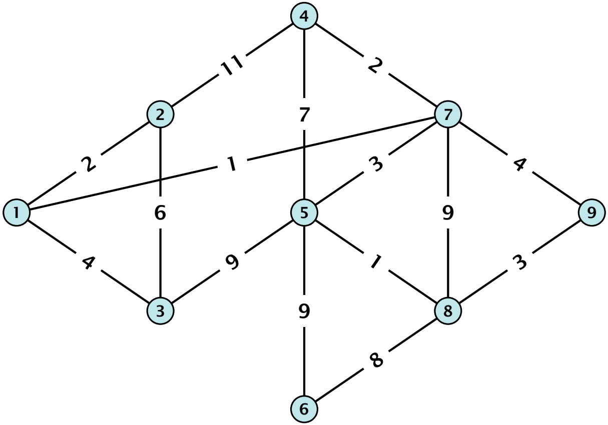
Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

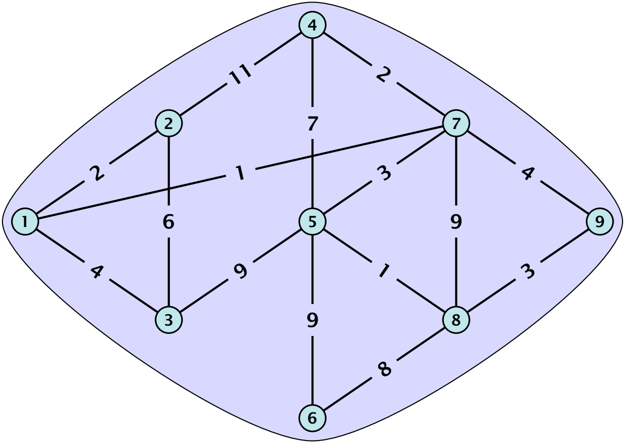
Details of the Split-operation

- Select S_i that contains at least two nodes a and b.
- Compute the connected components of the forest obtained from the current tree T after deleting S_i. Each of these components corresponds to a set of vertices from V.
- Consider the graph H obtained from G by contracting these connected components into single nodes.
- Compute a minimum a-b cut in H. Let A, and B denote the two sides of this cut.
- Split S_i in T into two sets/nodes $S_i^a \coloneqq S_i \cap A$ and $S_i^b \coloneqq S_i \cap B$ and add edge $\{S_i^a, S_i^b\}$ with capacity $f_H(a,b)$.
- Replace an edge $\{S_i, S_x\}$ by $\{S_i^a, S_x\}$ if $S_x \subset A$ and by $\{S_i^b, S_x\}$ if $S_x \subset B$.

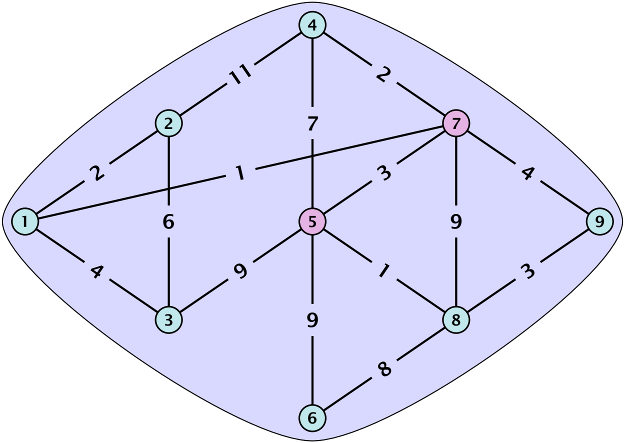
Example: Gomory-Hu Construction



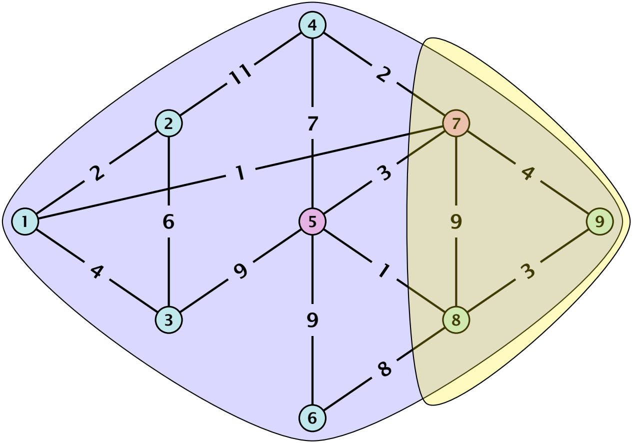
Example: Gomory-Hu Construction



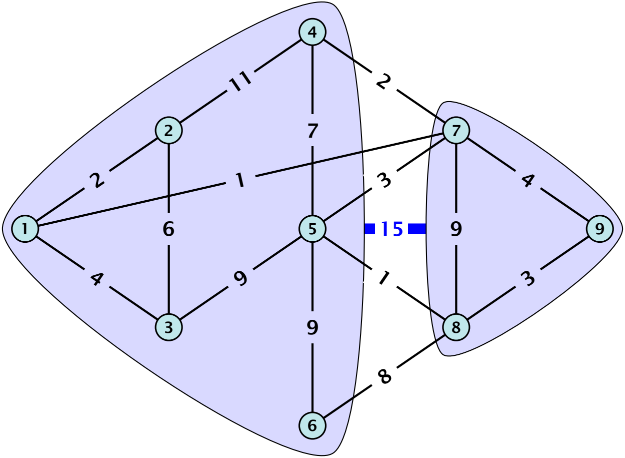
Example: Gomory-Hu Construction



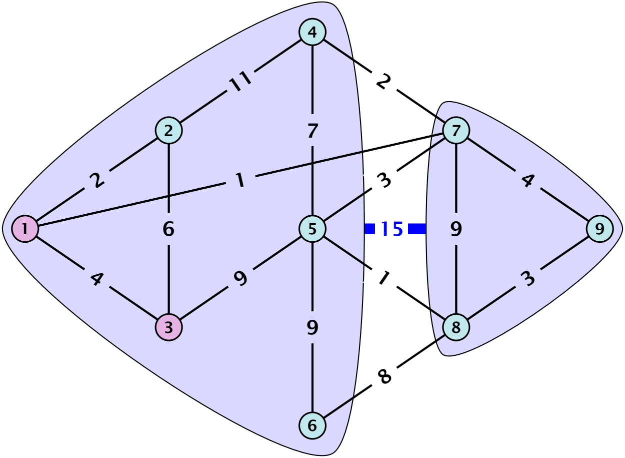
Example: Gomory-Hu Construction



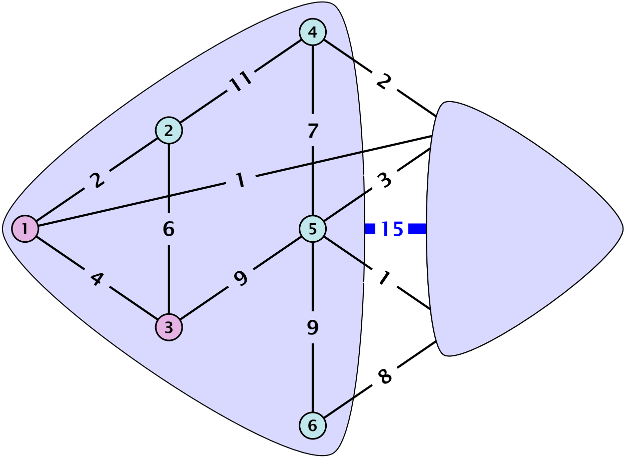
Example: Gomory-Hu Construction



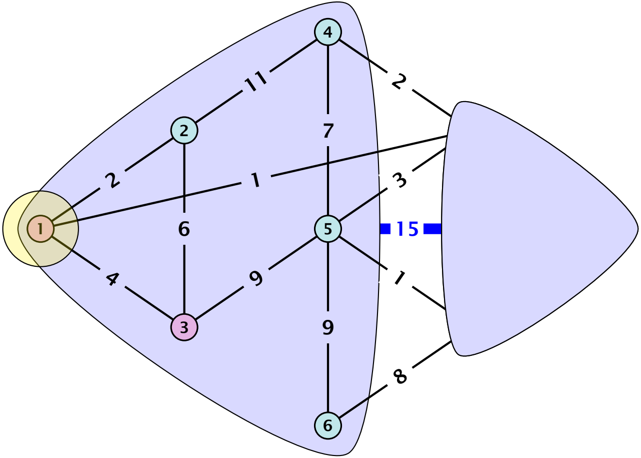
Example: Gomory-Hu Construction



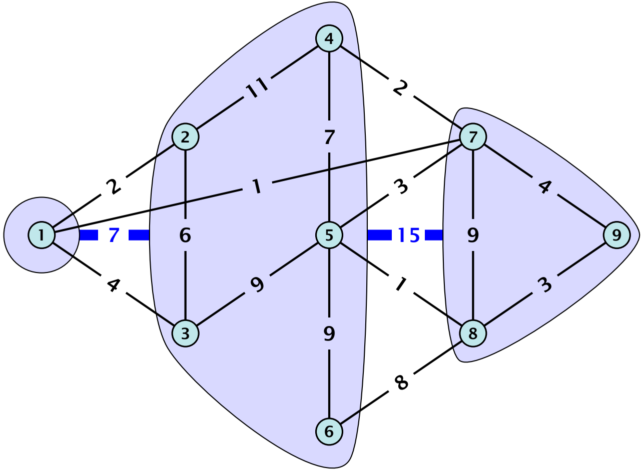
Example: Gomory-Hu Construction



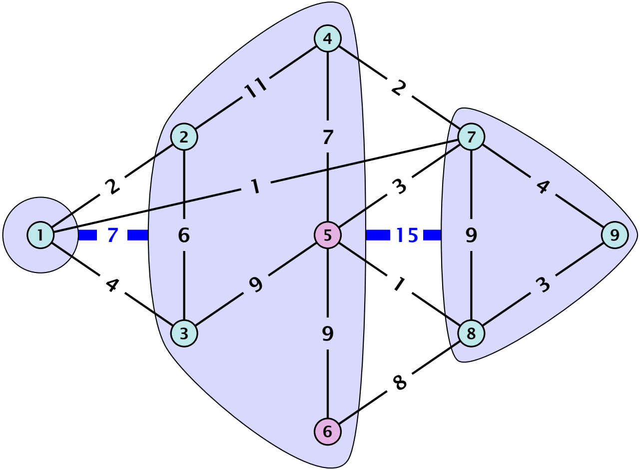
Example: Gomory-Hu Construction



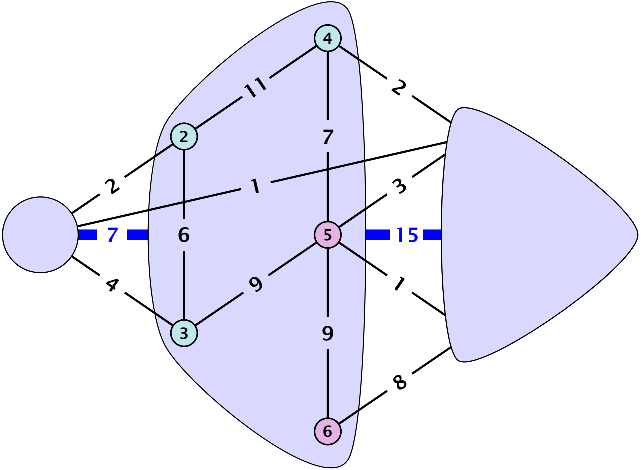
Example: Gomory-Hu Construction



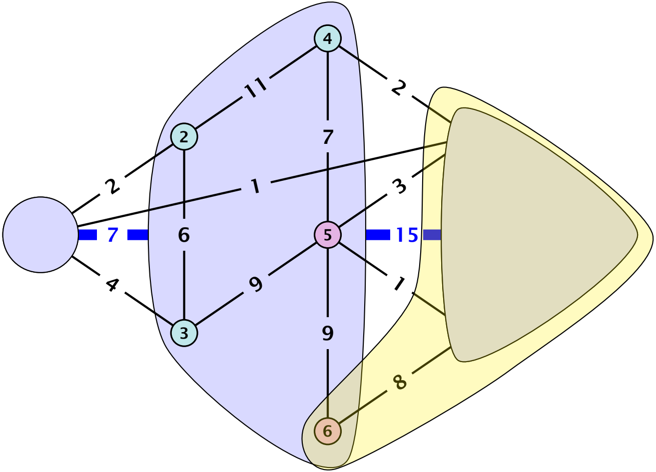
Example: Gomory-Hu Construction



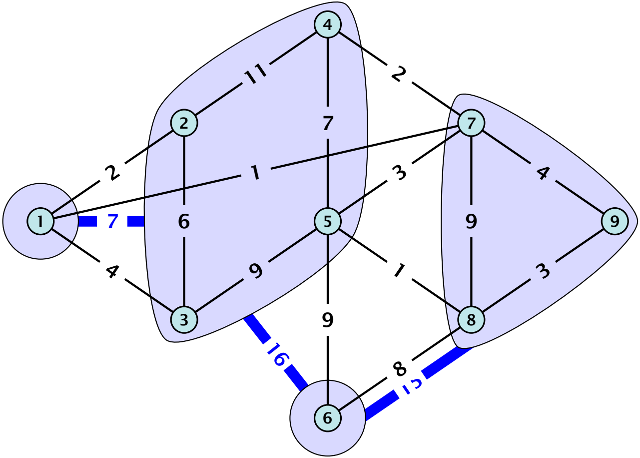
Example: Gomory-Hu Construction



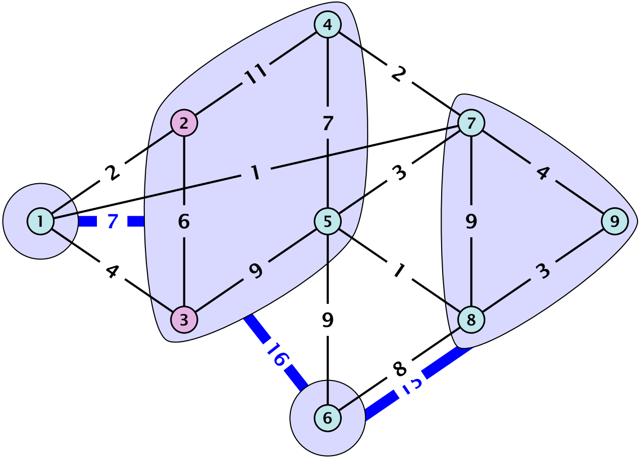
Example: Gomory-Hu Construction



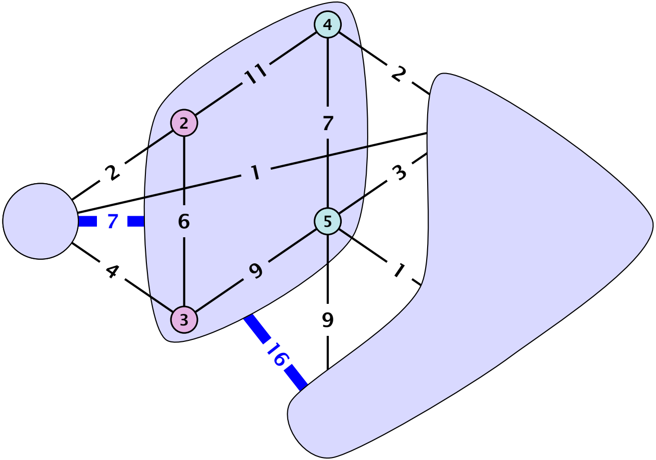
Example: Gomory-Hu Construction



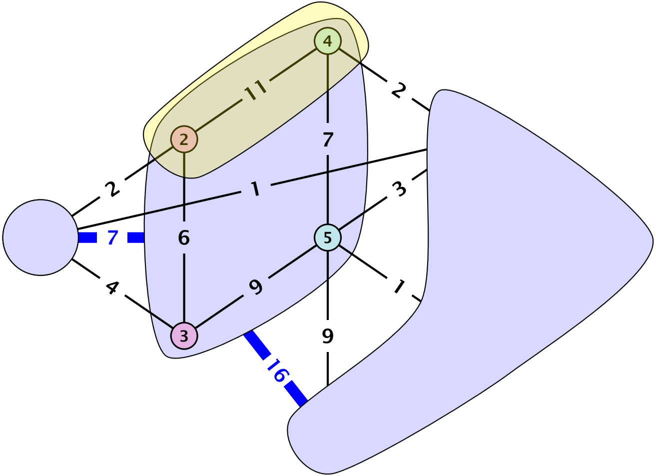
Example: Gomory-Hu Construction



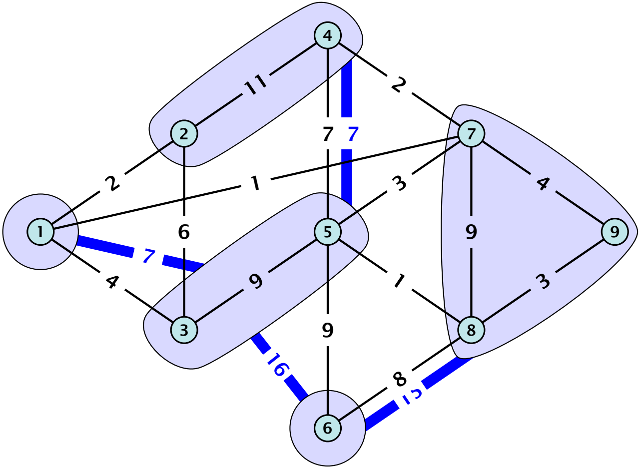
Example: Gomory-Hu Construction



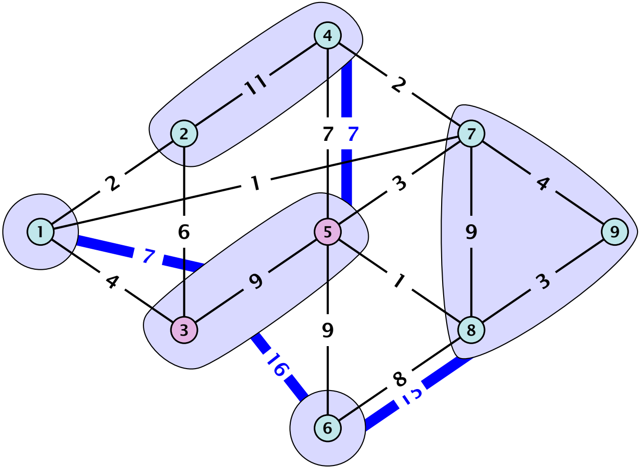
Example: Gomory-Hu Construction



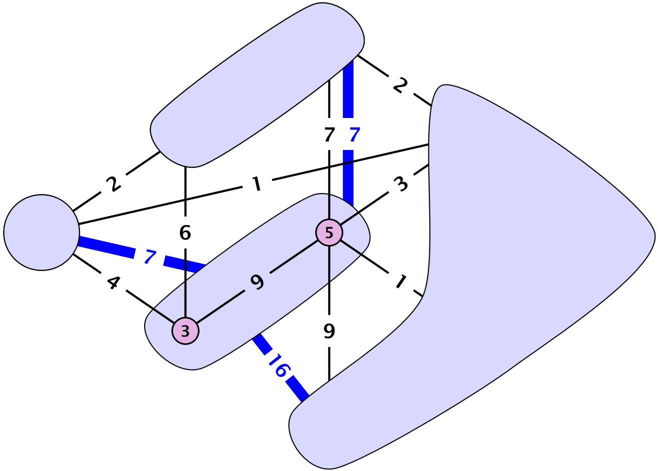
Example: Gomory-Hu Construction



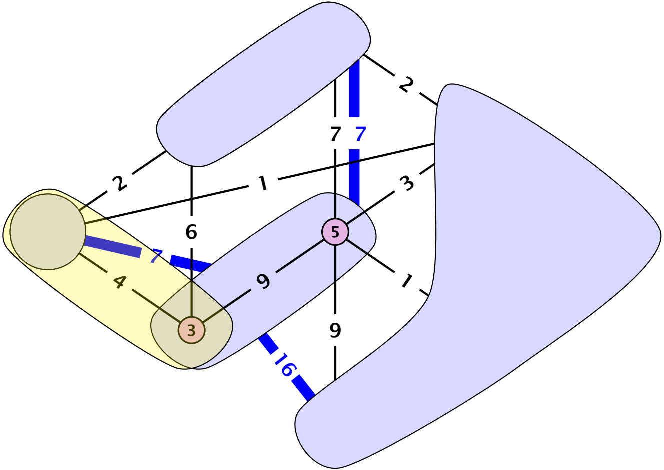
Example: Gomory-Hu Construction



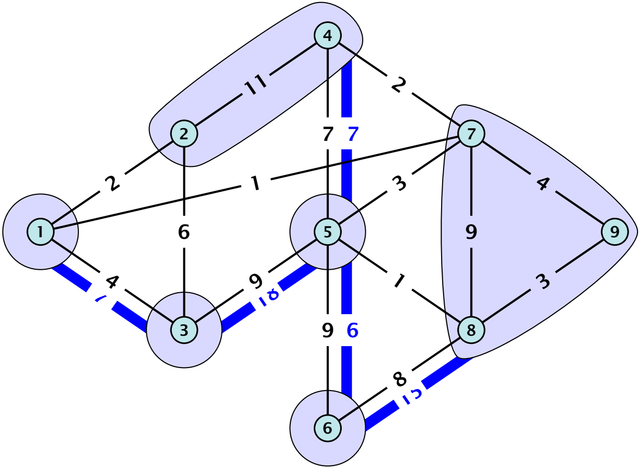
Example: Gomory-Hu Construction



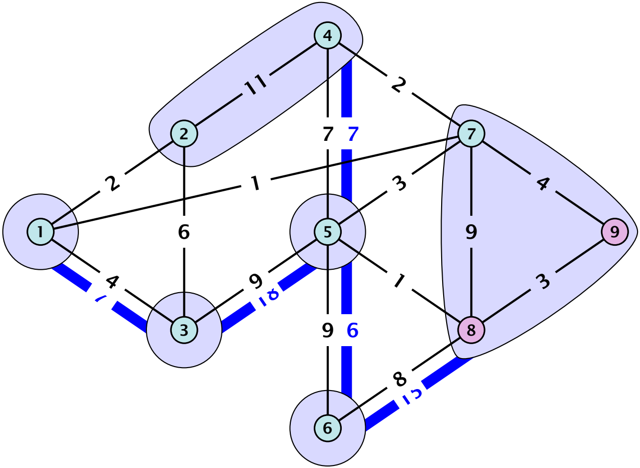
Example: Gomory-Hu Construction



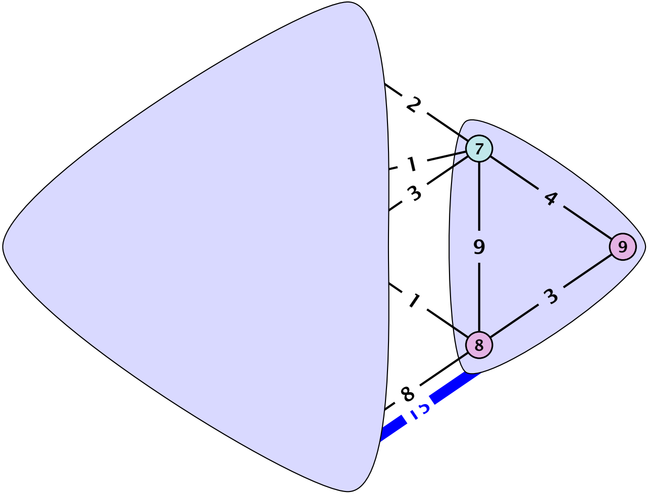
Example: Gomory-Hu Construction



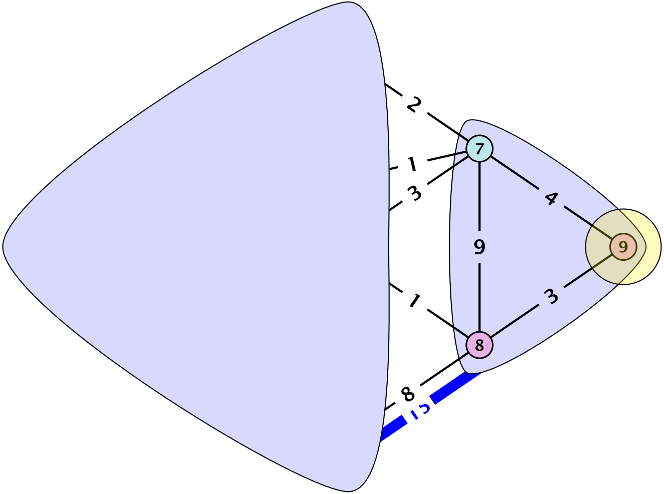
Example: Gomory-Hu Construction



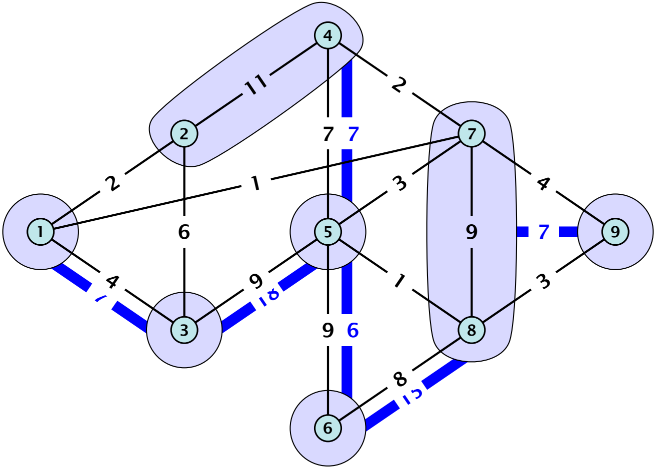
Example: Gomory-Hu Construction



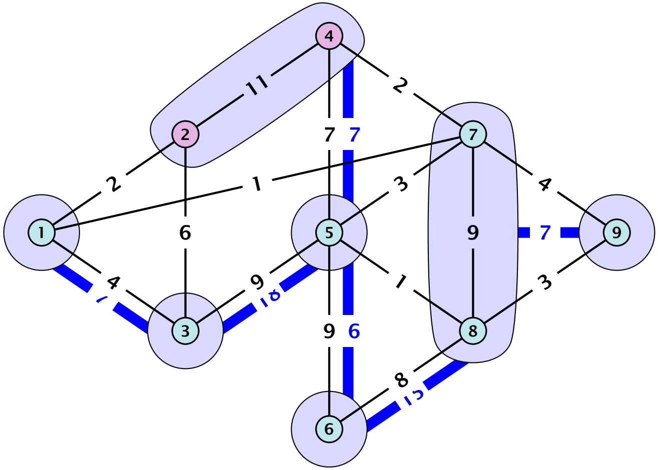
Example: Gomory-Hu Construction



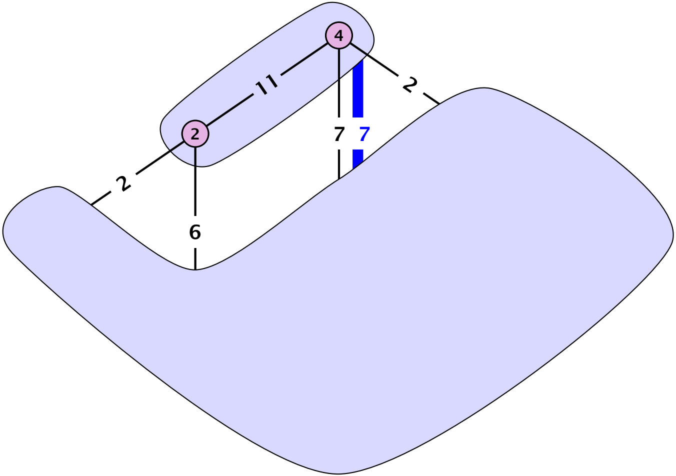
Example: Gomory-Hu Construction



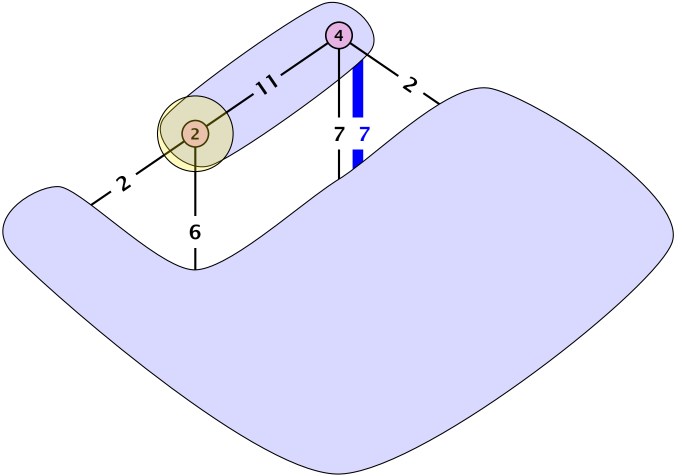
Example: Gomory-Hu Construction



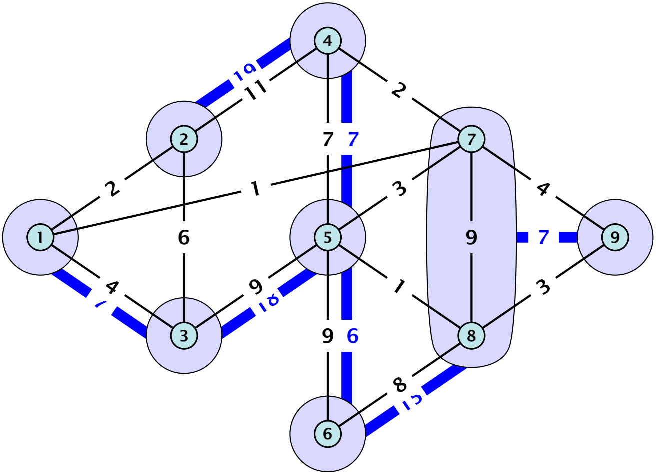
Example: Gomory-Hu Construction



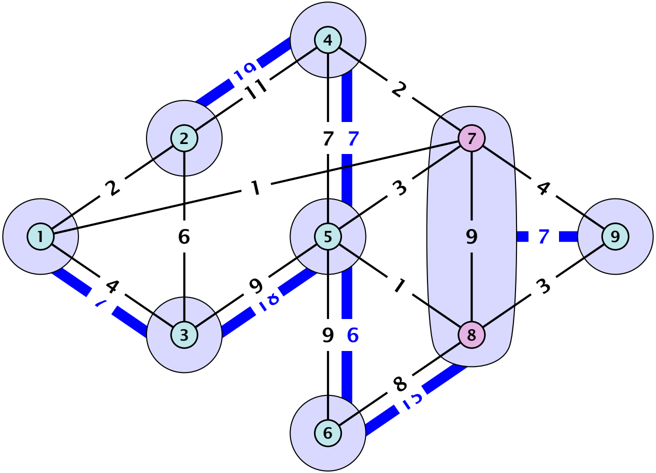
Example: Gomory-Hu Construction



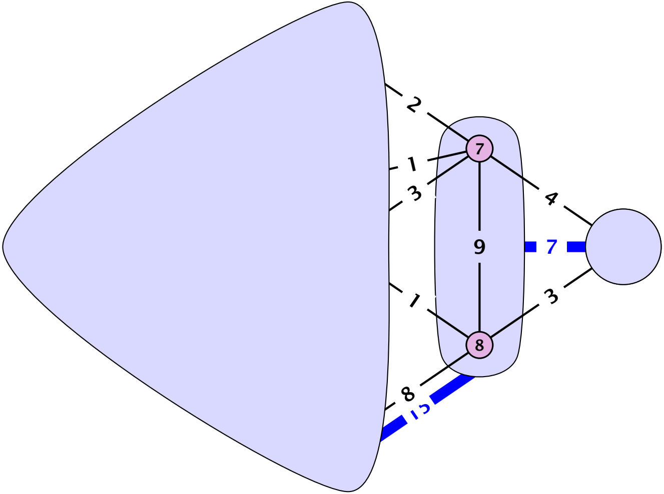
Example: Gomory-Hu Construction



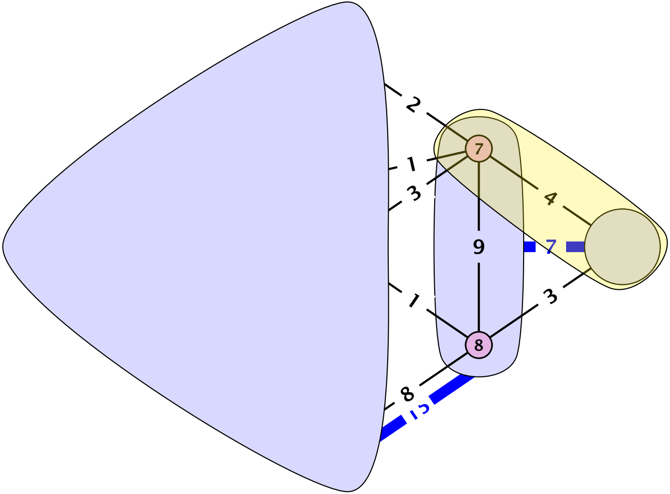
Example: Gomory-Hu Construction



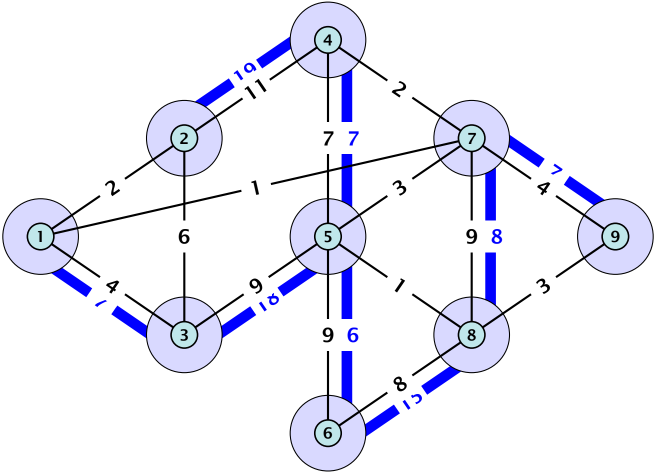
Example: Gomory-Hu Construction



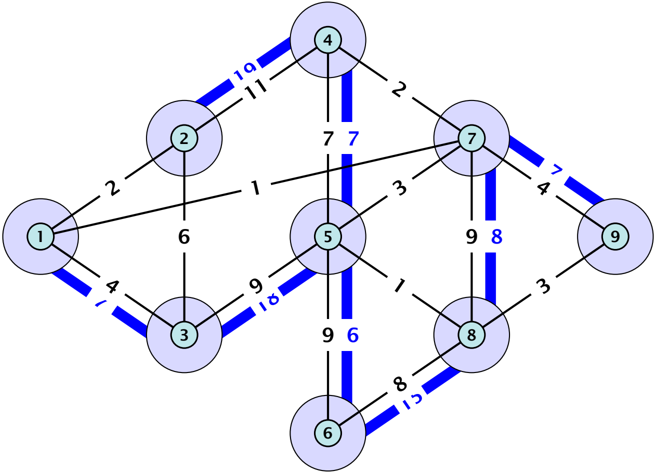
Example: Gomory-Hu Construction



Example: Gomory-Hu Construction



Example: Gomory-Hu Construction



Analysis

Lemma 92

For nodes $s, t, x \in V$ *we have* $f(s,t) \geq \min\{f(s,x), f(x,t)\}$

Lemma 93

For nodes $s, t, x_1, \ldots, x_k \in V$ *we have*
$f(s,t) \geq \min\{f(s,x_1), f(x_1,x_2), \ldots, f(x_{k-1},x_k), f(x_k,t)\}$

Analysis

Lemma 92

For nodes $s, t, x \in V$ *we have* $f(s,t) \geq \min\{f(s,x), f(x,t)\}$

Lemma 93

For nodes $s, t, x_1, \ldots, x_k \in V$ *we have*
$f(s,t) \geq \min\{f(s,x_1), f(x_1,x_2), \ldots, f(x_{k-1},x_k), f(x_k,t)\}$

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\mathrm{cap}(S \setminus X) \leq \mathrm{cap}(X)$.

Second case $r \notin X$.

- $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \cup S) \geq \mathrm{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\mathrm{cap}(S \cap X) \leq \mathrm{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\mathrm{cap}(X \setminus S) + \mathrm{cap}(S \setminus X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \setminus S) \geq \mathrm{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\mathrm{cap}(S \setminus X) \leq \mathrm{cap}(X)$.

Second case $r \notin X$.

- $\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$.
- $\mathrm{cap}(X \cup S) \geq \mathrm{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\mathrm{cap}(S \cap X) \leq \mathrm{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

Second case $r \notin X$.

- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

Lemma 94

Let S be some minimum r-s cut for some nodes $r, s \in V$ ($s \in S$), and let $v, w \in S$. Then there is a minimum v-w-cut T with $T \subset S$.

Proof: Let X be a minimum v-w cut with $X \cap S \neq \emptyset$ and $X \cap (V \setminus S) \neq \emptyset$. Note that $S \setminus X$ and $S \cap X$ are v-w cuts inside S.

We may assume w.l.o.g. $s \in X$.

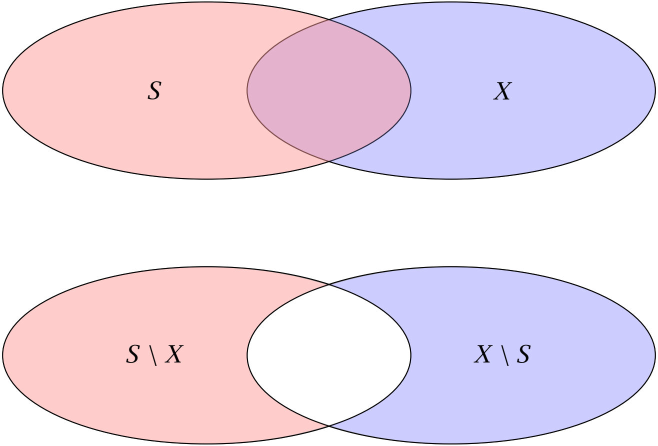
First case $r \in X$.

- $\text{cap}(X \setminus S) + \text{cap}(S \setminus X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \setminus S) \geq \text{cap}(S)$ because $X \setminus S$ is an r-s cut.
- This gives $\text{cap}(S \setminus X) \leq \text{cap}(X)$.

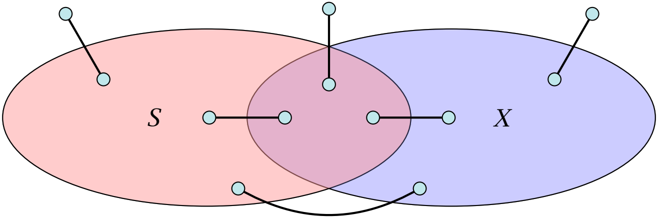
Second case $r \notin X$.

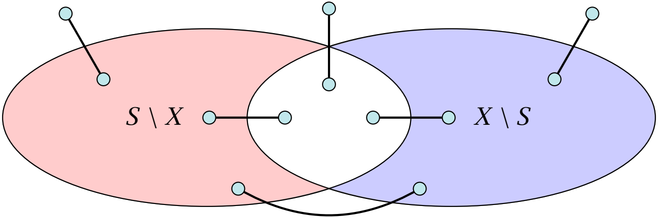
- $\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$.
- $\text{cap}(X \cup S) \geq \text{cap}(S)$ because $X \cup S$ is an r-s cut.
- This gives $\text{cap}(S \cap X) \leq \text{cap}(X)$.

$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$

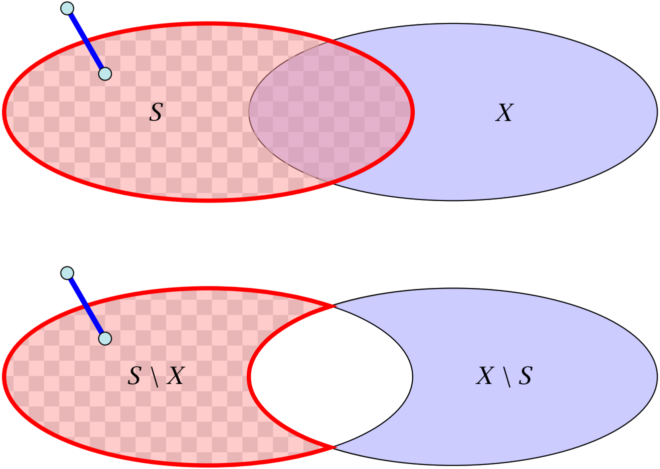


$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

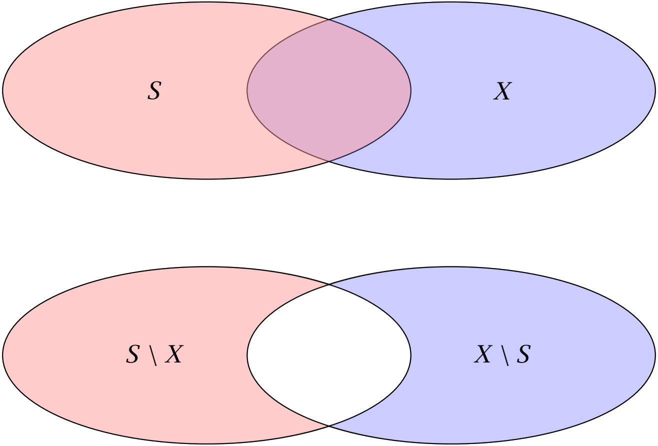




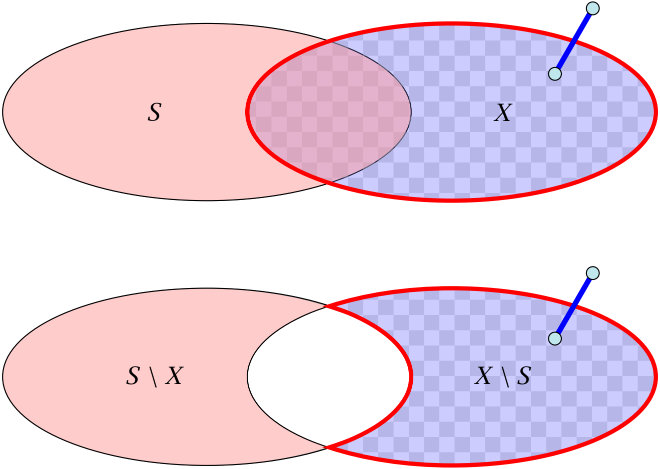
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



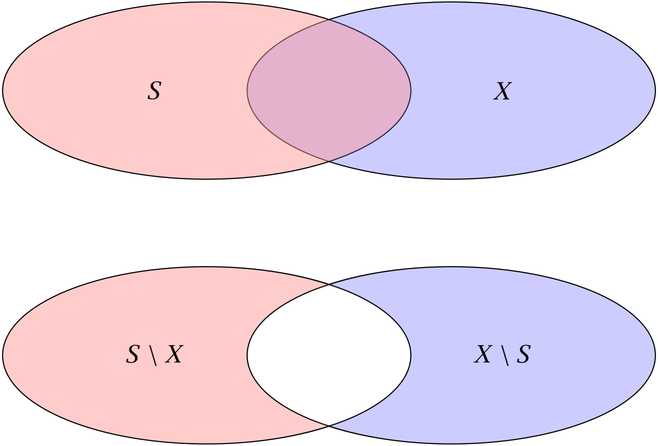
$\text{cap}(S \setminus X) + \text{cap}(X \setminus S) \leq \text{cap}(S) + \text{cap}(X)$



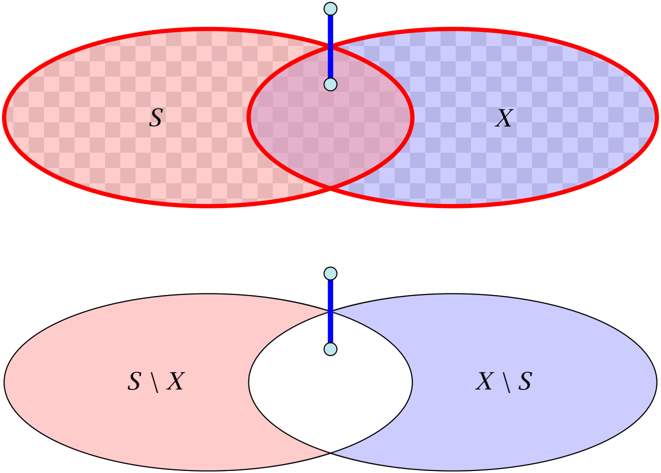
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



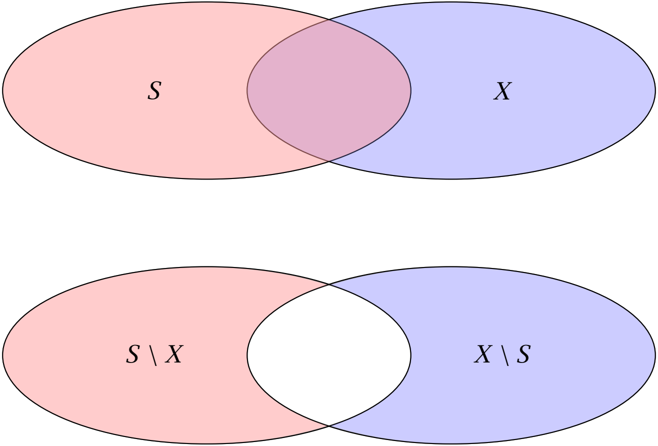
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



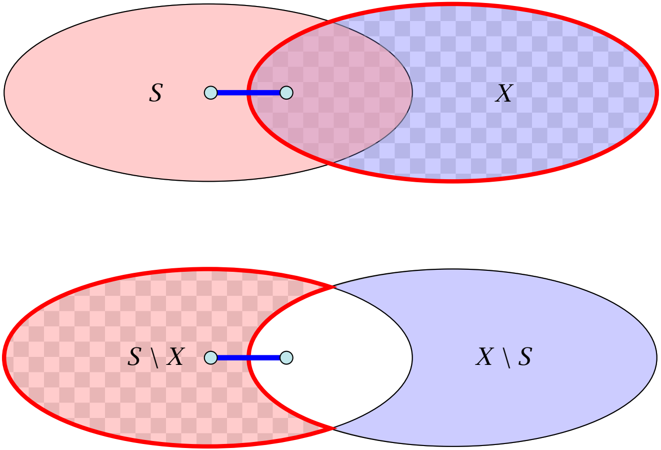
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



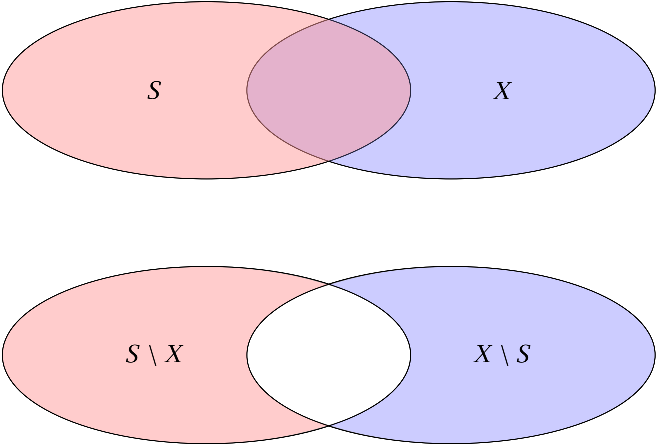
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



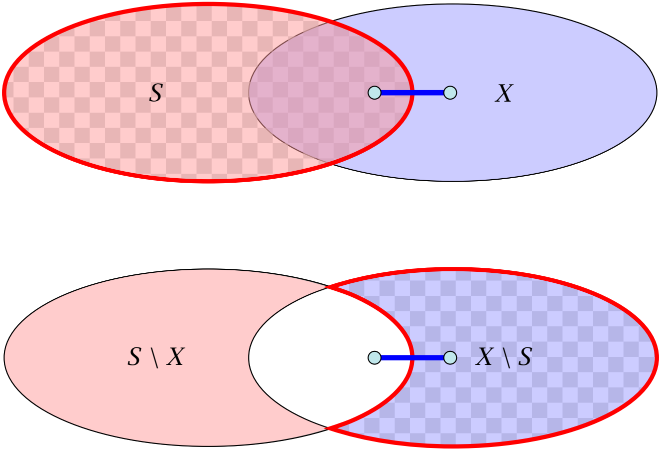
$\operatorname{cap}(S \setminus X) + \operatorname{cap}(X \setminus S) \le \operatorname{cap}(S) + \operatorname{cap}(X)$



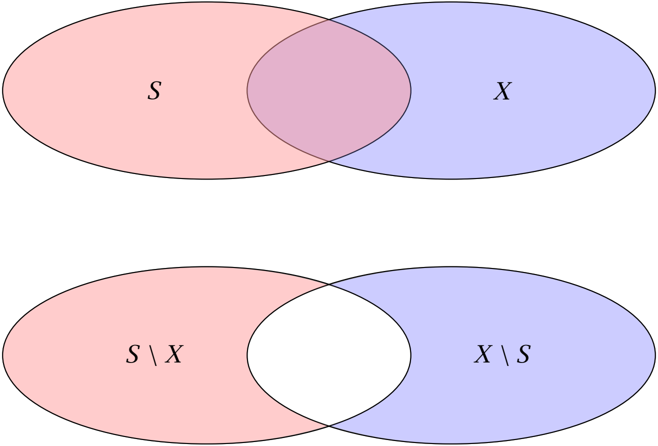
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



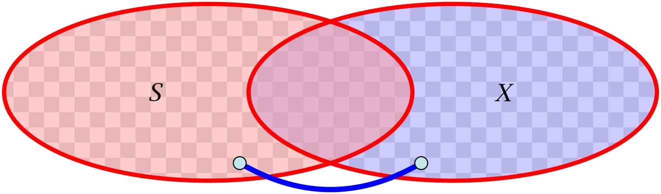
$$\mathbf{cap}(S \setminus X) + \mathbf{cap}(X \setminus S) \le \mathbf{cap}(S) + \mathbf{cap}(X)$$

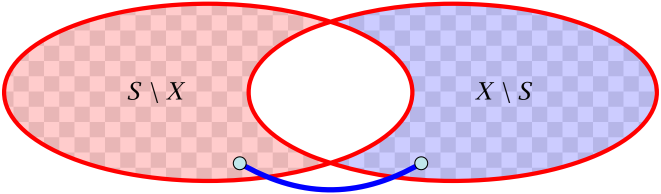


$$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$$

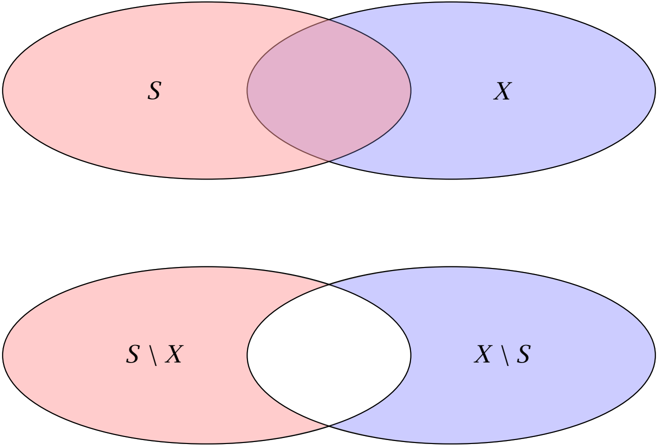


$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$

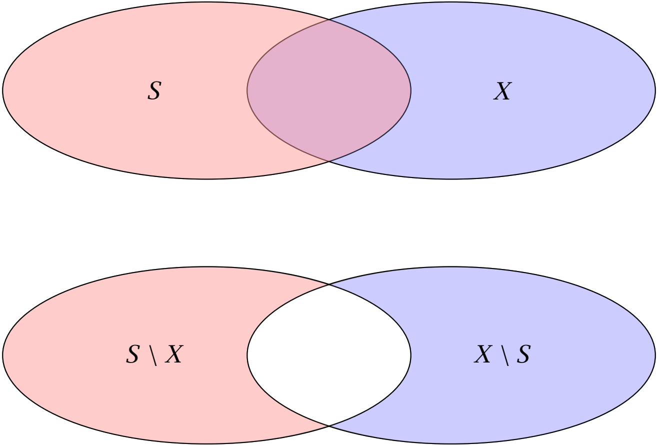




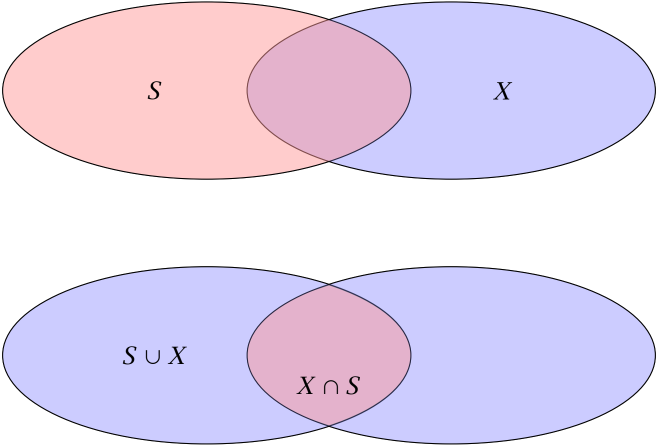
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



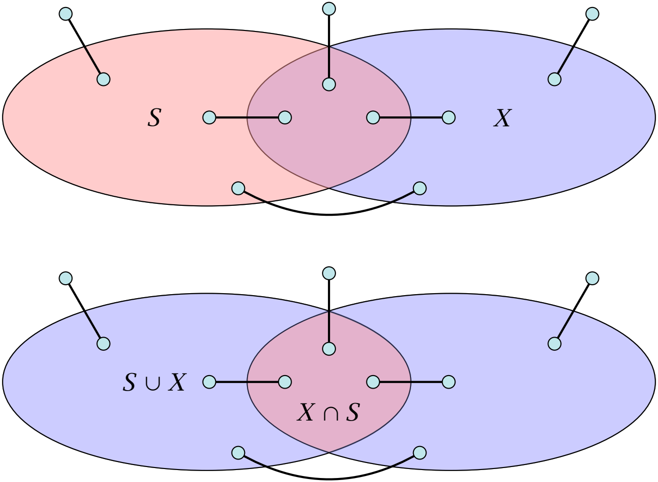
$\mathrm{cap}(S \setminus X) + \mathrm{cap}(X \setminus S) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



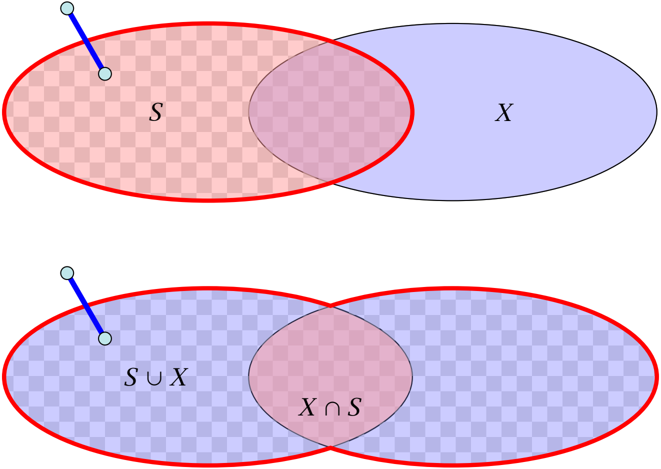
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



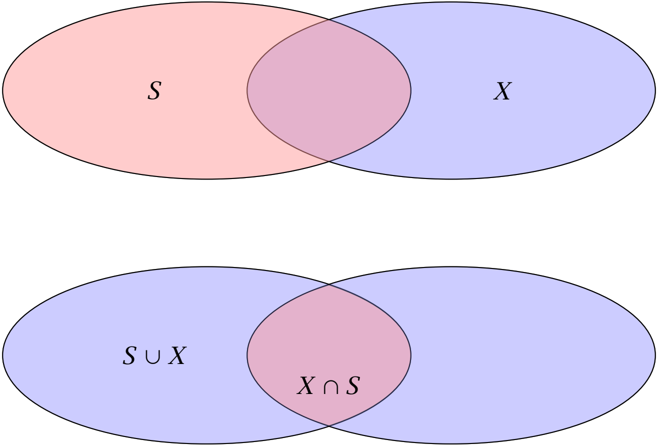
$\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$



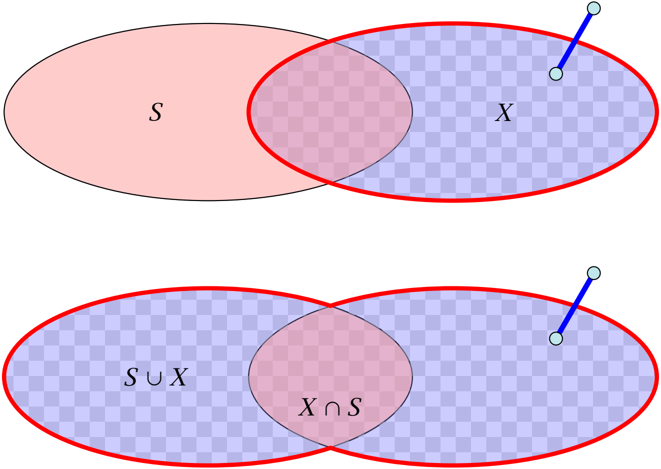
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



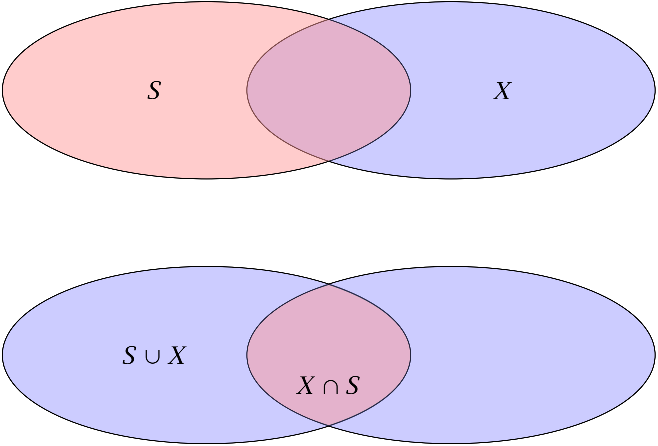
$$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$$



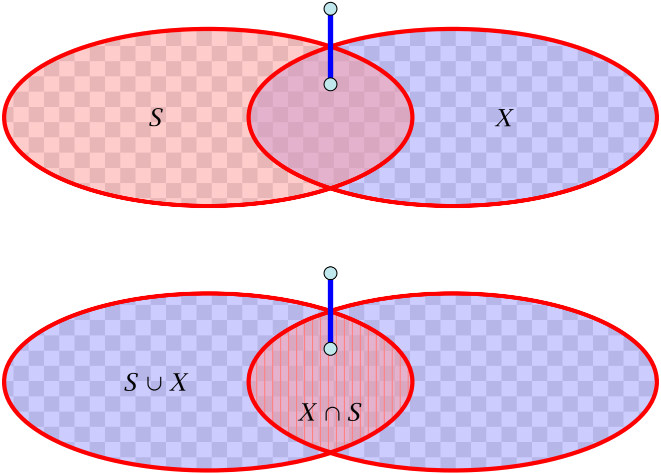
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



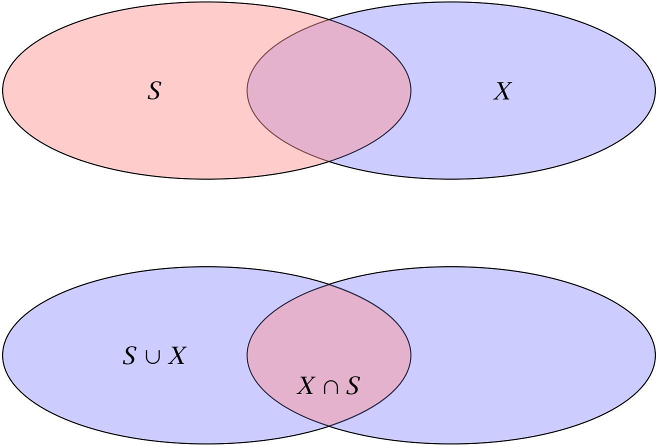
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



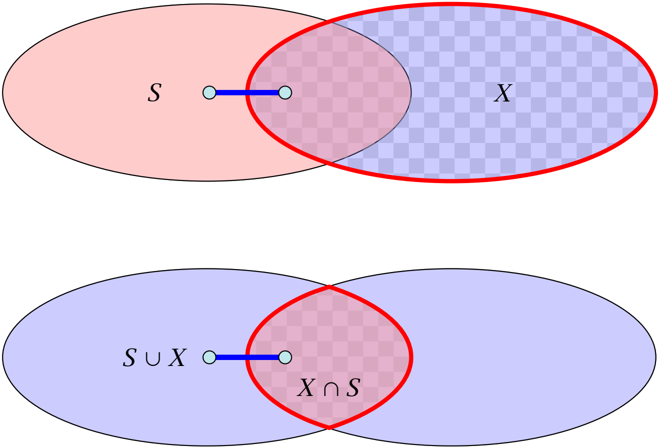
$\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$



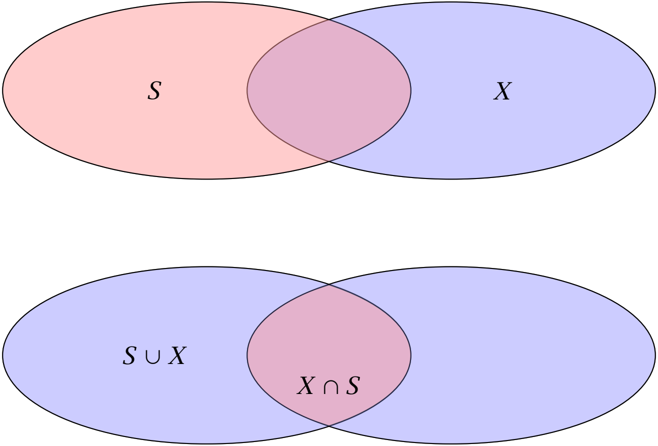
$$\mathbf{cap}(X \cup S) + \mathbf{cap}(S \cap X) \le \mathbf{cap}(S) + \mathbf{cap}(X)$$



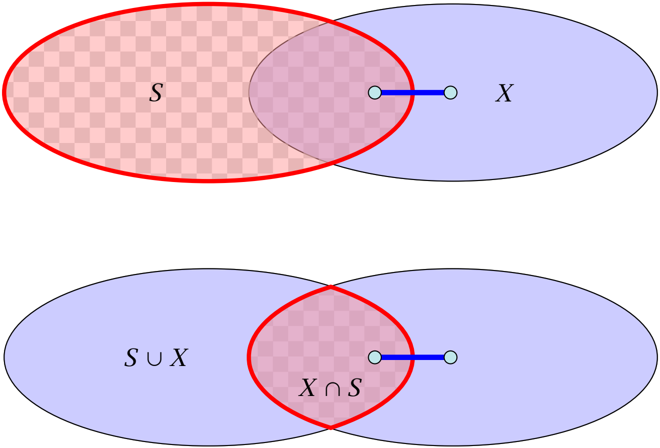
$\text{cap}(X \cup S) + \text{cap}(S \cap X) \leq \text{cap}(S) + \text{cap}(X)$



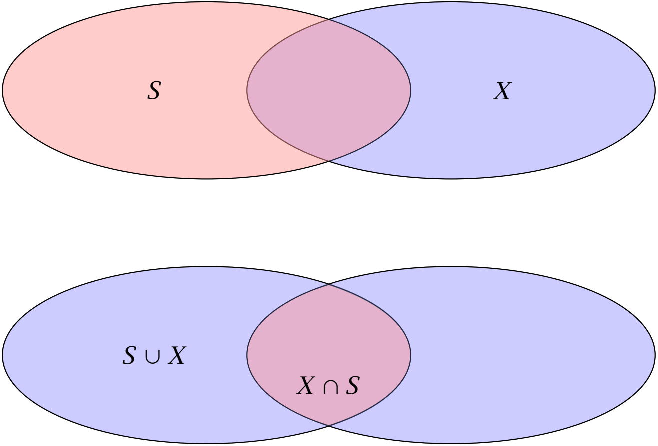
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



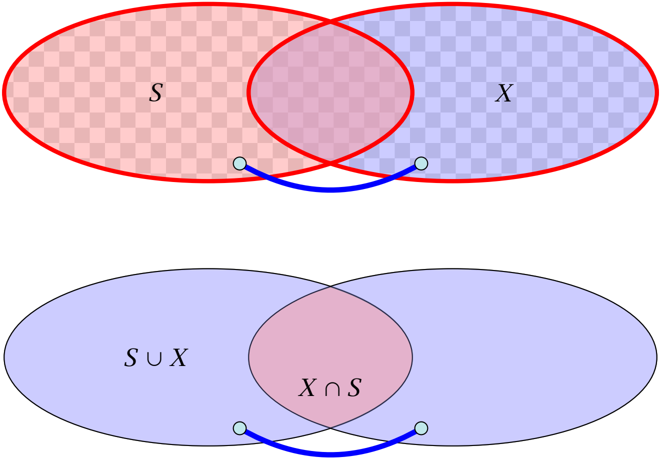
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



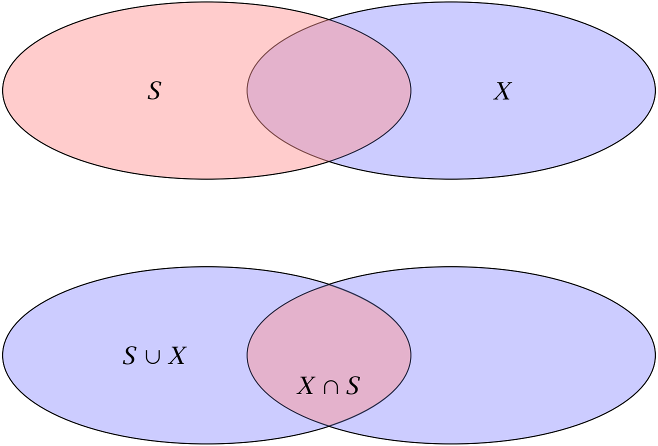
$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \leq \mathrm{cap}(S) + \mathrm{cap}(X)$



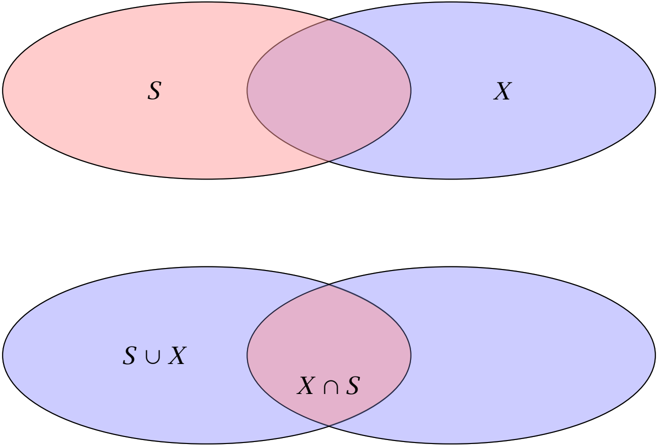
$\mathbf{cap}(X \cup S) + \mathbf{cap}(S \cap X) \leq \mathbf{cap}(S) + \mathbf{cap}(X)$



$\mathbf{cap}(X \cup S) + \mathbf{cap}(S \cap X) \le \mathbf{cap}(S) + \mathbf{cap}(X)$



$\mathrm{cap}(X \cup S) + \mathrm{cap}(S \cap X) \le \mathrm{cap}(S) + \mathrm{cap}(X)$



Analysis

Lemma 94 tells us that if we have a graph $G = (V, E)$ and we contract a subset $X \subset V$ that corresponds to some mincut, then the value of $f(s, t)$ does not change for two nodes $s, t \notin X$.

We will show (later) that the connected components that we contract during a split-operation each correspond to some mincut and, hence, $f_H(s, t) = f(s, t)$, where $f_H(s, t)$ is the value of a minimum s-t mincut in graph H.

Analysis

Invariant [existence of representatives]:
For any edge $\{S_i, S_j\}$ in T, there are vertices $a \in S_i$ and $b \in S_j$ such that $w(S_i, S_j) = f(a, b)$ and the cut defined by edge $\{S_i, S_j\}$ is a minimum a-b cut in G.

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$f_T(s,t)$$

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$f_T(s,t) = \min_{i \in \{0,\dots,k-1\}} \{w(x_i, x_{i+1})\}$$

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \ldots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$\begin{aligned} f_T(s,t) &= \min_{i\in\{0,\ldots,k-1\}} \{w(x_i, x_{i+1})\} \\ &= \min_{i\in\{0,\ldots,k-1\}} \{f(x_i, x_{i+1})\} \end{aligned}$$

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$\begin{aligned} f_T(s,t) &= \min_{i\in\{0,\dots,k-1\}} \{w(x_i, x_{i+1})\} \\ &= \min_{i\in\{0,\dots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \ . \end{aligned}$$

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \dots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$\begin{aligned} f_T(s,t) &= \min_{i\in\{0,\dots,k-1\}} \{w(x_i, x_{i+1})\} \\ &= \min_{i\in\{0,\dots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \ . \end{aligned}$$

- Let $\{x_j, x_{j+1}\}$ be the edge with minimum weight on the path.

Analysis

We first show that the invariant implies that at the end of the algorithm T is indeed a cut-tree.

- Let $s = x_0, x_1, \ldots, x_{k-1}, x_k = t$ be the unique simple path from s to t in the final tree T. From the invariant we get that $f(x_i, x_{i+1}) = w(x_i, x_{i+1})$ for all j.
- Then

$$\begin{aligned} f_T(s,t) &= \min_{i \in \{0,\ldots,k-1\}} \{w(x_i, x_{i+1})\} \\ &= \min_{i \in \{0,\ldots,k-1\}} \{f(x_i, x_{i+1})\} \le f(s,t) \enspace . \end{aligned}$$

- Let $\{x_j, x_{j+1}\}$ be the edge with minimum weight on the path.
- Since by the invariant this edge induces an s-t cut with capacity $f(x_j, x_{j+1})$ we get $f(s,t) \le f(x_j, x_{j+1}) = f_T(s,t)$.

Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between s and t in T.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in G (which separates s and t).
- Since, we can send a flow of value $f(x_j, x_{j+1})$ btw. s and t, this is an s-t mincut (cut property).

Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between s and t in T.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in G (which separates s and t).
- Since, we can send a flow of value $f(x_j, x_{j+1})$ btw. s and t, this is an s-t mincut (cut property).

Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between s and t in T.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in G (which separates s and t).
- Since, we can send a flow of value $f(x_j, x_{j+1})$ btw. s and t, this is an s-t mincut (cut property).

Analysis

- Hence, $f_T(s,t) = f(s,t)$ (flow equivalence).
- The edge $\{x_j, x_{j+1}\}$ is a mincut between s and t in T.
- By invariant, it forms a cut with capacity $f(x_j, x_{j+1})$ in G (which separates s and t).
- Since, we can send a flow of value $f(x_j, x_{j+1})$ btw. s and t, this is an s-t mincut (cut property).

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a,b)$ we can simply choose a and b as representatives.

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a, b)$ we can simply choose a and b as representatives.

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a,b)$ we can simply choose a and b as representatives.

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a, b)$ we can simply choose a and b as representatives.

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a, b)$ we can simply choose a and b as representatives.

Proof of Invariant

The invariant obviously holds at the beginning of the algorithm.

Now, we show that it holds after a split-operation provided that it was true before the operation.

Let S_i denote our selected cluster with nodes a and b. Because of the invariant all edges leaving $\{S_i\}$ in T correspond to some mincuts.

Therefore, contracting the connected components does not change the mincut btw. a and b due to Lemma 94.

After the split we have to choose representatives for all edges. For the new edge $\{S_i^a, S_i^b\}$ with capacity $w(S_i^a, S_i^b) = f_H(a, b)$ we can simply choose a and b as representatives.

Proof of Invariant

For edges that are not incident to S_i we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep x and s as representatives.

Otherwise, we choose x and a as representatives. We need to show that $f(x, a) = f(x, s)$.

Proof of Invariant

For edges that are not incident to S_i we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep x and s as representatives.

Otherwise, we choose x and a as representatives. We need to show that $f(x, a) = f(x, s)$.

Proof of Invariant

For edges that are not incident to S_i we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep x and s as representatives.

Otherwise, we choose x and a as representatives. We need to show that $f(x, a) = f(x, s)$.

Proof of Invariant

For edges that are not incident to S_i we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep x and s as representatives.

Otherwise, we choose x and a as representatives. We need to show that $f(x, a) = f(x, s)$.

Proof of Invariant

For edges that are not incident to S_i we do not need to change representatives as the neighbouring sets do not change.

Consider an edge $\{X, S_i\}$, and suppose that before the split it used representatives $x \in X$, and $s \in S_i$. Assume that this edge is replaced by $\{X, S_i^a\}$ in the new tree (the case when it is replaced by $\{X, S_i^b\}$ is analogous).

If $s \in S_i^a$ we can keep x and s as representatives.

Otherwise, we choose x and a as representatives. We need to show that $f(x, a) = f(x, s)$.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \le f(x,s)$.

The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \ge \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \ge f(s,x)$.

Also, $f'(a,v_B) \ge f(a,b) \ge f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \geq f(s,x)$.

Also, $f'(a,v_B) \geq f(a,b) \geq f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \le f(x,s)$.

The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \ge \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \ge f(s,x)$.

Also, $f'(a,v_B) \ge f(a,b) \ge f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \geq f(s,x)$.

Also, $f'(a,v_B) \geq f(a,b) \geq f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \leq f(x,s)$.

The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \geq \min\{f'(x,v_B), f'(v_B,a)\}$.

Since $s \in B$ we have $f'(v_B,x) \geq f(s,x)$.

Also, $f'(a,v_B) \geq f(a,b) \geq f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

Proof of Invariant

Because the invariant was true before the split we know that the edge $\{X, S_i\}$ induces a cut in G of capacity $f(x,s)$. Since, x and a are on opposite sides of this cut, we know that $f(x,a) \le f(x,s)$.

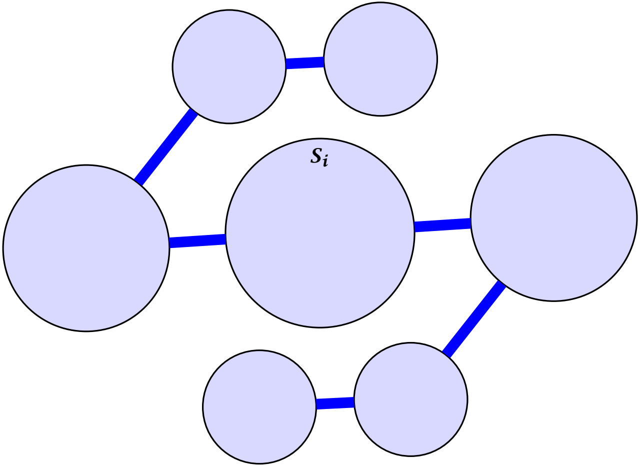
The set B forms a mincut separating a from b. Contracting all nodes in this set gives a new graph G' where the set B is represented by node v_B. Because of Lemma 94 we know that $f'(x,a) = f(x,a)$ as $x, a \notin B$.

We further have $f'(x,a) \ge \min\{f'(x,v_B), f'(v_B,a)\}$.

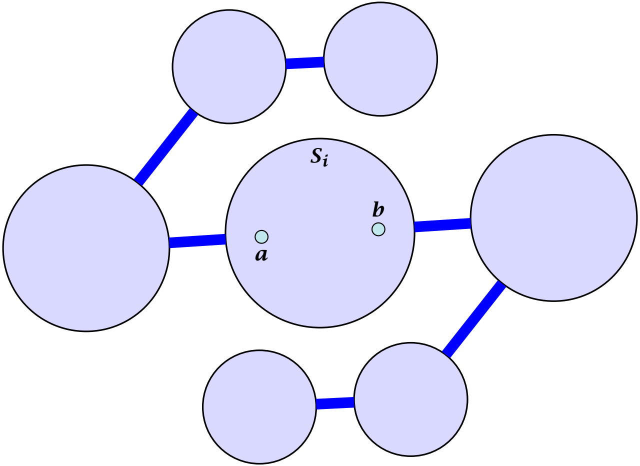
Since $s \in B$ we have $f'(v_B,x) \ge f(s,x)$.

Also, $f'(a,v_B) \ge f(a,b) \ge f(x,s)$ since the a-b cut that splits S_i into S_i^a and S_i^b also separates s and x.

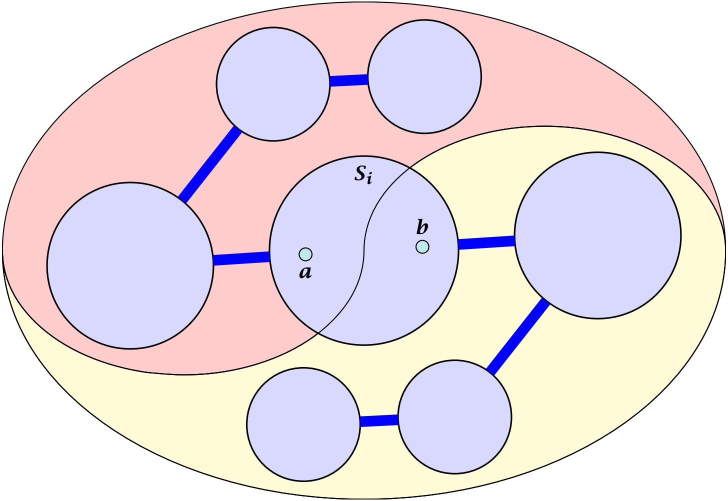
Analysis



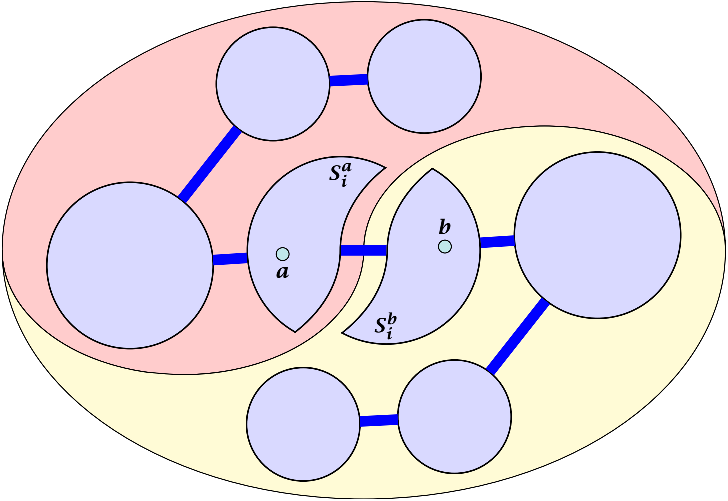
Analysis



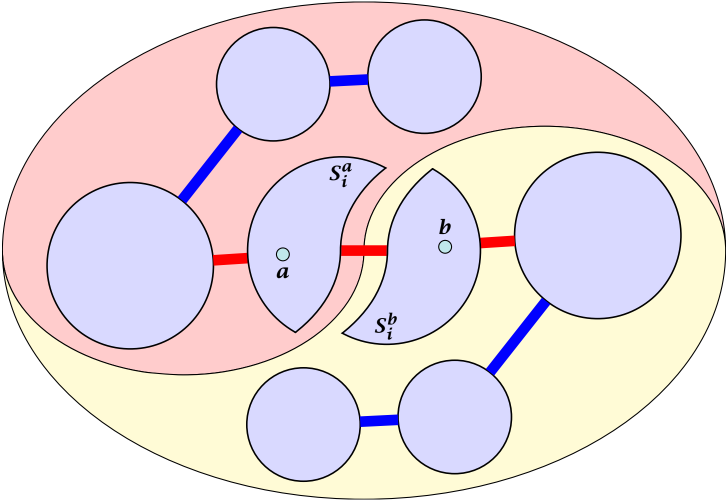
Analysis



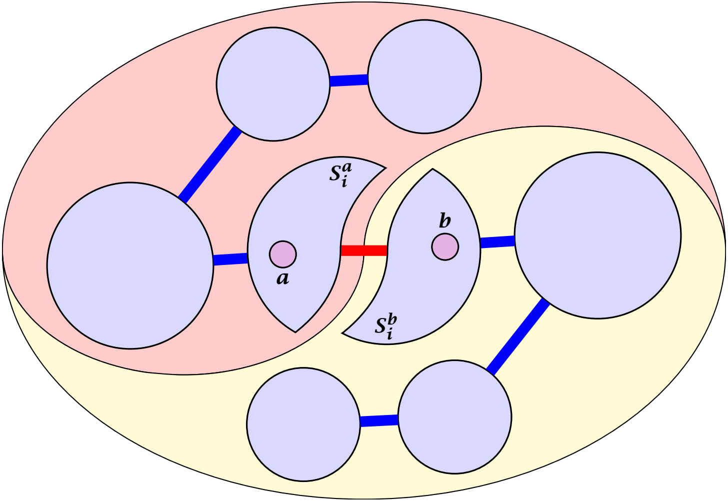
Analysis



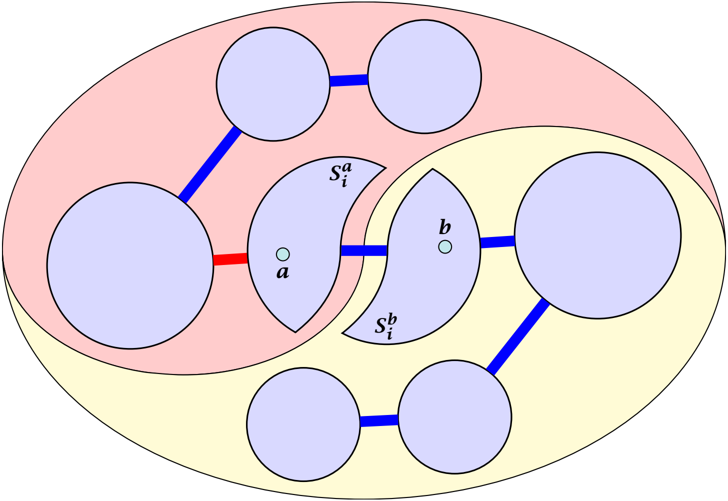
Analysis



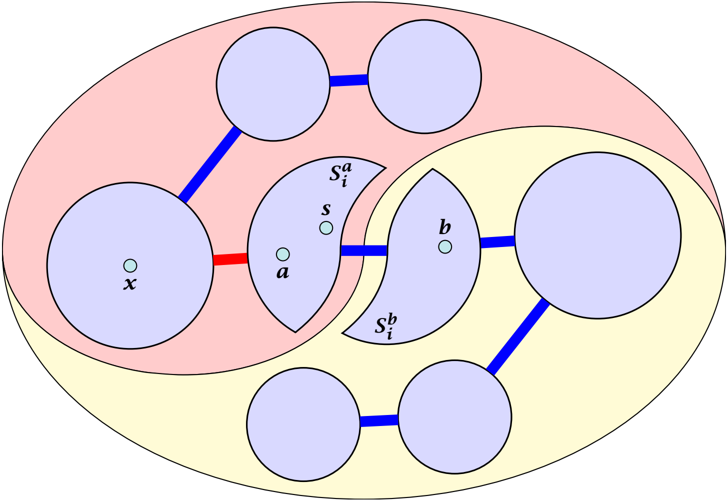
Analysis



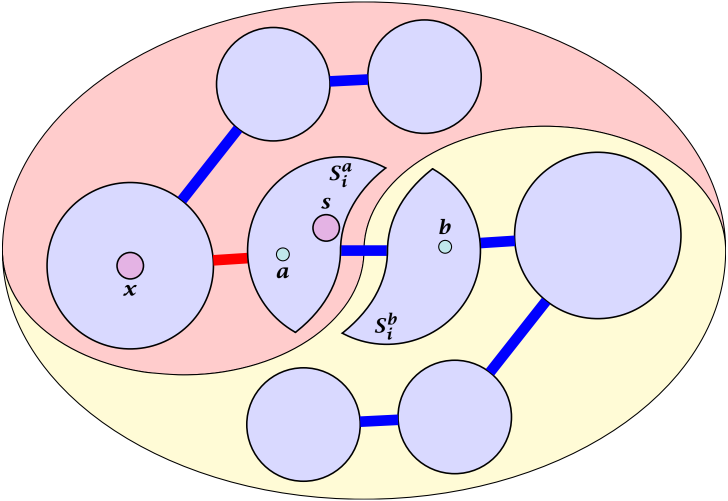
Analysis



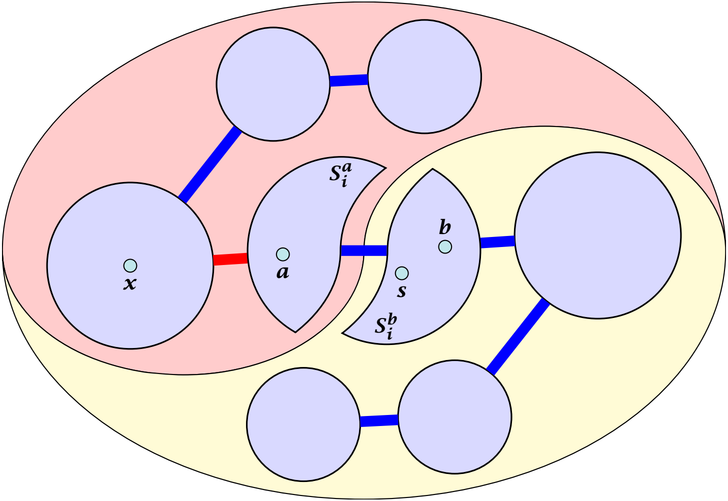
Analysis



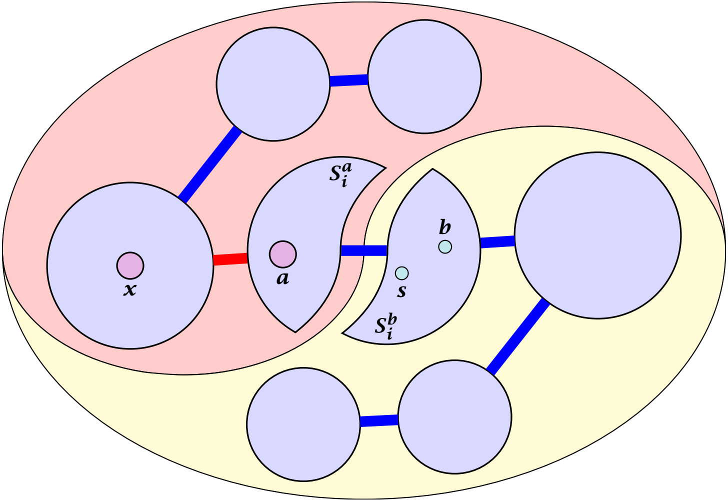
Analysis



Analysis



Analysis

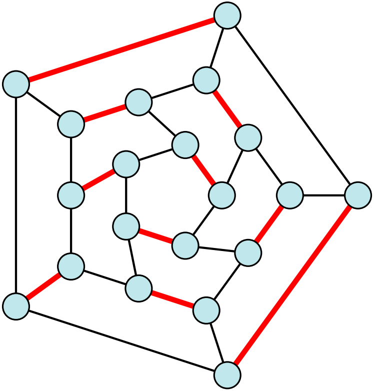


Part V

Matchings

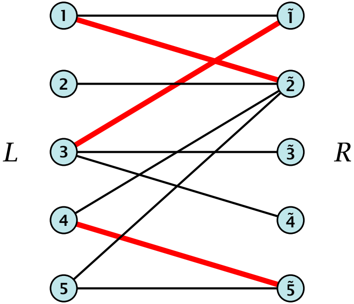
Matching

- Input: undirected graph $G = (V, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



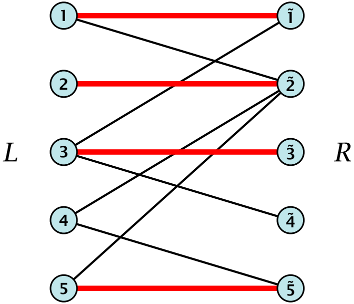
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



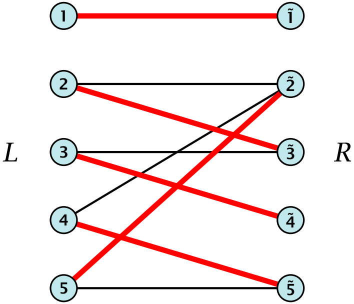
Bipartite Matching

- Input: undirected, bipartite graph $G = (L \uplus R, E)$.
- $M \subseteq E$ is a matching if each node appears in at most one edge in M.
- Maximum Matching: find a matching of maximum cardinality



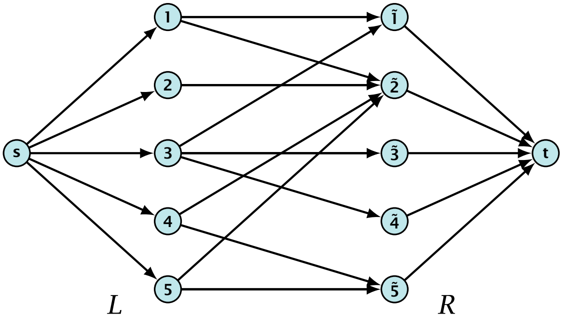
Bipartite Matching

- A matching M is perfect if it is of cardinality $|M| = |V|/2$.
- For a bipartite graph $G = (L \uplus R, E)$ this means $|M| = |L| = |R| = n$.



19 Bipartite Matching via Flows

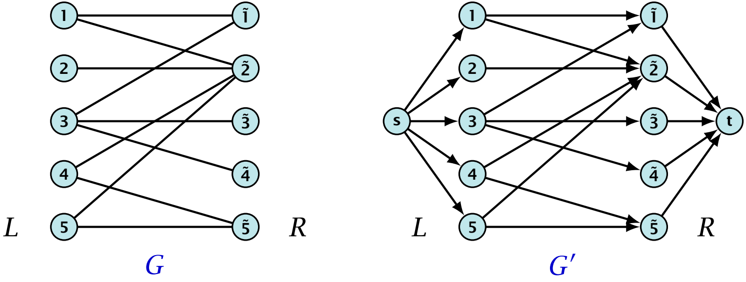
- Input: undirected, bipartite graph $G = (L \uplus R \uplus \{s,t\}, E')$.
- Direct all edges from L to R.
- Add source s and connect it to all nodes on the left.
- Add t and connect all nodes on the right to t.
- All edges have unit capacity.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

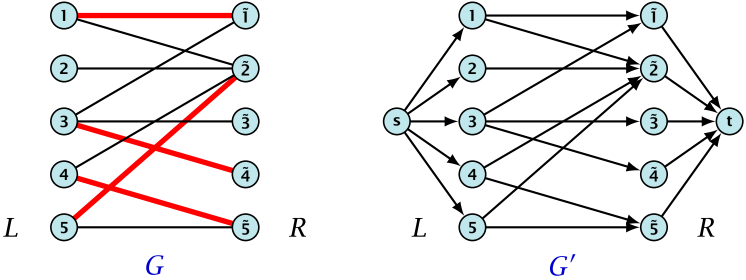
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

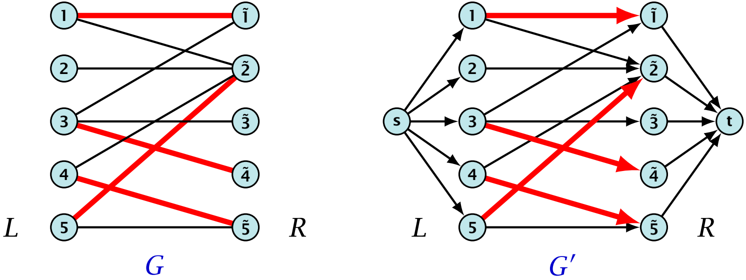
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

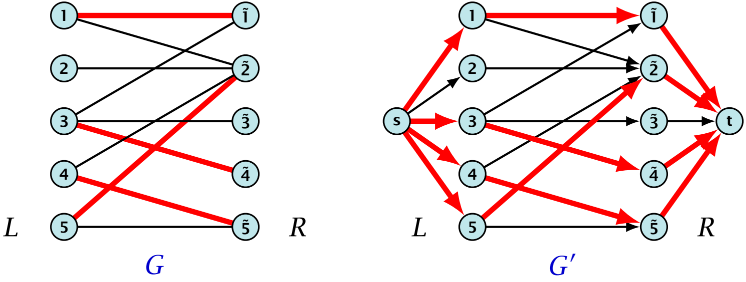
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in $G \leq$ value of maxflow in G'

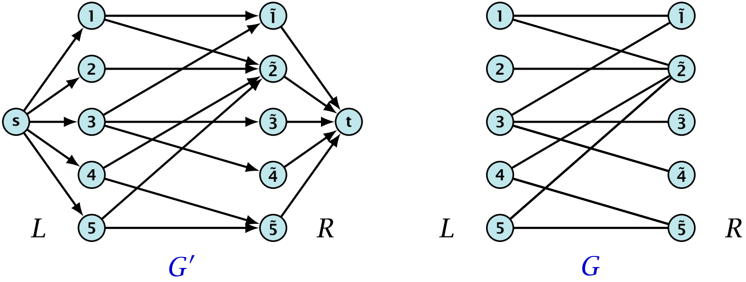
- Given a maximum matching M of cardinality k.
- Consider flow f that sends one unit along each of k paths.
- f is a flow and has cardinality k.



Proof

Max cardinality matching in G ≥ value of maxflow in G'

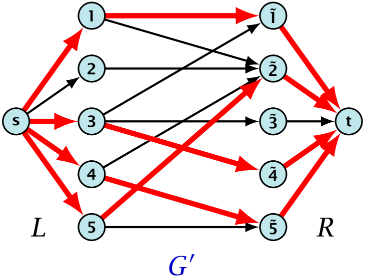
- Let f be a maxflow in G' of value k
- Integrality theorem ⇒ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

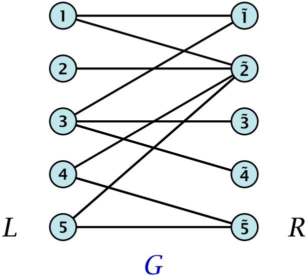


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem ⇒ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

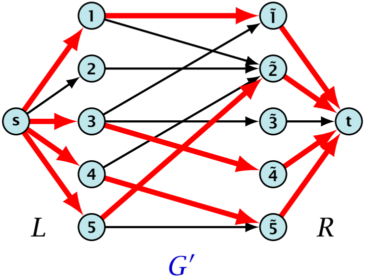


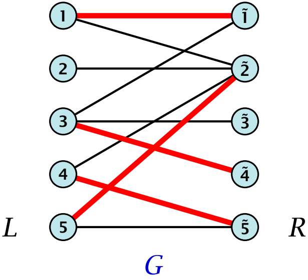


Proof

Max cardinality matching in G ≥ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem ⇒ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.

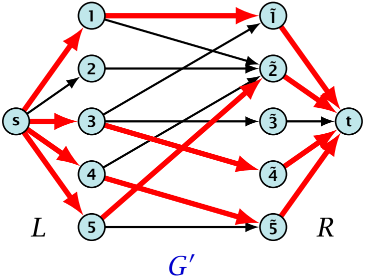


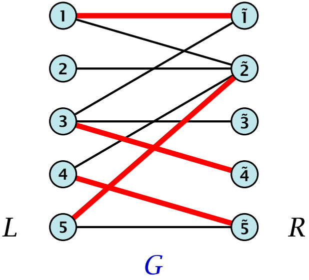


Proof

Max cardinality matching in $G \geq$ value of maxflow in G'

- Let f be a maxflow in G' of value k
- Integrality theorem $\Rightarrow$ k integral; we can assume f is 0/1.
- Consider M= set of edges from L to R with $f(e) = 1$.
- Each node in L and R participates in at most one edge in M.
- $|M| = k$, as the flow must use at least k middle edges.





19 Bipartite Matching via Flows

Which flow algorithm to use?

- Generic augmenting path: $\mathcal{O}(m\,\mathrm{val}(f^*)) = \mathcal{O}(mn)$.
- Capacity scaling: $\mathcal{O}(m^2 \log C) = \mathcal{O}(m^2)$.

20 Augmenting Paths for Matchings

Definitions.

- Given a matching M in a graph G, a vertex that is not incident to any edge of M is called a free vertex w.r..t. M.
- For a matching M a path P in G is called an alternating path if edges in M alternate with edges not in M.
- An alternating path is called an augmenting path for matching M if it ends at distinct free vertices.

Theorem 95

A matching M is a maximum matching if and only if there is no augmenting path w.r.t. M.

20 Augmenting Paths for Matchings

Definitions.

- Given a matching M in a graph G, a vertex that is not incident to any edge of M is called a free vertex w.r..t. M.
- For a matching M a path P in G is called an alternating path if edges in M alternate with edges not in M.
- An alternating path is called an augmenting path for matching M if it ends at distinct free vertices.

Theorem 95

A matching M is a maximum matching if and only if there is no augmenting path w.r.t. M.

20 Augmenting Paths for Matchings

Definitions.

- Given a matching M in a graph G, a vertex that is not incident to any edge of M is called a free vertex w. r. .t. M.
- For a matching M a path P in G is called an alternating path if edges in M alternate with edges not in M.
- An alternating path is called an augmenting path for matching M if it ends at distinct free vertices.

Theorem 95

A matching M is a maximum matching if and only if there is no augmenting path w. r. t. M.

20 Augmenting Paths for Matchings

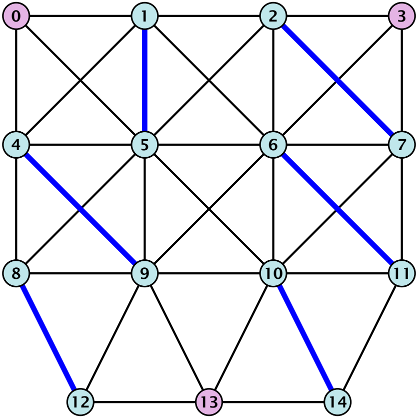
Definitions.

- Given a matching M in a graph G, a vertex that is not incident to any edge of M is called a free vertex w. r. .t. M.
- For a matching M a path P in G is called an alternating path if edges in M alternate with edges not in M.
- An alternating path is called an augmenting path for matching M if it ends at distinct free vertices.

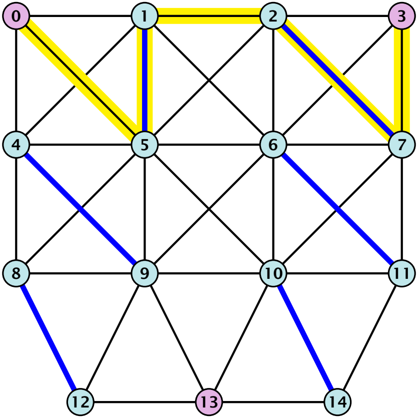
Theorem 95

A matching M is a maximum matching if and only if there is no augmenting path w. r. t. M.

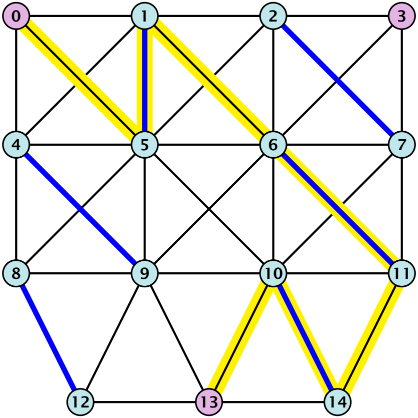
Augmenting Paths in Action



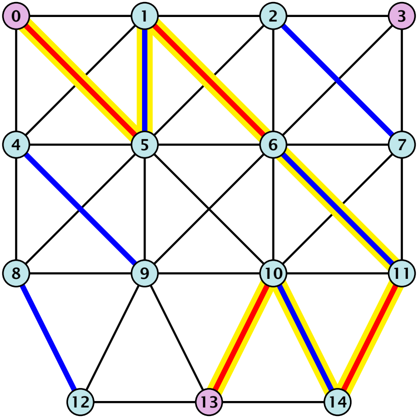
Augmenting Paths in Action



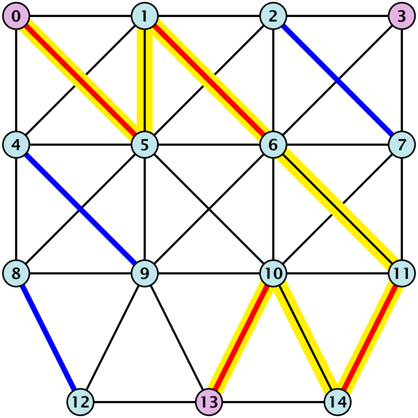
Augmenting Paths in Action



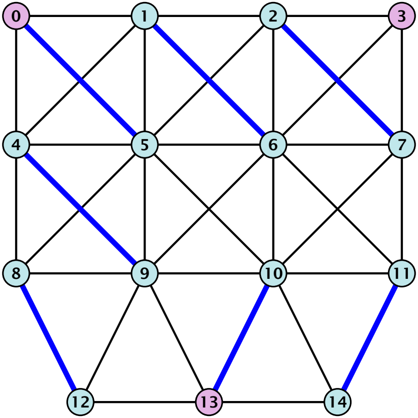
Augmenting Paths in Action



Augmenting Paths in Action



Augmenting Paths in Action



20 Augmenting Paths for Matchings

Proof.

- ⇒ If M is maximum there is no augmenting path P, because we could switch matching and non-matching edges along P. This gives matching $M' = M \oplus P$ with larger cardinality.
- ⇐ Suppose there is a matching M' with larger cardinality. Consider the graph H with edge-set $M' \oplus M$ (i.e., only edges that are in either M or M' but not in both).

  Each vertex can be incident to at most two edges (one from M and one from M'). Hence, the connected components are alternating cycles or alternating path.

  As $|M'| > |M|$ there is one connected component that is a path P for which both endpoints are incident to edges from M'. P is an alternating path.

20 Augmenting Paths for Matchings

Proof.

- ⇒ If M is maximum there is no augmenting path P, because we could switch matching and non-matching edges along P. This gives matching $M' = M \oplus P$ with larger cardinality.
- ⇐ Suppose there is a matching M' with larger cardinality. Consider the graph H with edge-set $M' \oplus M$ (i.e., only edges that are in either M or M' but not in both).

  Each vertex can be incident to at most two edges (one from M and one from M'). Hence, the connected components are alternating cycles or alternating path.

  As $|M'| > |M|$ there is one connected component that is a path P for which both endpoints are incident to edges from M'. P is an alternating path.

20 Augmenting Paths for Matchings

Proof.

- ⇒ If M is maximum there is no augmenting path P, because we could switch matching and non-matching edges along P. This gives matching $M' = M \oplus P$ with larger cardinality.
- ⇐ Suppose there is a matching M' with larger cardinality. Consider the graph H with edge-set $M' \oplus M$ (i.e., only edges that are in either M or M' but not in both).

  Each vertex can be incident to at most two edges (one from M and one from M'). Hence, the connected components are alternating cycles or alternating path.

  As $|M'| > |M|$ there is one connected component that is a path P for which both endpoints are incident to edges from M'. P is an alternating path.

20 Augmenting Paths for Matchings

Proof.

- ⇒ If M is maximum there is no augmenting path P, because we could switch matching and non-matching edges along P. This gives matching $M' = M \oplus P$ with larger cardinality.
- ⇐ Suppose there is a matching M' with larger cardinality. Consider the graph H with edge-set $M' \oplus M$ (i.e., only edges that are in either M or M' but not in both).

  Each vertex can be incident to at most two edges (one from M and one from M'). Hence, the connected components are alternating cycles or alternating path.

  As $|M'| > |M|$ there is one connected component that is a path P for which both endpoints are incident to edges from M'. P is an alternating path.

20 Augmenting Paths for Matchings

Algorithmic idea:
As long as you find an augmenting path augment your matching using this path. When you arrive at a matching for which no augmenting path exists you have a maximum matching.

Theorem 96
Let G be a graph, M a matching in G, and let u be a free vertex w.r.t. M. Further let P denote an augmenting path w.r.t. M and let $M' = M \oplus P$ denote the matching resulting from augmenting M with P. If there was no augmenting path starting at u in M then there is no augmenting path starting at u in M'.

20 Augmenting Paths for Matchings

Algorithmic idea:
As long as you find an augmenting path augment your matching using this path. When you arrive at a matching for which no augmenting path exists you have a maximum matching.

Theorem 96

Let G be a graph, M a matching in G, and let u be a free vertex w.r.t. M. Further let P denote an augmenting path w.r.t. M and let $M' = M \oplus P$ denote the matching resulting from augmenting M with P. If there was no augmenting path starting at u in M then there is no augmenting path starting at u in M'.

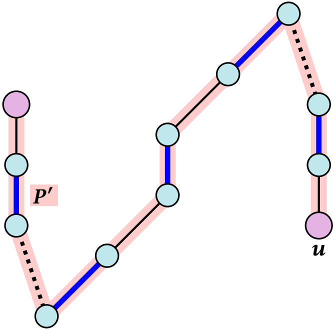
20 Augmenting Paths for Matchings

Proof

20 Augmenting Paths for Matchings

Proof

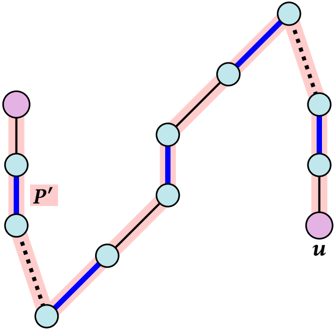
- Assume there is an augmenting path P' w.r.t. M' starting at u.



20 Augmenting Paths for Matchings

Proof

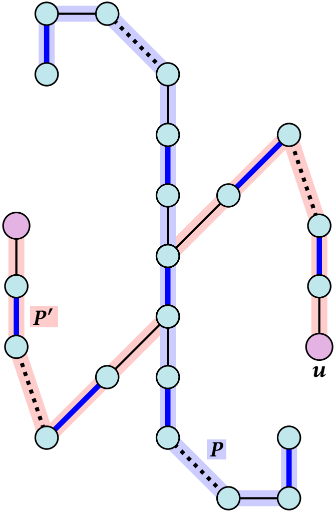
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).



20 Augmenting Paths for Matchings

Proof

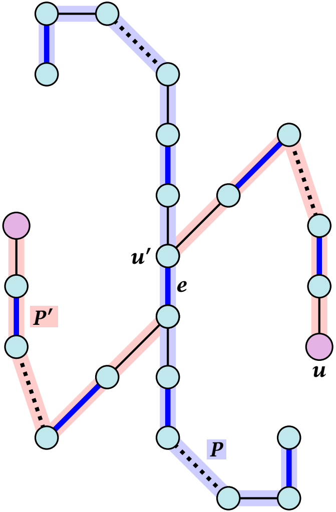
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M ($\lightning$).



20 Augmenting Paths for Matchings

Proof

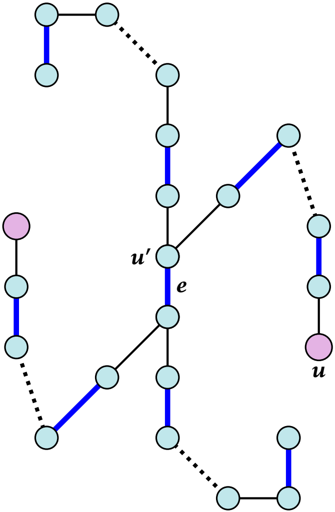
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.



20 Augmenting Paths for Matchings

Proof

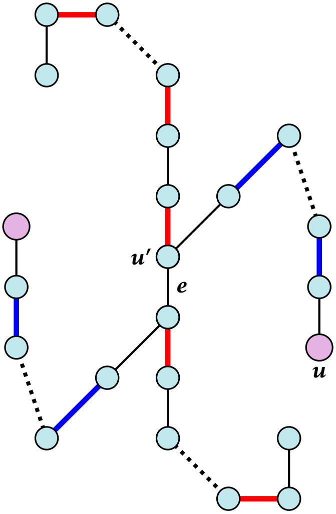
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.



20 Augmenting Paths for Matchings

Proof

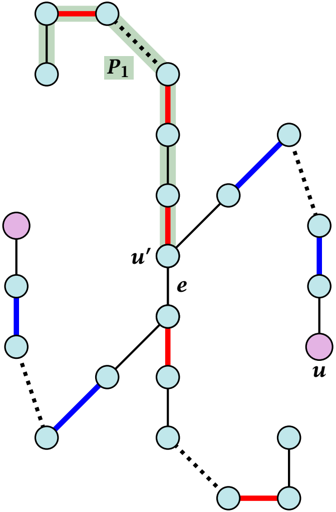
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.



20 Augmenting Paths for Matchings

Proof

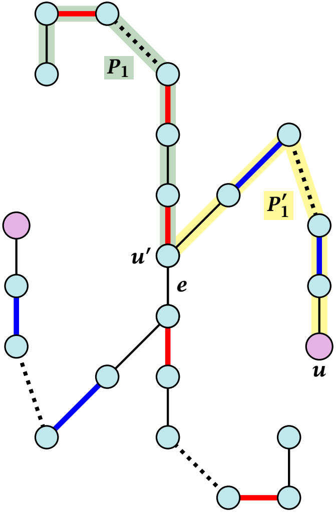
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.
- u' splits P into two parts one of which does not contain e. Call this part P_1. Denote the sub-path of P' from u to u' with P'_1.



20 Augmenting Paths for Matchings

Proof

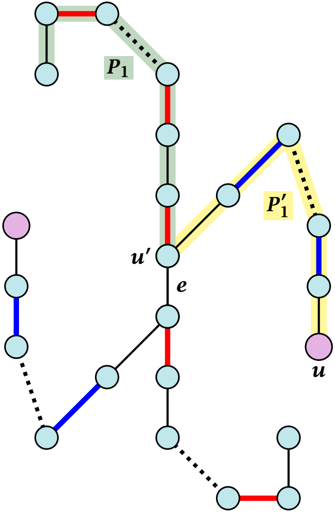
- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.
- u' splits P into two parts one of which does not contain e. Call this part P_1. Denote the sub-path of P' from u to u' with P_1'.



20 Augmenting Paths for Matchings

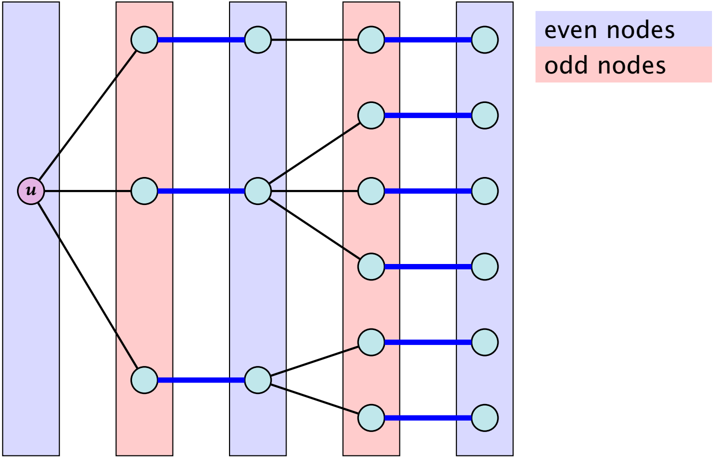
Proof

- Assume there is an augmenting path P' w.r.t. M' starting at u.
- If P' and P are node-disjoint, P' is also augmenting path w.r.t. M (↯).
- Let u' be the first node on P' that is in P, and let e be the matching edge from M' incident to u'.
- u' splits P into two parts one of which does not contain e. Call this part P_1. Denote the sub-path of P' from u to u' with P_1'.
- $P_1 \circ P_1'$ is augmenting path in M (↯).



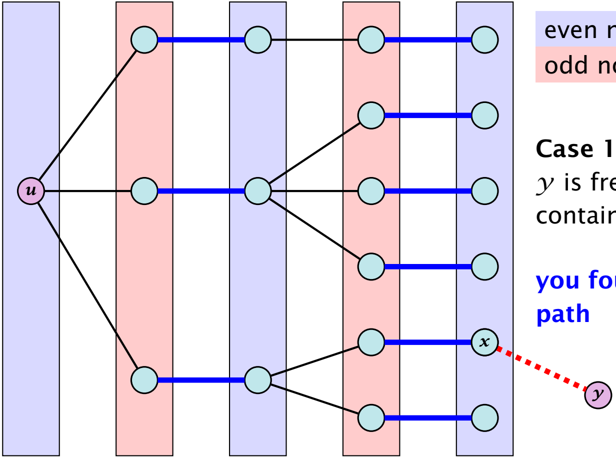
How to find an augmenting path?

Construct an alternating tree.



How to find an augmenting path?

Construct an alternating tree.



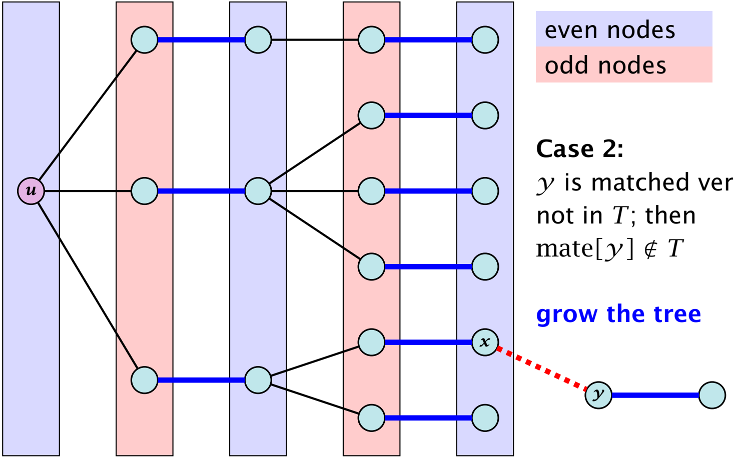
even nodes
odd nodes

Case 1:
y is free vertex not contained in T

you found alternating path

How to find an augmenting path?

Construct an alternating tree.



even nodes

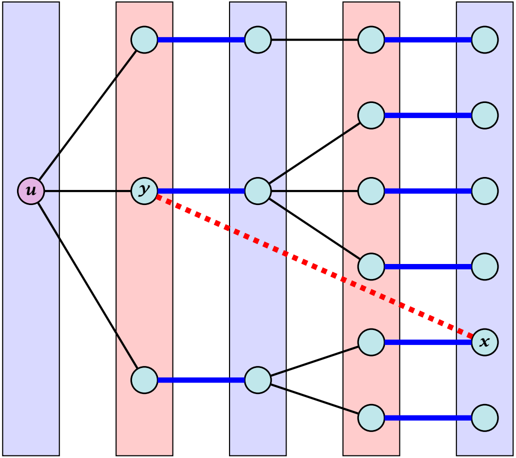
odd nodes

Case 2:
y is matched vertex not in T; then $\mathrm{mate}[y] \notin T$

grow the tree

How to find an augmenting path?

Construct an alternating tree.



even nodes

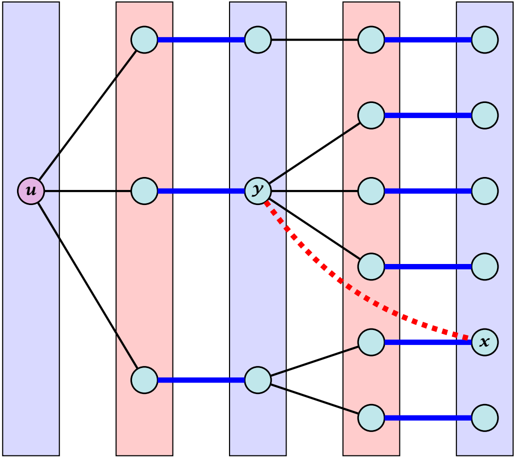
odd nodes

Case 3:
y is already contained
in T as an odd vertex

ignore successor y

How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

Case 4:
y is already contained in T as an even vertex

can't ignore y

does not happen in bipartite graphs

Algorithm 1 BiMatch$(G, match)$

```
for $x \in V$ do $mate[x] \leftarrow 0$;
$r \leftarrow 0$; $free \leftarrow n$;
while $free \geq 1$ and $r < n$ do
    $r \leftarrow r + 1$
    if $mate[r] = 0$ then
        for $i = 1$ to $m$ do $parent[i'] \leftarrow 0$
        $Q \leftarrow \emptyset$; $Q.\mathrm{append}(r)$; $aug \leftarrow \mathrm{false}$;
        while $aug = \mathrm{false}$ and $Q \neq \emptyset$ do
            $x \leftarrow Q.\mathrm{dequeue}()$;
            if $\exists y \in A_x$: $mate[y] = 0$ then
                augment$(mate, parent, y)$;
                $aug \leftarrow \mathrm{true}$; $free \leftarrow free - 1$;
            else
                if $parent[y] = 0$ then
                    $parent[y] \leftarrow x$;
                    $Q.\mathrm{enqueue}(y)$;
```

graph $G = (S \cup S', E)$;
$S = \{1, \dots, n\}$;
$S = \{1', \dots, n'\}$

initial matching empty

free: number of unmatched nodes in S

r: root of current tree

if r is unmatched start tree construction

initialize empty tree

no augmen. path but unexamined leaves

free neighbour found

add new node y to Q

21 Weighted Bipartite Matching

Weighted Bipartite Matching/Assignment

- Input: undirected, bipartite graph $G = L \cup R, E$.
- an edge $e = (\ell, r)$ has weight $w_e \geq 0$
- find a matching of maximum weight, where the weight of a matching is the sum of the weights of its edges

Simplifying Assumptions (wlog [why?]):

- assume that $|L| = |R| = n$
- assume that there is an edge between every pair of nodes $(\ell, r) \in V \times V$

Weighted Bipartite Matching

Theorem 97 (Halls Theorem)

A bipartite graph $G = (L \cup R, E)$ has a perfect matching if and only if for all sets $S \subseteq L$, $|\Gamma(S)| \geq |S|$, where $\Gamma(S)$ denotes the set of nodes in R that have a neighbour in S.

Halls Theorem

Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \triangleq L \cap S$ and $R_S \triangleq R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
    - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
    - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
    - This gives $R_S \geq |\Gamma(L_S)|$.
    - The size of the cut is $|L| - |L_S| + |R_S|$.
    - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neighbours.
- $\Rightarrow$ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neighbours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- ⇐ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neigbhours.
- ⇒ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Halls Theorem

Proof:

- $\Leftarrow$ Of course, the condition is necessary as otherwise not all nodes in S could be matched to different neigbhours.
- $\Rightarrow$ For the other direction we need to argue that the minimum cut in the graph G' is at least $|L|$.
  - Let S denote a minimum cut and let $L_S \overset{\text{def}}{=} L \cap S$ and $R_S \overset{\text{def}}{=} R \cap S$ denote the portion of S inside L and R, respectively.
  - Clearly, all neighbours of nodes in L_S have to be in S, as otherwise we would cut an edge of infinite capacity.
  - This gives $R_S \geq |\Gamma(L_S)|$.
  - The size of the cut is $|L| - |L_S| + |R_S|$.
  - Using the fact that $|\Gamma(L_S)| \geq L_S$ gives that this is at least $|L|$.

Algorithm Outline

Idea:

We introduce a node weighting $\vec{x}$. Let for a node $v \in V$, $x_v \geq 0$ denote the weight of node v.

- Suppose that the node weights dominate the edge weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- Let $H(\vec{x})$ denote the subgraph of G that only contains edges that are tight w.r.t. the node weighting $\vec{x}$, i.e. edges $e = (u, v)$ for which $w_e = (u, v)$.
- Try to compute a perfect matching in the subgraph $H(\vec{x})$. If you are successful you found an optimal matching.

Algorithm Outline

Idea:
We introduce a node weighting $\vec{x}$. Let for a node $v \in V$, $x_v \geq 0$ denote the weight of node v.

- Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- Let $H(\vec{x})$ denote the subgraph of G that only contains edges that are tight w.r.t. the node weighting $\vec{x}$, i.e. edges $e = (u, v)$ for which $w_e = (u, v)$.
- Try to compute a perfect matching in the subgraph $H(\vec{x})$. If you are successful you found an optimal matching.

Algorithm Outline

Idea:

We introduce a node weighting $\vec{x}$. Let for a node $v \in V$, $x_v \geq 0$ denote the weight of node v.

- Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u,v).$$

- Let $H(\vec{x})$ denote the subgraph of G that only contains edges that are tight w.r.t. the node weighting $\vec{x}$, i.e. edges $e = (u,v)$ for which $w_e = (u,v)$.
- Try to compute a perfect matching in the subgraph $H(\vec{x})$. If you are successful you found an optimal matching.

Algorithm Outline

Idea:

We introduce a node weighting $\vec{x}$. Let for a node $v \in V$, $x_v \geq 0$ denote the weight of node v.

- Suppose that the node weights dominate the edge-weights in the following sense:

$$x_u + x_v \geq w_e \text{ for every edge } e = (u, v).$$

- Let $H(\vec{x})$ denote the subgraph of G that only contains edges that are tight w.r.t. the node weighting $\vec{x}$, i.e. edges $e = (u, v)$ for which $w_e = (u, v)$.
- Try to compute a perfect matching in the subgraph $H(\vec{x})$. If you are successful you found an optimal matching.

Algorithm Outline

Reason:

- The weight of your matching M^* is

$$\sum_{(u,v)\in M^*} w_{(u,v)} = \sum_{(u,v)\in M^*} (x_u + x_v) = \sum_v x_v \ .$$

- Any other matching M has

$$\sum_{(u,v)\in M} w_{(u,v)} \leq \sum_{(u,v)\in M} (x_u + x_v) \leq \sum_v x_v \ .$$

Algorithm Outline

What if you don't find a perfect matching?

Then, Halls theorem guarantees you that there is a set $S \subseteq L$, with $|\Gamma(S)| < |S|$, where Γ denotes the neighbourhood w.r.t. the subgraph $H(\vec{x})$.

Idea: reweight such that:

- the total weight assigned to nodes decreases
- the weight function still dominates the edge-weights

If we can do this we have an algorithm that terminates with an optimal solution (we analyze the running time later).

Algorithm Outline

What if you don't find a perfect matching?

Then, Halls theorem guarantees you that there is a set $S \subseteq L$, with $|\Gamma(S)| < |S|$, where Γ denotes the neighbourhood w.r.t. the subgraph $H(\vec{x})$.

Idea: reweight such that:

- the total weight assigned to nodes decreases
- the weight function still dominates the edge-weights

If we can do this we have an algorithm that terminates with an optimal solution (we analyze the running time later).

Algorithm Outline

What if you don't find a perfect matching?

Then, Halls theorem guarantees you that there is a set $S \subseteq L$, with $|\Gamma(S)| < |S|$, where Γ denotes the neighbourhood w.r.t. the subgraph $H(\vec{x})$.

Idea: reweight such that:

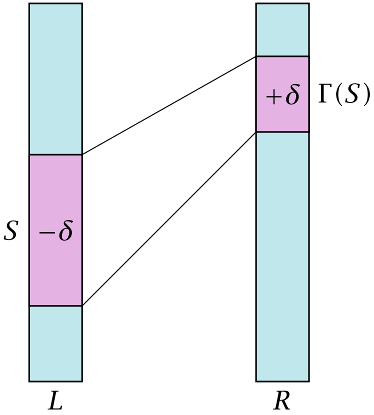
- the total weight assigned to nodes decreases
- the weight function still dominates the edge-weights

If we can do this we have an algorithm that terminates with an optimal solution (we analyze the running time later).

Changing Node Weights

Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in S by $-\delta$.

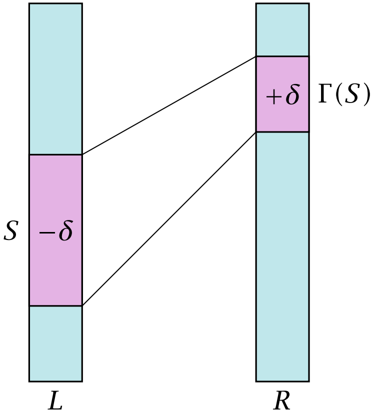
- Total node-weight decreases.
- Only edges from S to $R-\Gamma(S)$ decrease in their weight.
- Since, none of these edges is tight (otw. the edge would be contained in $H(\vec{x})$, and hence would go between S and $\Gamma(S)$) we can do this decrement for small enough $\delta > 0$ until a new edge gets tight.



Changing Node Weights

Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in S by $-\delta$.

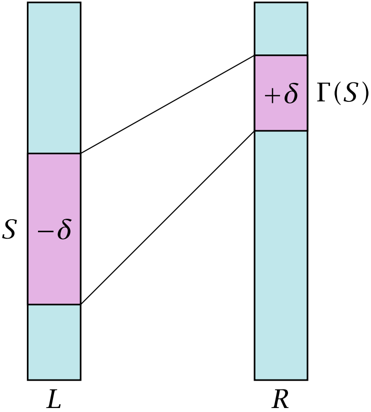
- Total node-weight decreases.
- Only edges from S to $R - \Gamma(S)$ decrease in their weight.
- Since, none of these edges is tight (otw. the edge would be contained in $H(\vec{x})$, and hence would go between S and $\Gamma(S)$) we can do this decrement for small enough $\delta > 0$ until a new edge gets tight.



Changing Node Weights

Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in S by $-\delta$.

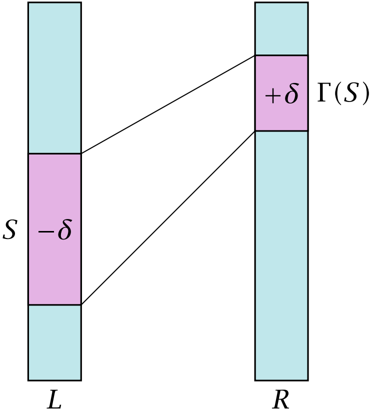
- Total node-weight decreases.
- Only edges from S to $R - \Gamma(S)$ decrease in their weight.
- Since, none of these edges is tight (otw. the edge would be contained in $H(\vec{x})$, and hence would go between S and $\Gamma(S)$) we can do this decrement for small enough $\delta > 0$ until a new edge gets tight.



Changing Node Weights

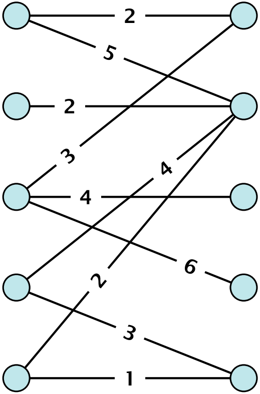
Increase node-weights in $\Gamma(S)$ by $+\delta$, and decrease the node-weights in S by $-\delta$.

- Total node-weight decreases.
- Only edges from S to $R - \Gamma(S)$ decrease in their weight.
- Since, none of these edges is tight (otw. the edge would be contained in $H(\vec{x})$, and hence would go between S and $\Gamma(S)$) we can do this decrement for small enough $\delta > 0$ until a new edge gets tight.



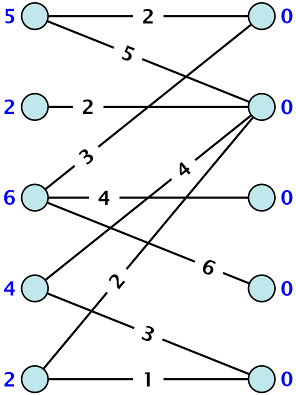
Weighted Bipartite Matching

Edges not drawn have weight 0.



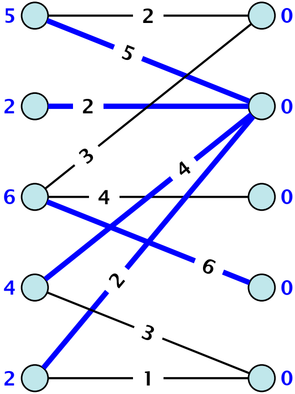
Weighted Bipartite Matching

Edges not drawn have weight 0.



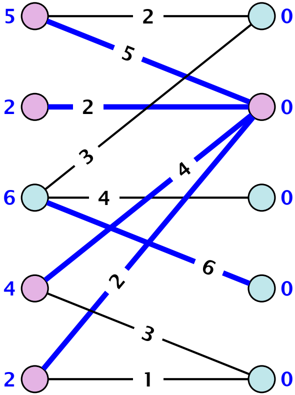
Weighted Bipartite Matching

Edges not drawn have weight 0.



Weighted Bipartite Matching

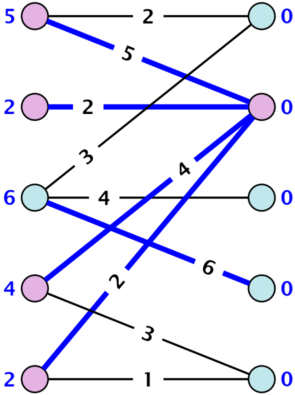
Edges not drawn have weight 0.



Weighted Bipartite Matching

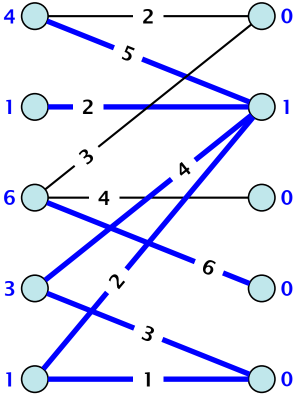
Edges not drawn have weight 0.

$\delta = 1$



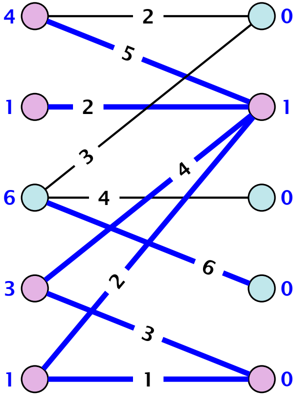
Weighted Bipartite Matching

Edges not drawn have weight 0.



Weighted Bipartite Matching

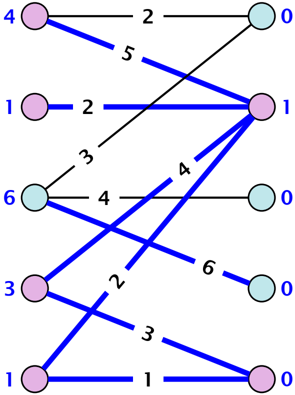
Edges not drawn have weight 0.



Weighted Bipartite Matching

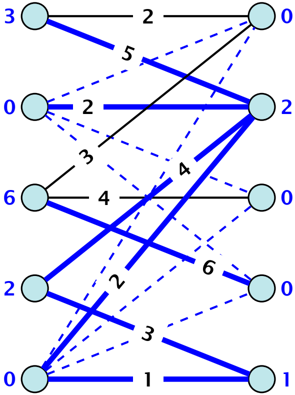
Edges not drawn have weight 0.

$\delta = 1$



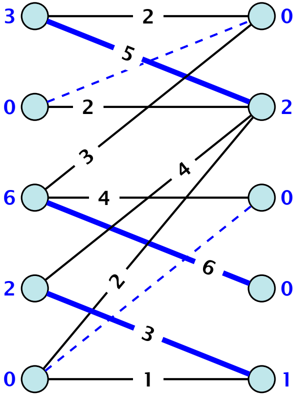
Weighted Bipartite Matching

Edges not drawn have weight 0.



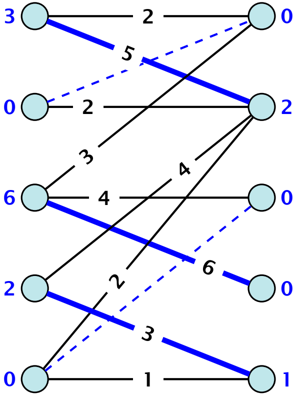
Weighted Bipartite Matching

Edges not drawn have weight 0.



Weighted Bipartite Matching

Edges not drawn have weight 0.



Analysis

How many iterations do we need?

- One reweighting step increases the number of edges out of S by at least one.
- Assume that we have a maximum matching that saturates the set $\Gamma(S)$, in the sense that every node in $\Gamma(S)$ is matched to a node in S (we will show that we can always find S and a matching such that this holds).
- This matching is still contained in the new graph, because all its edges either go between $\Gamma(S)$ and S or between $L - S$ and $R - \Gamma(S)$.
- Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

Analysis

How many iterations do we need?

- One reweighting step increases the number of edges out of S by at least one.
- Assume that we have a maximum matching that saturates the set $\Gamma(S)$, in the sense that every node in $\Gamma(S)$ is matched to a node in S (we will show that we can always find S and a matching such that this holds).
- This matching is still contained in the new graph, because all its edges either go between $\Gamma(S)$ and S or between $L - S$ and $R - \Gamma(S)$.
- Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

Analysis

How many iterations do we need?

- One reweighting step increases the number of edges out of S by at least one.
- Assume that we have a maximum matching that saturates the set $\Gamma(S)$, in the sense that every node in $\Gamma(S)$ is matched to a node in S (we will show that we can always find S and a matching such that this holds).
- This matching is still contained in the new graph, because all its edges either go between $\Gamma(S)$ and S or between $L - S$ and $R - \Gamma(S)$.
- Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

Analysis

How many iterations do we need?

- One reweighting step increases the number of edges out of S by at least one.
- Assume that we have a maximum matching that saturates the set $\Gamma(S)$, in the sense that every node in $\Gamma(S)$ is matched to a node in S (we will show that we can always find S and a matching such that this holds).
- This matching is still contained in the new graph, because all its edges either go between $\Gamma(S)$ and S or between $L - S$ and $R - \Gamma(S)$.
- Hence, reweighting does not decrease the size of a maximum matching in the tight sub-graph.

Analysis

- We will show that after at most n reweighting steps the size of the maximum matching can be increased by finding an augmenting path.
- This gives a polynomial running time.

Analysis

How do we find S?

- Start on the left and compute an alternating tree, starting at any free node u.
- If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at u).
- The set of even vertices is on the left and the set of odd vertices is on the right and contains all neighbours of even nodes.
- All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex u. Hence, $|V_{\text{odd}}| = |\Gamma(V_{\text{even}})| < |V_{\text{even}}|$, and all odd vertices are saturated in the current matching.

Analysis

How do we find S?

- Start on the left and compute an alternating tree, starting at any free node u.
- If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at u).
- The set of even vertices is on the left and the set of odd vertices is on the right and contains all neighbours of even nodes.
- All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex u. Hence, $|V_{\text{odd}}| = |\Gamma(V_{\text{even}})| < |V_{\text{even}}|$, and all odd vertices are saturated in the current matching.

Analysis

How do we find S?

- Start on the left and compute an alternating tree, starting at any free node u.
- If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at u).
- The set of even vertices is on the left and the set of odd vertices is on the right and contains all neighbours of even nodes.
- All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex u. Hence, $|V_{\text{odd}}| = |\Gamma(V_{\text{even}})| < |V_{\text{even}}|$, and all odd vertices are saturated in the current matching.

Analysis

How do we find S?

- Start on the left and compute an alternating tree, starting at any free node u.
- If this construction stops, there is no perfect matching in the tight subgraph (because for a perfect matching we need to find an augmenting path starting at u).
- The set of even vertices is on the left and the set of odd vertices is on the right and contains all neighbours of even nodes.
- All odd vertices are matched to even vertices. Furthermore, the even vertices additionally contain the free vertex u. Hence, $|V_{\text{odd}}| = |\Gamma(V_{\text{even}})| < |V_{\text{even}}|$, and all odd vertices are saturated in the current matching.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

Analysis

- The current matching does not have any edges from V_{odd} to outside of $L \setminus V_{\text{even}}$ (edges that may possibly deleted by changing weights).
- After changing weights, there is at least one more edge connecting V_{even} to a node outside of V_{odd}. After at most n reweights we can do an augmentation.
- A reweighting can be trivially performed in time $\mathcal{O}(n^2)$ (keeping track of the tight edges).
- An augmentation takes at most $\mathcal{O}(n)$ time.
- In total we otain a running time of $\mathcal{O}(n^4)$.
- A more careful implementation of the algorithm obtains a running time of $\mathcal{O}(n^3)$.

A Fast Matching Algorithm

Algorithm 54 Bimatch-Hopcroft-Karp(G)

```
M ← ∅
repeat
    let $\mathcal{P} = \{P_1, \dots, P_k\}$ be maximal set of
    vertex-disjoint, shortest augmenting path w.r.t. $M$.
    $M \leftarrow M \oplus (P_1 \cup \dots \cup P_k)$
until $\mathcal{P} = \emptyset$
return $M$
```

We call one iteration of the repeat-loop a phase of the algorithm.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \triangleq |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \overset{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \stackrel{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \overset{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \overset{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

Lemma 98

Given a matching M and a maximal matching M^* there exist $|M^*| - |M|$ vertex-disjoint augmenting path w.r.t. M.

Proof:

- Similar to the proof that a matching is optimal iff it does not contain an augmenting paths.
- Consider the graph $G = (V, M \oplus M^*)$, and mark edges in this graph blue if they are in M and red if they are in M^*.
- The connected components of G are cycles and paths.
- The graph contains $k \overset{\text{def}}{=} |M^*| - |M|$ more red edges than blue edges.
- Hence, there are at least k components that form a path starting and ending with a blue edge. These are augmenting paths w.r.t. M.

Analysis

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. M (let $\ell = |P_i|$).
- $M' \overset{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let P be an augmenting path in M'.

Lemma 99

The set $A \overset{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$ contains at least $(k+1)\ell$ edges.

Analysis

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. M (let $\ell = |P_i|$).
- $M' \overset{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let P be an augmenting path in M'.

Lemma 99

The set $A \overset{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$ *contains at least* $(k+1)\ell$ *edges.*

Analysis

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. M (let $\ell = |P_i|$).
- $M' \stackrel{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let P be an augmenting path in M'.

Lemma 99

The set $A \stackrel{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$ *contains at least* $(k+1)\ell$ *edges.*

Analysis

- Let $P_1, \dots, P_k$ be a maximal collection of vertex-disjoint, shortest augmenting paths w.r.t. M (let $\ell = |P_i|$).
- $M' \overset{\text{def}}{=} M \oplus (P_1 \cup \dots \cup P_k) = M \oplus P_1 \oplus \dots \oplus P_k$.
- Let P be an augmenting path in M'.

Lemma 99

The set $A \overset{\text{def}}{=} M \oplus (M' \oplus P) = (P_1 \cup \dots \cup P_k) \oplus P$ *contains at least* $(k+1)\ell$ *edges.*

Analysis

Proof.

- The set describes exactly the symmetric difference between matchings M and $M' \oplus P$.
- Hence, the set contains at least $k+1$ vertex-disjoint augmenting paths w.r.t. M as $|M'| = |M| + k + 1$.
- Each of these paths is of length at least ℓ.

Analysis

Proof.

- The set describes exactly the symmetric difference between matchings M and $M' \oplus P$.
- Hence, the set contains at least $k+1$ vertex-disjoint augmenting paths w.r.t. M as $|M'| = |M| + k + 1$.
- Each of these paths is of length at least ℓ.

Analysis

Proof.

- The set describes exactly the symmetric difference between matchings M and $M' \oplus P$.
- Hence, the set contains at least $k+1$ vertex-disjoint augmenting paths w.r.t. M as $|M'| = |M| + k + 1$.
- Each of these paths is of length at least ℓ.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \leq k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \leq |A| \leq k\ell + |P| - 1$, and hence $|P| \geq \ell + 1$.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \leq k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \leq |A| \leq k\ell + |P| - 1$, and hence $|P| \geq \ell + 1$.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \le k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \le |A| \le k\ell + |P| - 1$, and hence $|P| \ge \ell + 1$.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \leq k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \leq |A| \leq k\ell + |P| - 1$, and hence $|P| \geq \ell + 1$.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \le k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \le |A| \le k\ell + |P| - 1$, and hence $|P| \ge \ell + 1$.

Analysis

Lemma 100

P is of length at least $\ell + 1$. This shows that the length of a shortest augmenting path increases between two phases of the Hopcroft-Karp algorithm.

Proof.

- If P does not intersect any of the $P_1, \dots, P_k$, this follows from the maximality of the set $\{P_1, \dots, P_k\}$.
- Otherwise, at least one edge from P coincides with an edge from paths $\{P_1, \dots, P_k\}$.
- This edge is not contained in A.
- Hence, $|A| \leq k\ell + |P| - 1$.
- The lower bound on $|A|$ gives $(k+1)\ell \leq |A| \leq k\ell + |P| - 1$, and hence $|P| \geq \ell + 1$.

Analysis

If the shortest augmenting path w.r.t. a matching M has ℓ edges then the cardinality of the maximum matching is of size at most $|M + |\frac{|V|}{\ell+1}$.

Proof.

The symmetric difference between M and M^* contains $|M^*| - |M|$ vertex-disjoint augmenting paths. Each of these paths contains at least $\ell + 1$ vertices. Hence, there can be at most $\frac{|V|}{\ell+1}$ of them.

Analysis

If the shortest augmenting path w.r.t. a matching M has ℓ edges then the cardinality of the maximum matching is of size at most $|M + |\frac{|V|}{\ell+1}$.

Proof.

The symmetric difference between M and M^* contains $|M^*| - |M|$ vertex-disjoint augmenting paths. Each of these paths contains at least $\ell + 1$ vertices. Hence, there can be at most $\frac{|V|}{\ell+1}$ of them.

Analysis

Lemma 101

The Hopcroft-Karp algorithm requires at most $2\sqrt{|V|}$ *phases.*

Proof.

- After iteration $\lfloor\sqrt{|V|}\rfloor$ the length of a shortest augmenting path must be at least $\lfloor\sqrt{|V|}\rfloor + 1 \geq \sqrt{|V|}$.
- Hence, there can be at most $|V|/(\sqrt{|V|}+1) \leq \sqrt{|V|}$ additional augmentations.

Analysis

Lemma 101

The Hopcroft-Karp algorithm requires at most $2\sqrt{|V|}$ *phases.*

Proof.

- After iteration $\lfloor\sqrt{|V|}\rfloor$ the length of a shortest augmenting path must be at least $\lfloor\sqrt{|V|}\rfloor + 1 \geq \sqrt{|V|}$.
- Hence, there can be at most $|V|/(\sqrt{|V|} + 1) \leq \sqrt{|V|}$ additional augmentations.

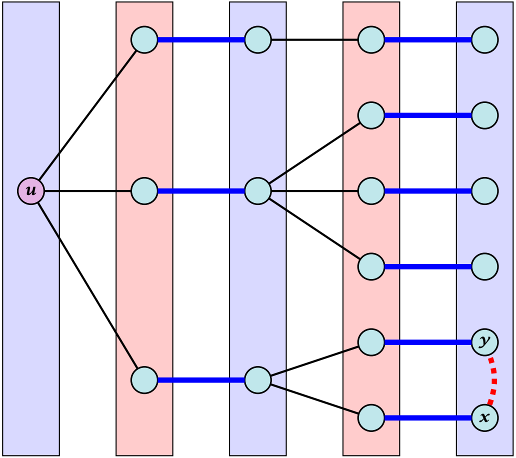
Analysis

Lemma 102

One phase of the Hopcroft-Karp algorithm can be implemented in time $\mathcal{O}(m)$.

How to find an augmenting path?

Construct an alternating tree.



even nodes

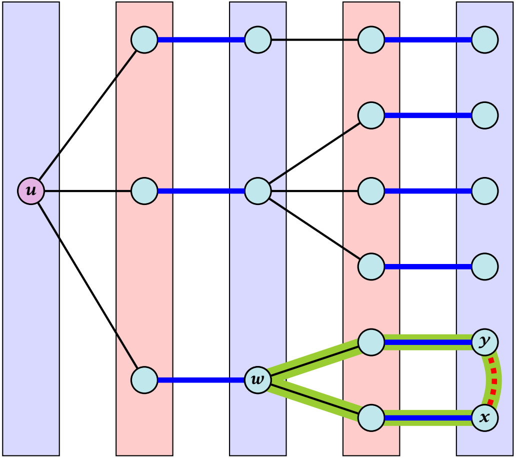
odd nodes

Case 4:
y is already contained in T as an even vertex

can't ignore y

How to find an augmenting path?

Construct an alternating tree.



even nodes

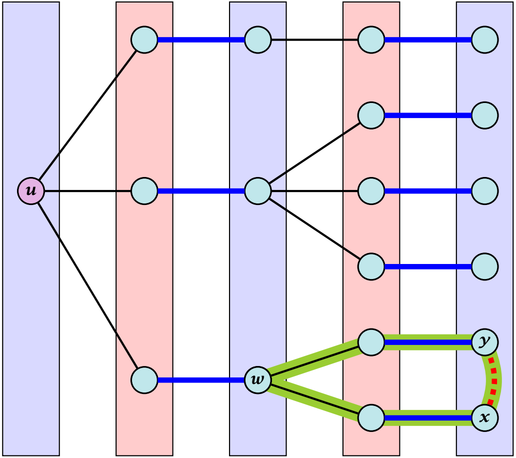
odd nodes

Case 4:
y is already contained in T as an even vertex

can't ignore y

How to find an augmenting path?

Construct an alternating tree.



even nodes

odd nodes

Case 4:
y is already contained in T as an even vertex

can't ignore y

The cycle $w \leftrightarrow y - x \leftrightarrow w$ is called a blossom.
w is called the base of the blossom (even node!!!).
The path u-w path is called the stem of the blossom.

Flowers and Blossoms

Definition 103

A flower in a graph $G = (V, E)$ w.r.t. a matching M and a (free) root node r, is a subgraph with two components:

- A stem is an even length alternating path that starts at the root node r and terminates at some node w. We permit the possibility that $r = w$ (empty stem).
- A blossom is an odd length alternating cycle that starts and terminates at the terminal node w of a stem and has no other node in common with the stem. w is called the base of the blossom.

Flowers and Blossoms

Definition 103

A flower in a graph $G = (V, E)$ w.r.t. a matching M and a (free) root node r, is a subgraph with two components:

- A stem is an even length alternating path that starts at the root node r and terminates at some node w. We permit the possibility that $r = w$ (empty stem).
- A blossom is an odd length alternating cycle that starts and terminates at the terminal node w of a stem and has no other node in common with the stem. w is called the base of the blossom.

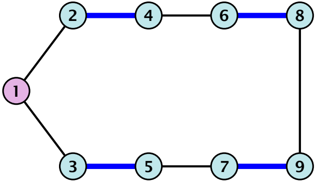
Flowers and Blossoms

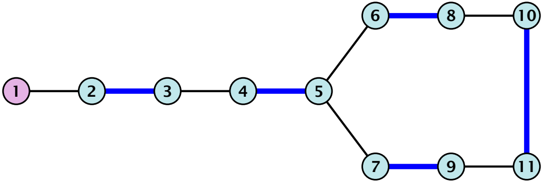
Definition 103

A flower in a graph $G = (V, E)$ w.r.t. a matching M and a (free) root node r, is a subgraph with two components:

- A stem is an even length alternating path that starts at the root node r and terminates at some node w. We permit the possibility that $r = w$ (empty stem).
- A blossom is an odd length alternating cycle that starts and terminates at the terminal node w of a stem and has no other node in common with the stem. w is called the base of the blossom.

Flowers and Blossoms





Flowers and Blossoms

Properties:

1. A stem spans $2\ell + 1$ nodes and contains ℓ matched edges for some integer $\ell \geq 0$.
2. A blossom spans $2k + 1$ nodes and contains k matched edges for some integer $k \geq 1$. The matched edges match all nodes of the blossom except the base.
3. The base of a blossom is an even node (if the stem is part of an alternating tree starting at r).

Flowers and Blossoms

Properties:

1. A stem spans $2\ell + 1$ nodes and contains ℓ matched edges for some integer $\ell \geq 0$.
2. A blossom spans $2k + 1$ nodes and contains k matched edges for some integer $k \geq 1$. The matched edges match all nodes of the blossom except the base.
3. The base of a blossom is an even node (if the stem is part of an alternating tree starting at r).

Flowers and Blossoms

Properties:

1. A stem spans $2\ell + 1$ nodes and contains ℓ matched edges for some integer $\ell \geq 0$.
2. A blossom spans $2k + 1$ nodes and contains k matched edges for some integer $k \geq 1$. The matched edges match all nodes of the blossom except the base.
3. The base of a blossom is an even node (if the stem is part of an alternating tree starting at r).

Flowers and Blossoms

Properties:

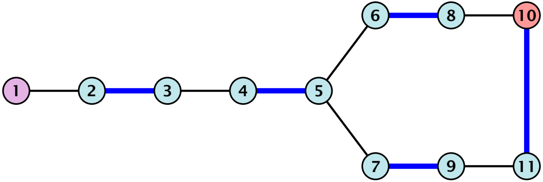
4. Every node x in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.
5. The even alternating path to x terminates with a matched edge and the odd path with an unmatched edge.

Flowers and Blossoms

Properties:

4. Every node x in the blossom (except its base) is reachable from the root (or from the base of the blossom) through two distinct alternating paths; one with even and one with odd length.
5. The even alternating path to x terminates with a matched edge and the odd path with an unmatched edge.

Flowers and Blossoms



When during the alternating tree construction we discover a blossom B we replace the graph G by $G' = G/B$, which is obtained from G by contracting the blossom B.

- Delete all vertices in B (and its incident edges) from G.
- Add a new (pseudo-)vertex b. The new vertex b is connected to all vertices in $V \setminus B$ that had at least one edge to a vertex from B.

When during the alternating tree construction we discover a blossom B we replace the graph G by $G' = G/B$, which is obtained from G by contracting the blossom B.

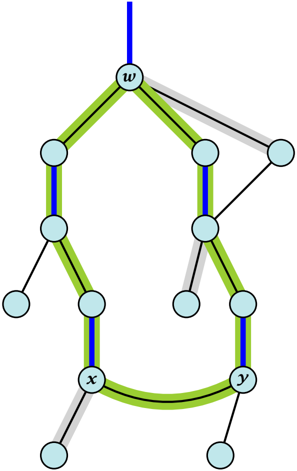
- Delete all vertices in B (and its incident edges) from G.
- Add a new (pseudo-)vertex b. The new vertex b is connected to all vertices in $V \setminus B$ that had at least one edge to a vertex from B.

When during the alternating tree construction we discover a blossom B we replace the graph G by $G' = G/B$, which is obtained from G by contracting the blossom B.

- Delete all vertices in B (and its incident edges) from G.
- Add a new (pseudo-)vertex b. The new vertex b is connected to all vertices in $V \setminus B$ that had at least one edge to a vertex from B.

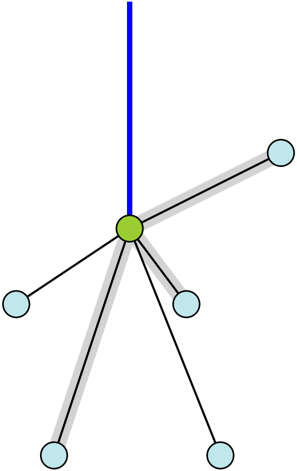
Shrinking Blossoms

- Edges of T that connect a node u not in B to a node in B become tree edges in T' connecting u to b.
- Matching edges (there is at most one) that connect a node u not in B to a node in B become matching edges in M'.
- Nodes that are connected in G to at least one node in B become connected to b in G'.



Shrinking Blossoms

- Edges of T that connect a node u not in B to a node in B become tree edges in T' connecting u to b.
- Matching edges (there is at most one) that connect a node u not in B to a node in B become matching edges in M'.
- Nodes that are connected in G to at least one node in B become connected to b in G'.



Algorithm 55 search(r, $found$)

```
set $\bar{A}(i) \leftarrow A(i)$ for all nodes $i$
$found \leftarrow \text{false}$
unlabel all nodes;
give an even label to $r$ and initialize $list \leftarrow \{r\}$
while $list \neq \emptyset$ do
    delete a node $i$ from $list$
    examine($i$, $found$)
    if $found = \text{true}$ then
        return
```

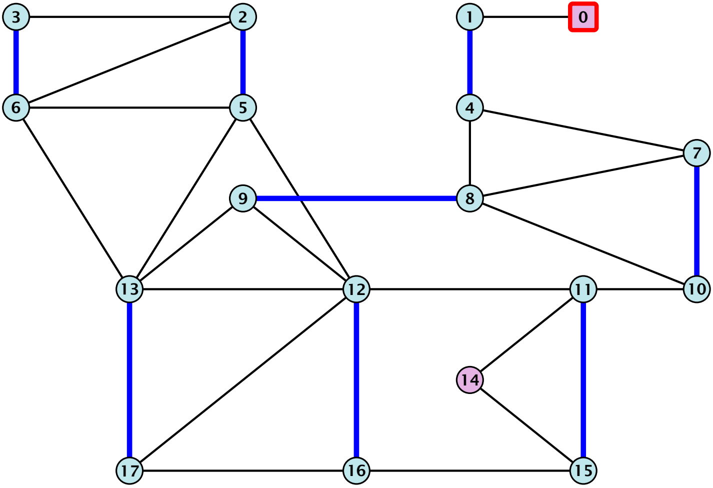
Algorithm 56 examine(i, $found$)

```
for all $j \in \bar{A}(i)$ do
    if $j$ is even then contract$(i, j)$ and return
    if $j$ is unmatched then
        $q \leftarrow j$;
        $\text{pred}(q) \leftarrow i$;
        $found \leftarrow \text{true}$;
        return
    if $j$ is matched and unlabeled then
        $\text{pred}(j) \leftarrow i$;
        $\text{pred}(\text{mate}(j)) \leftarrow j$;
```

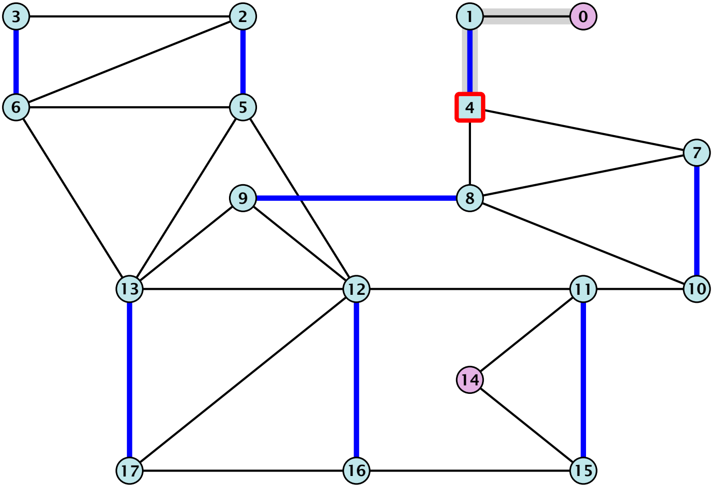
Algorithm 57 contract(i, j)

```
trace pred-indices of $i$ and $j$ to identify a blossom $B$
create new node $b$ and set $\bar{A}(b) \leftarrow \cup_{x \in B} \bar{A}(k)$
label $b$ even and add to *list*
update $\bar{A}(j) \leftarrow \bar{A}(j) \cup \{b\}$ for each $j \in \bar{A}(b)$
form a circular doubly linked list of nodes in $B$
delete nodes in $B$ from the graph
```

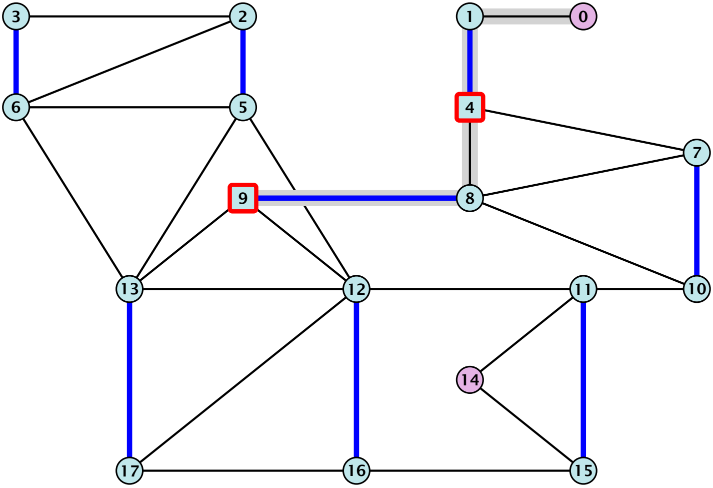
Example: Blossom Algorithm



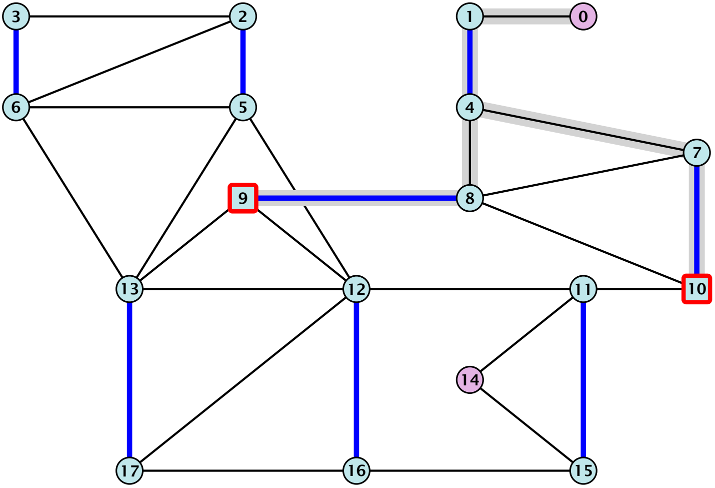
Example: Blossom Algorithm



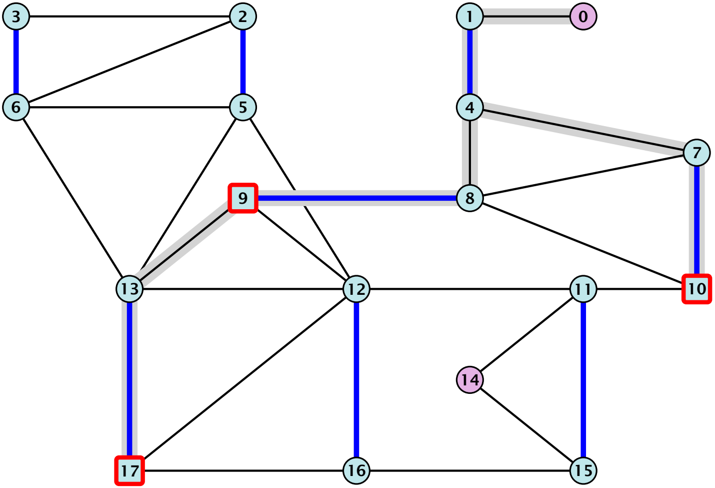
Example: Blossom Algorithm



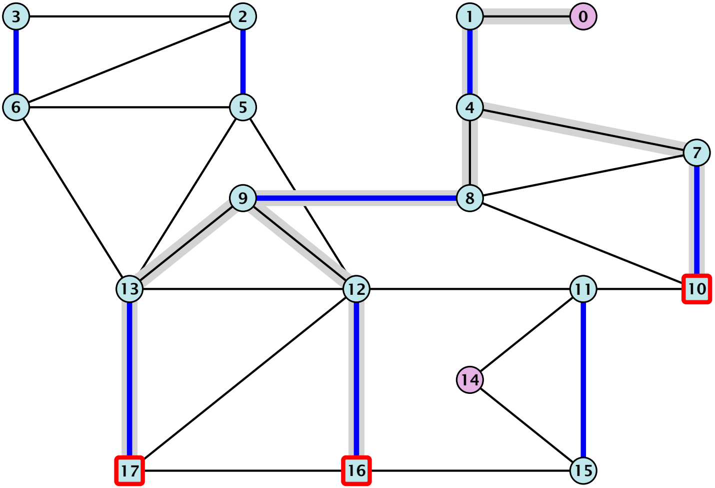
Example: Blossom Algorithm



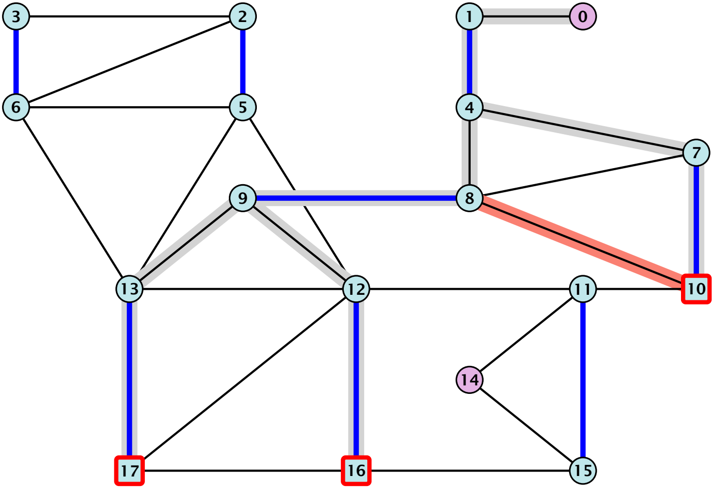
Example: Blossom Algorithm



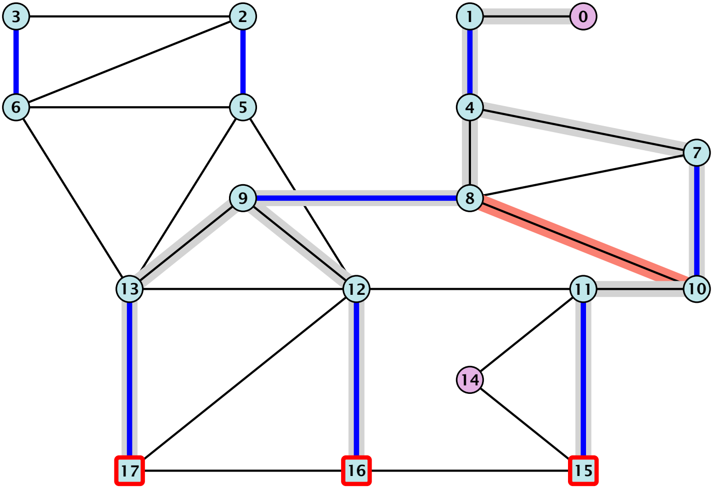
Example: Blossom Algorithm



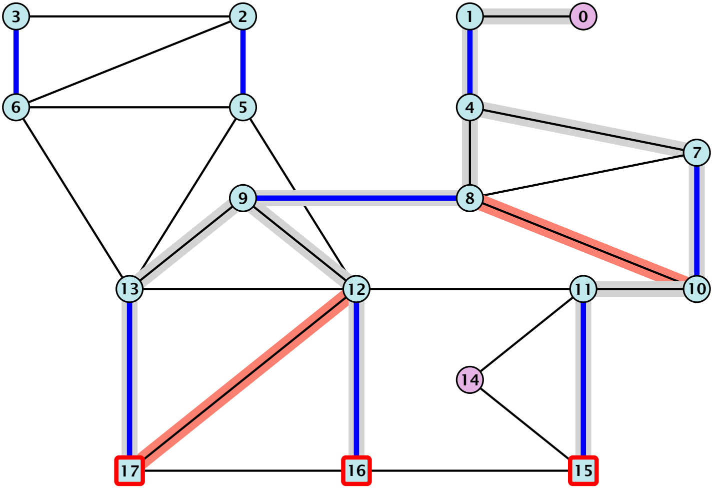
Example: Blossom Algorithm



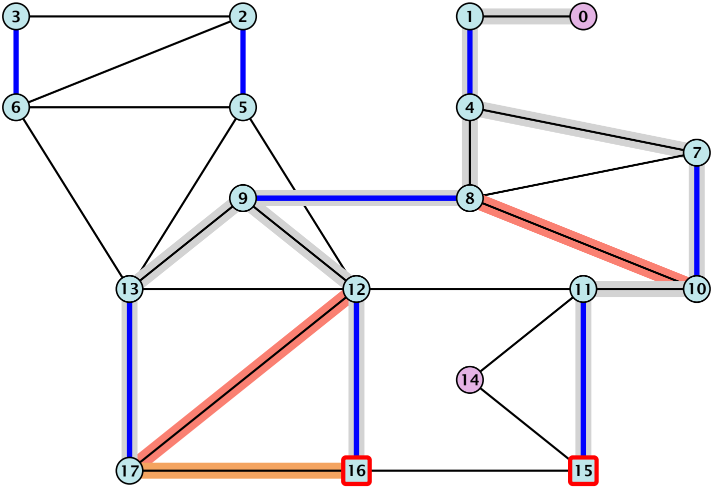
Example: Blossom Algorithm



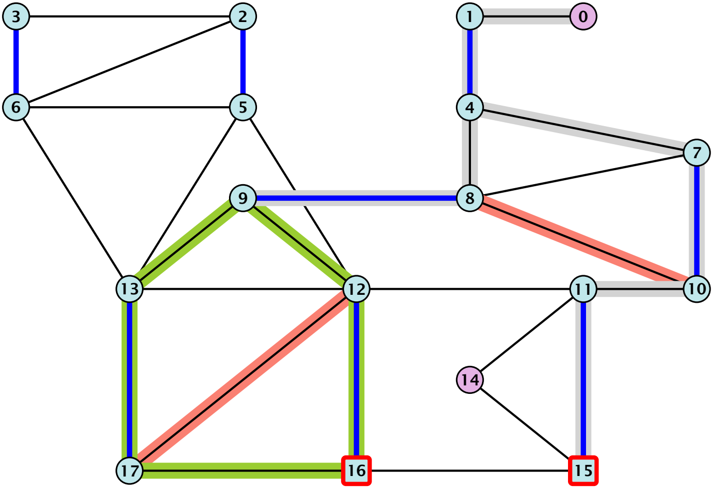
Example: Blossom Algorithm



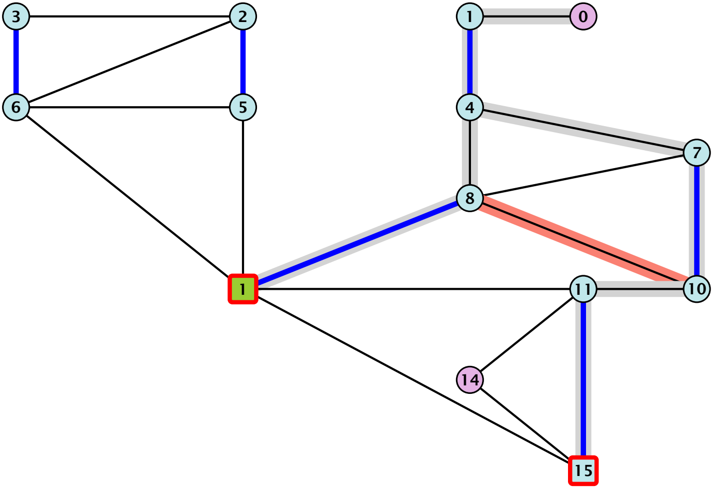
Example: Blossom Algorithm



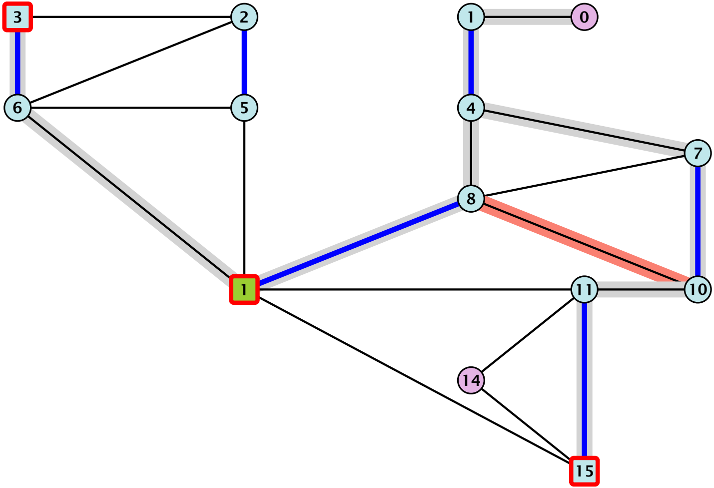
Example: Blossom Algorithm



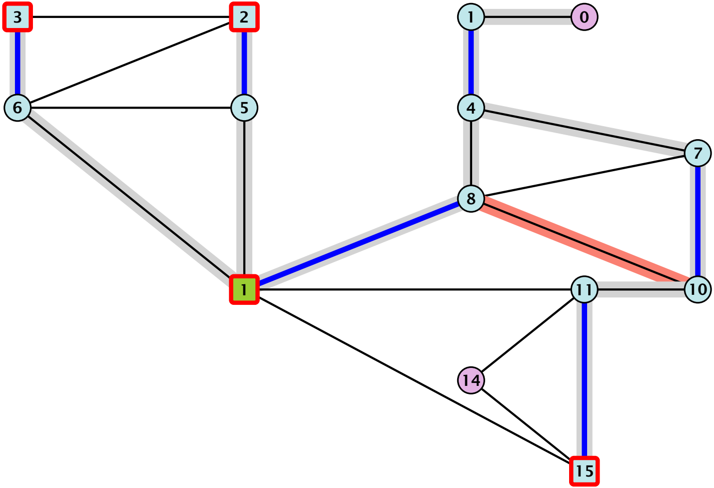
Example: Blossom Algorithm



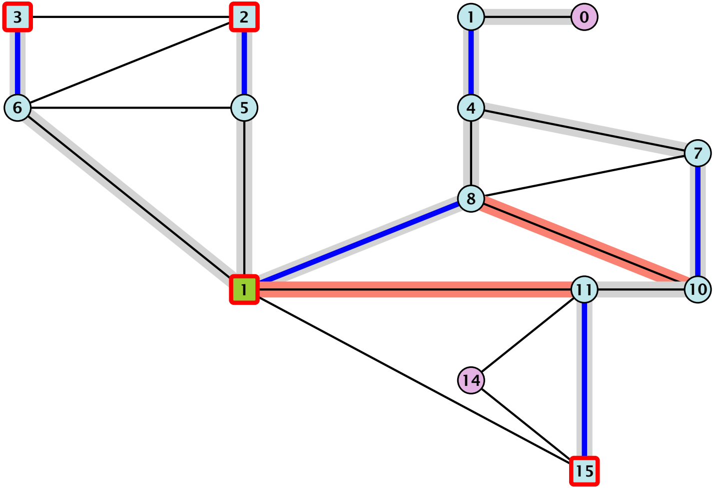
Example: Blossom Algorithm



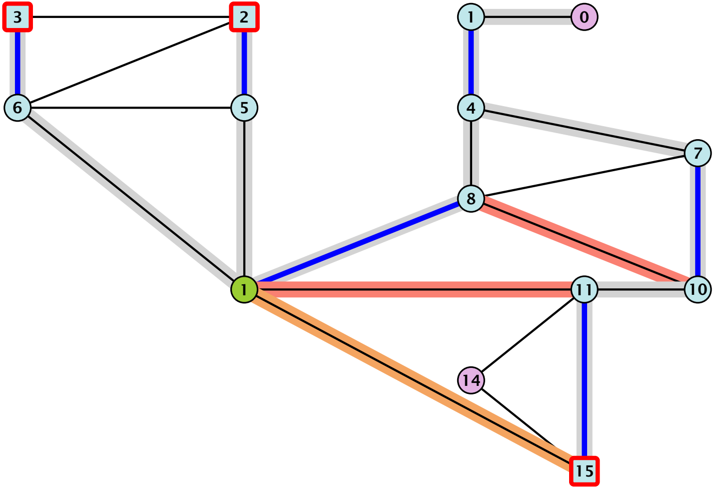
Example: Blossom Algorithm



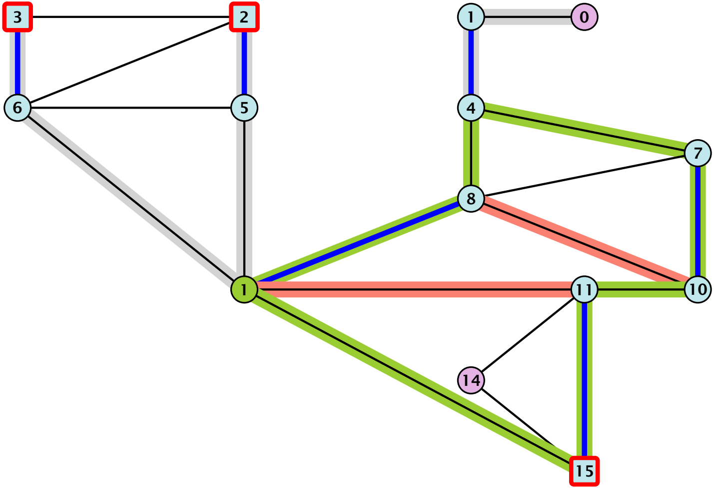
Example: Blossom Algorithm



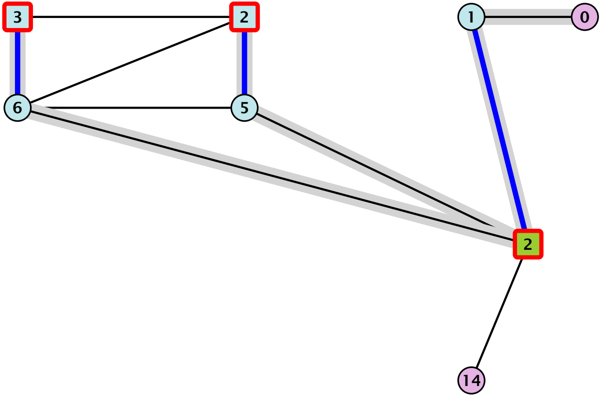
Example: Blossom Algorithm



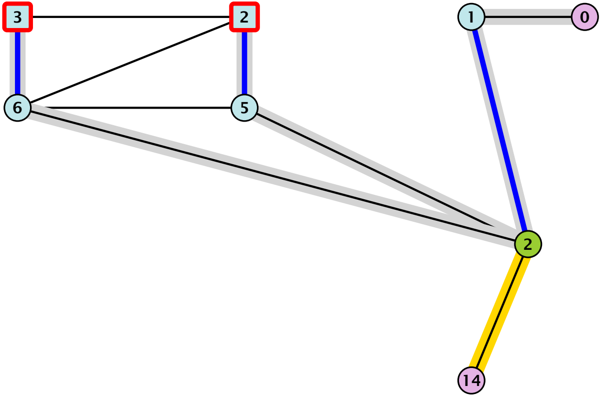
Example: Blossom Algorithm



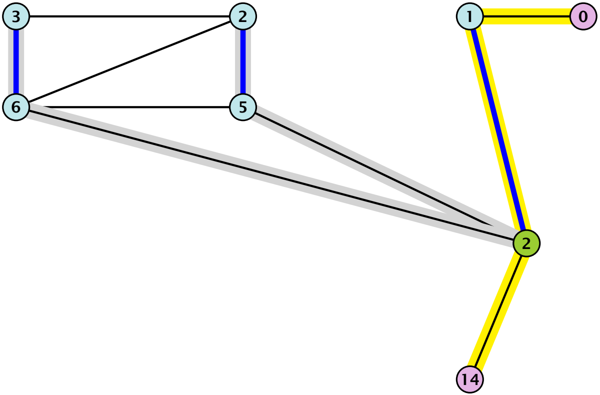
Example: Blossom Algorithm



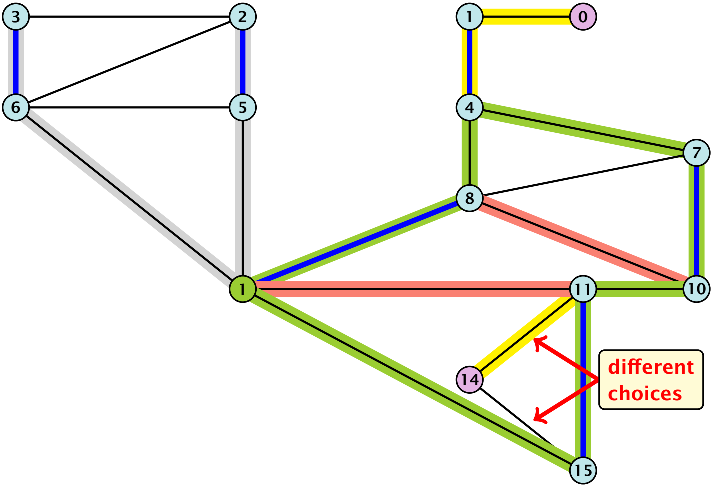
Example: Blossom Algorithm



Example: Blossom Algorithm

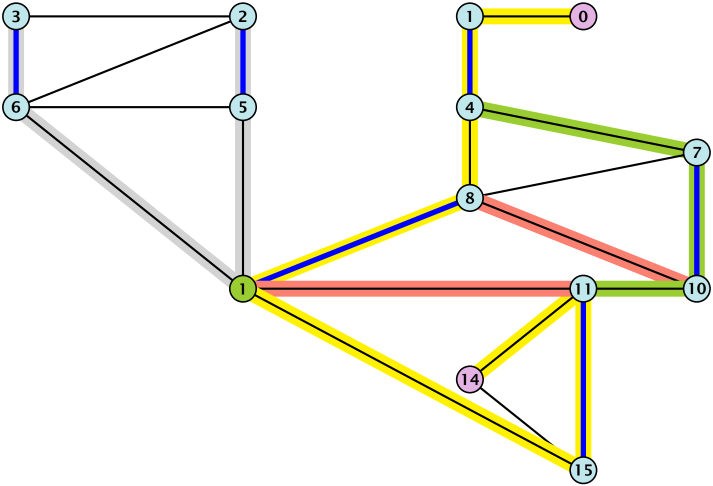


Example: Blossom Algorithm

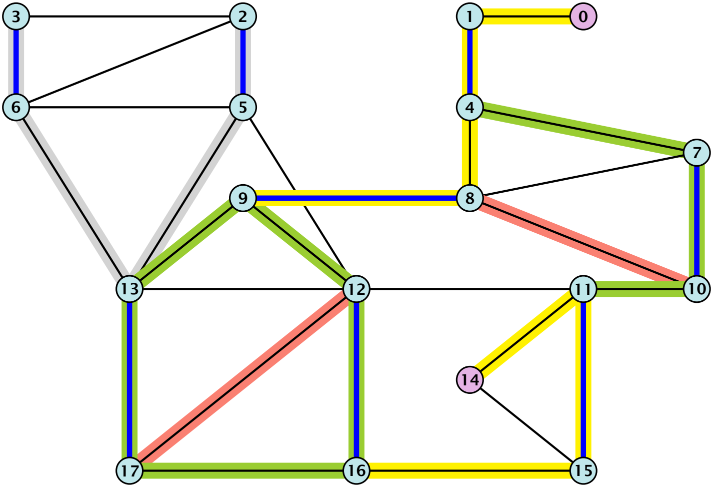


different choices

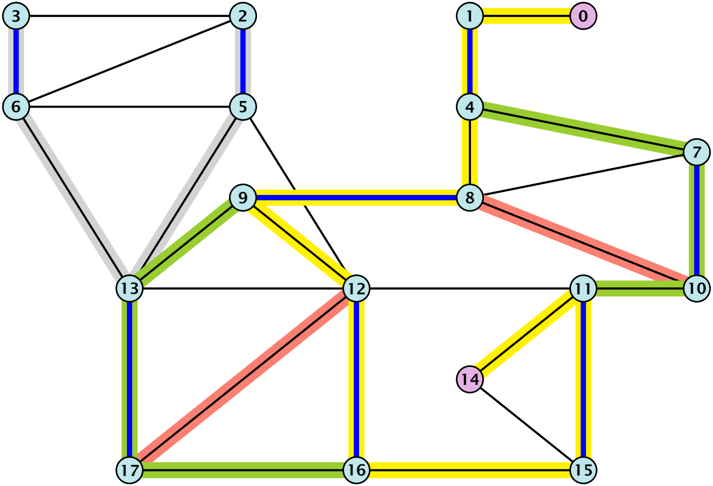
Example: Blossom Algorithm



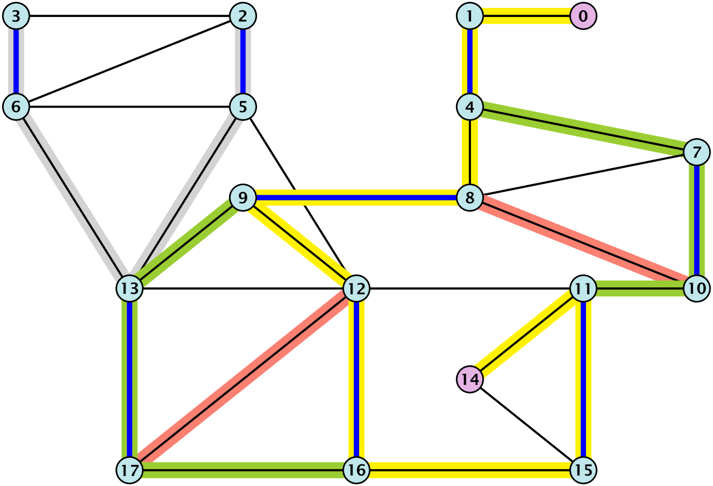
Example: Blossom Algorithm



Example: Blossom Algorithm



Example: Blossom Algorithm



Assume that we have contracted a blossom B w.r.t. a matching M whose base is w. We created graph $G' = G/B$ with pseudonode b. Let M' be the matching in the contracted graph.

Lemma 104

If G' contains an augmenting path p' starting at r (or the pseudo-node containing r) w.r.t. to the matching M' then G contains an augmenting path starting at r w.r.t. matching M.

Assume that we have contracted a blossom B w.r.t. a matching M whose base is w. We created graph $G' = G/B$ with pseudonode b. Let M' be the matching in the contracted graph.

Lemma 104

If G' contains an augmenting path p' starting at r (or the pseudo-node containing r) w.r.t. to the matching M' then G contains an augmenting path starting at r w.r.t. matching M.

Proof.

If p' does not contain b it is also an augmenting path in G.

Proof.

If p' does not contain b it is also an augmenting path in G.

Case 1: non-empty stem

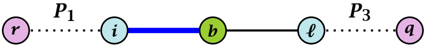
- Next suppose that the stem is non-empty.

Proof.

If p' does not contain b it is also an augmenting path in G.

Case 1: non-empty stem

- Next suppose that the stem is non-empty.

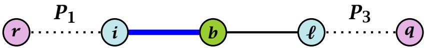


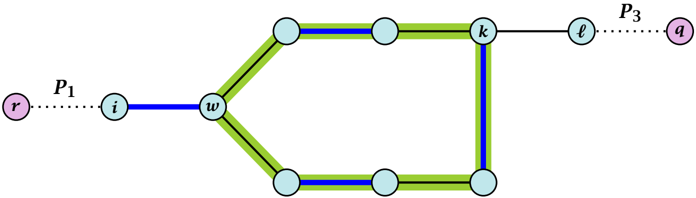
Proof.

If p' does not contain b it is also an augmenting path in G.

Case 1: non-empty stem

- Next suppose that the stem is non-empty.





- After the expansion ℓ must be incident to some node in the blossom. Let this node be k.
- If $k \neq w$ there is an alternating path P_2 from w to k that ends in a matching edge.
- $P_1 \circ (i, w) \circ P_2 \circ (k, \ell) \circ P_3$ is an alternating path.
- If $k = w$ then $P_1 \circ (i, w) \circ (w, \ell) \circ P_3$ is an alternating path.

Proof.

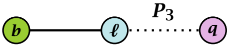
Case 2: empty stem

- If the stem is empty then after expanding the blossom, $w = r$.

Proof.

Case 2: empty stem

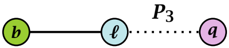
- If the stem is empty then after expanding the blossom, $w = r$.

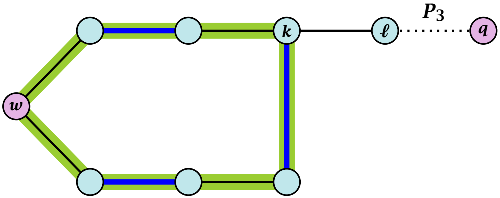


Proof.

Case 2: empty stem

- If the stem is empty then after expanding the blossom, $w = r$.

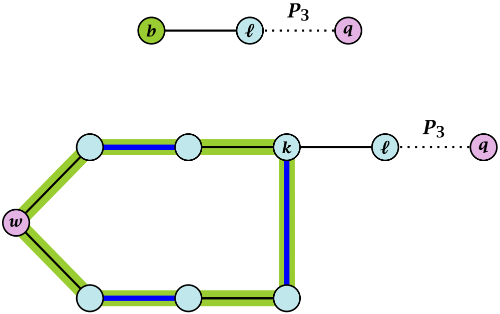




Proof.

Case 2: empty stem

- If the stem is empty then after expanding the blossom, $w = r$.



- The path $r \circ P_2 \circ (k, \ell) \circ P_3$ is an alternating path.

Lemma 105

If G contains an augmenting path P from r to q w.r.t. matching M then G' contains an augmenting path from r (or the pseudo-node containing r) to q w.r.t. M'.

Proof.

- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i, j) \circ P_2$, for some node j and (i, j) is unmatched.

$(b, j) \circ P_2$ is an augmenting path in the contracted network.

Proof.

- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i,j) \circ P_2$, for some node j and (i,j) is unmatched.

$(b,j) \circ P_2$ is an augmenting path in the contracted network.

Proof.

- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i, j) \circ P_2$, for some node j and (i, j) is unmatched.

$(b, j) \circ P_2$ is an augmenting path in the contracted network.

Proof.

- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i,j) \circ P_2$, for some node j and (i,j) is unmatched.

$(b,j) \circ P_2$ is an augmenting path in the contracted network.

Proof.

- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i, j) \circ P_2$, for some node j and (i, j) is unmatched.

$(b, j) \circ P_2$ is an augmenting path in the contracted network.

Proof.

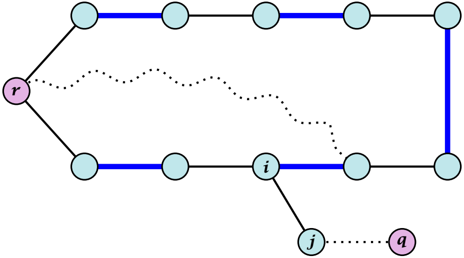
- If P does not contain a node from B there is nothing to prove.
- We can assume that r and q are the only free nodes in G.

Case 1: empty stem

Let i be the last node on the path P that is part of the blossom.

P is of the form $P_1 \circ (i, j) \circ P_2$, for some node j and (i, j) is unmatched.

$(b, j) \circ P_2$ is an augmenting path in the contracted network.


r
i
j
q


b
j
q

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

Case 2: non-empty stem

Let P_3 be alternating path from r to w. Define $M_+ = M \oplus P_3$.

In M_+, r is matched and w is unmatched.

G must contain an augmenting path w.r.t. matching M_+, since M and M_+ have same cardinality.

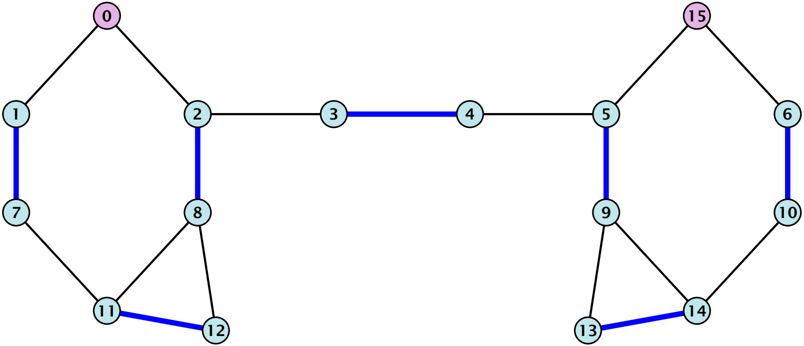
This path must go between w and q as these are the only unmatched vertices w.r.t. M_+.

For M'_+ the blossom has an empty stem. Case 1 applies.

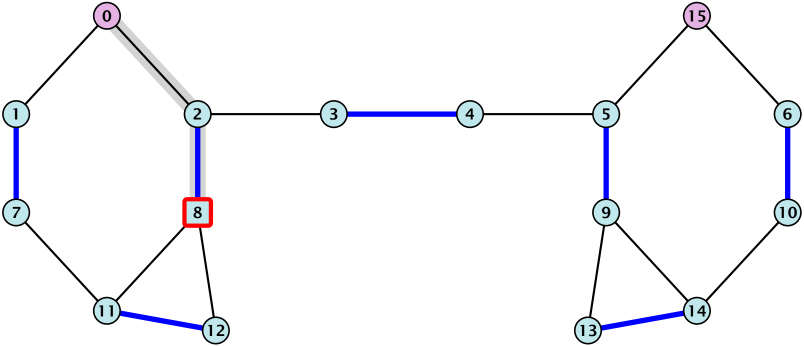
G' has an augmenting path w.r.t. M'_+. It must also have an augmenting path w.r.t. M', as both matchings have the same cardinality.

This path must go between r and q.

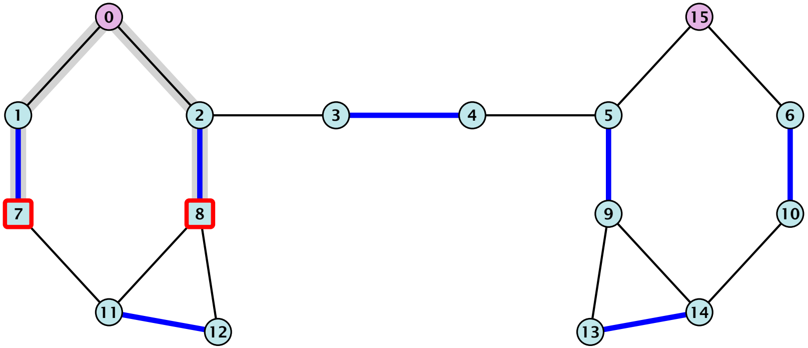
Example: Blossom Algorithm



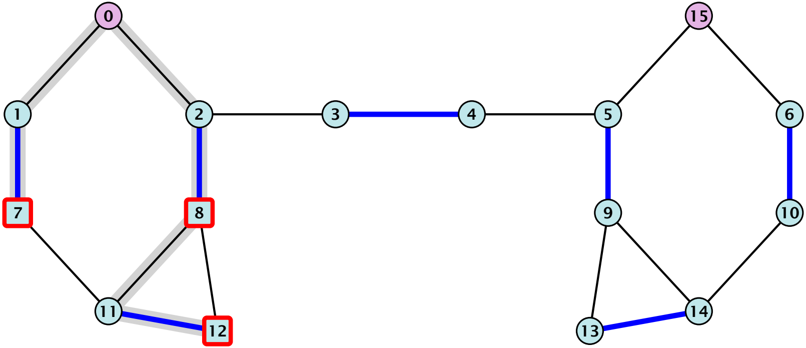
Example: Blossom Algorithm



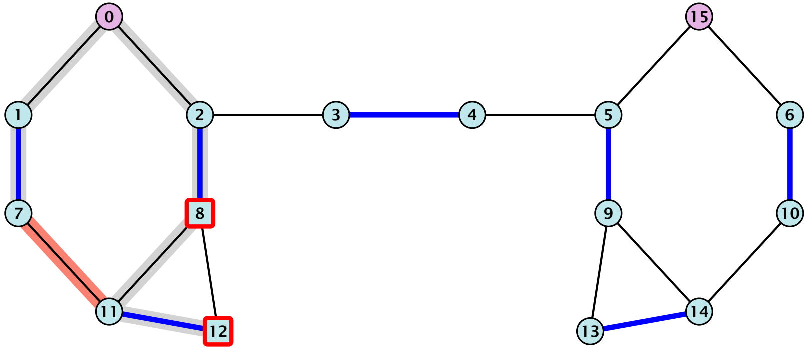
Example: Blossom Algorithm



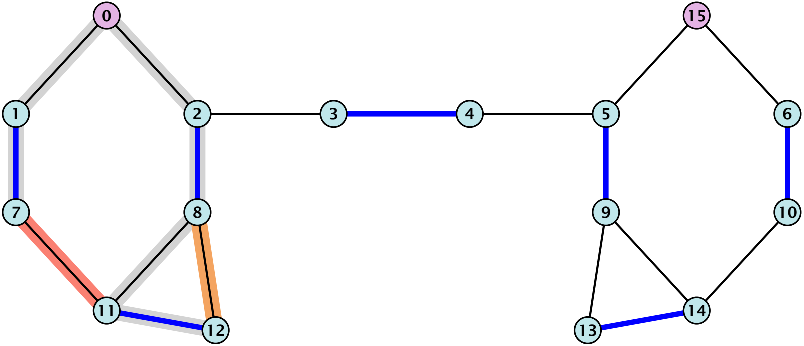
Example: Blossom Algorithm



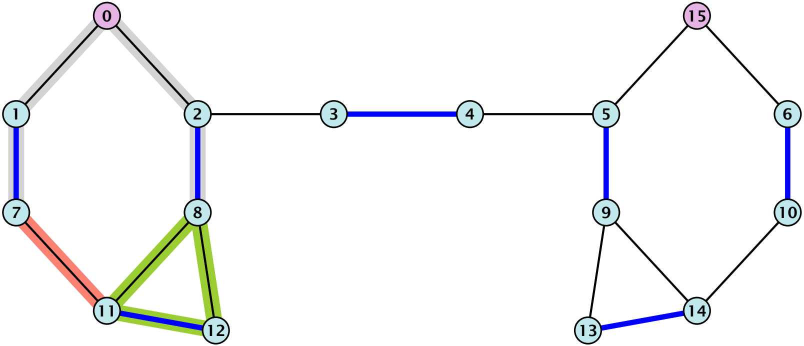
Example: Blossom Algorithm



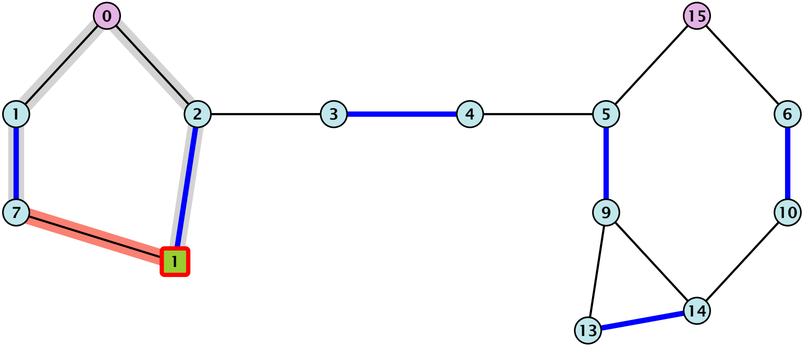
Example: Blossom Algorithm



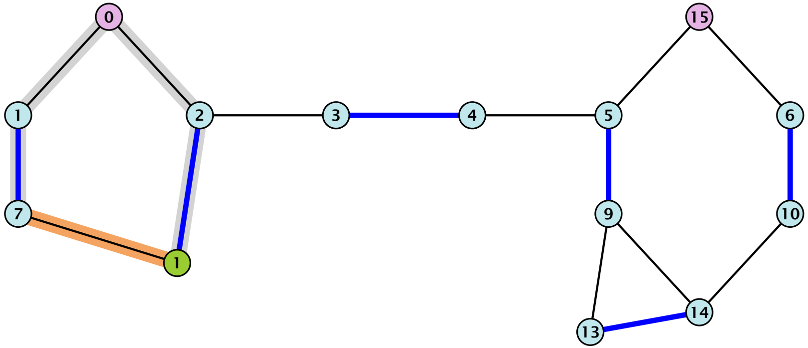
Example: Blossom Algorithm



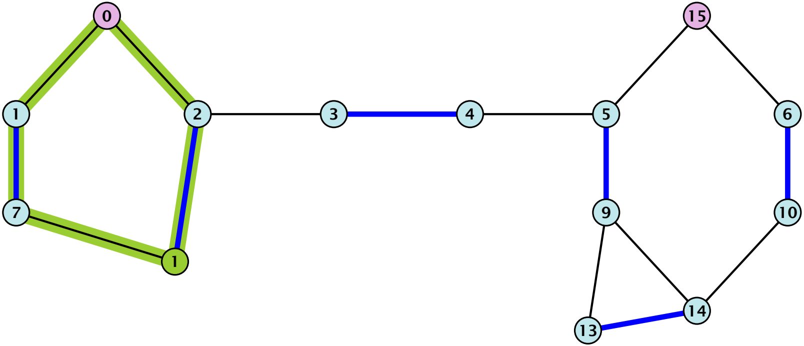
Example: Blossom Algorithm



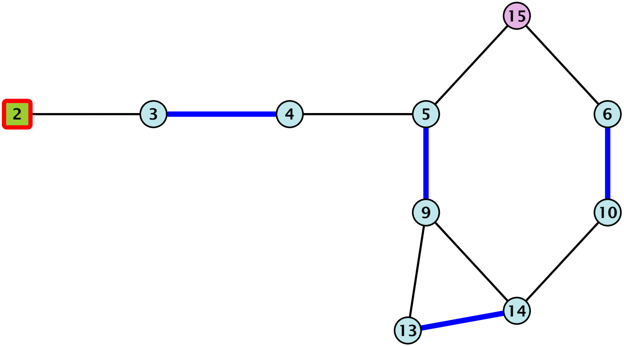
Example: Blossom Algorithm



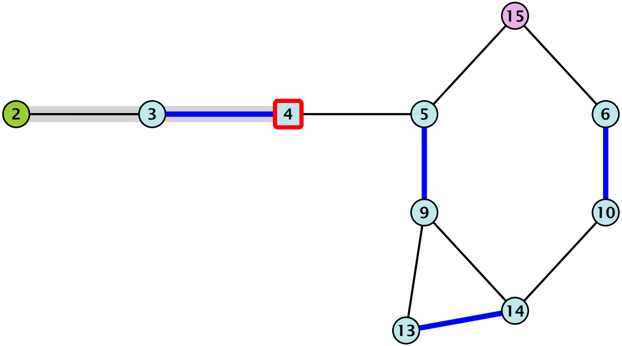
Example: Blossom Algorithm



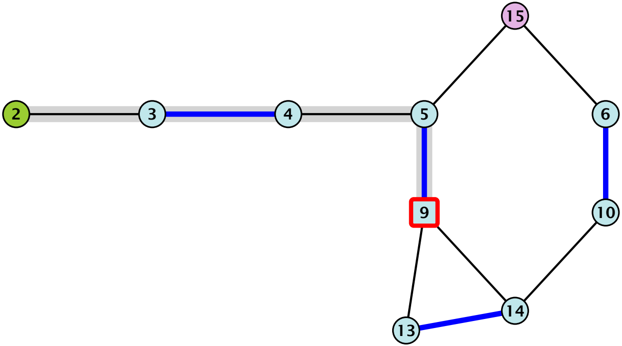
Example: Blossom Algorithm



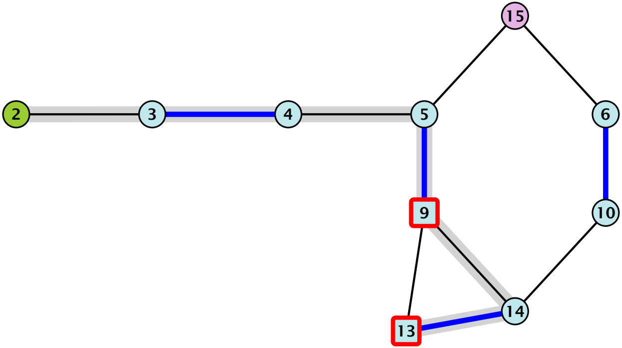
Example: Blossom Algorithm



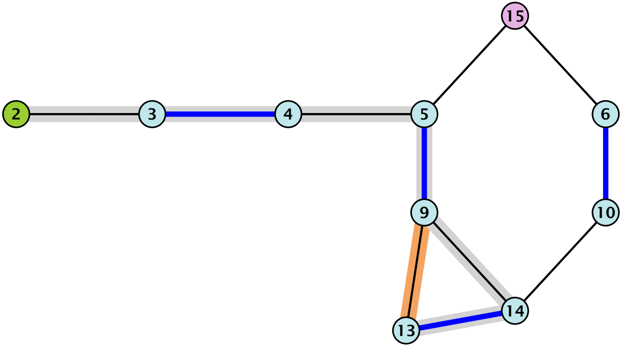
Example: Blossom Algorithm



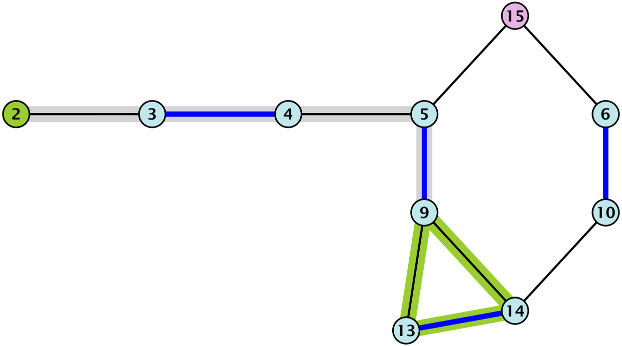
Example: Blossom Algorithm



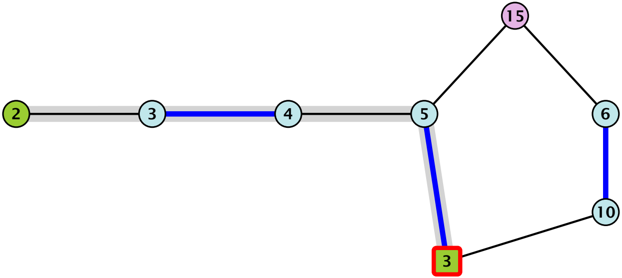
Example: Blossom Algorithm



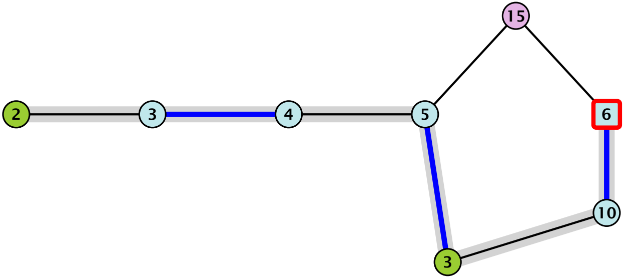
Example: Blossom Algorithm



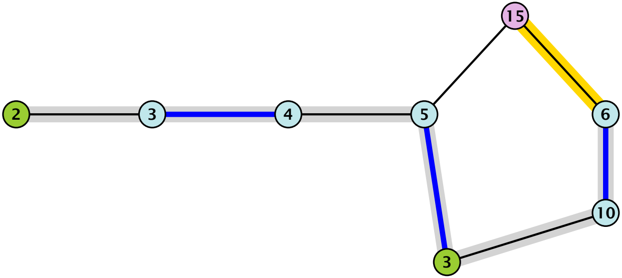
Example: Blossom Algorithm



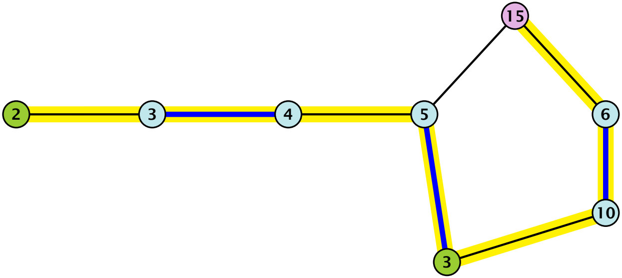
Example: Blossom Algorithm



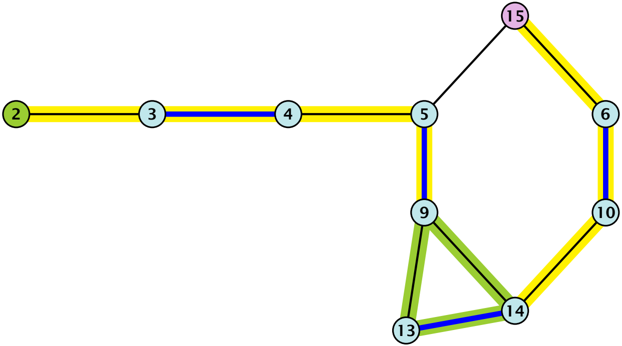
Example: Blossom Algorithm



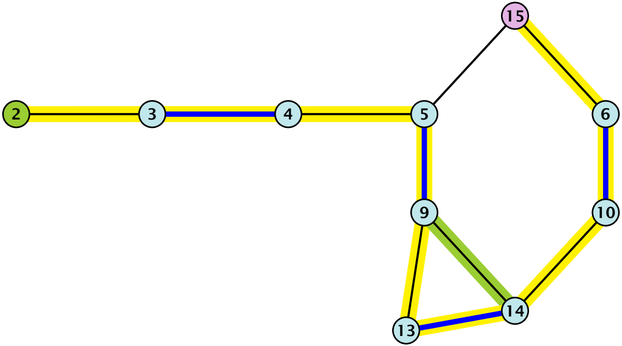
Example: Blossom Algorithm



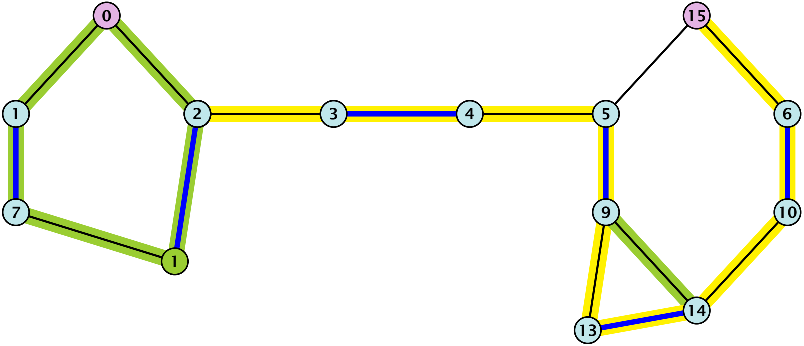
Example: Blossom Algorithm



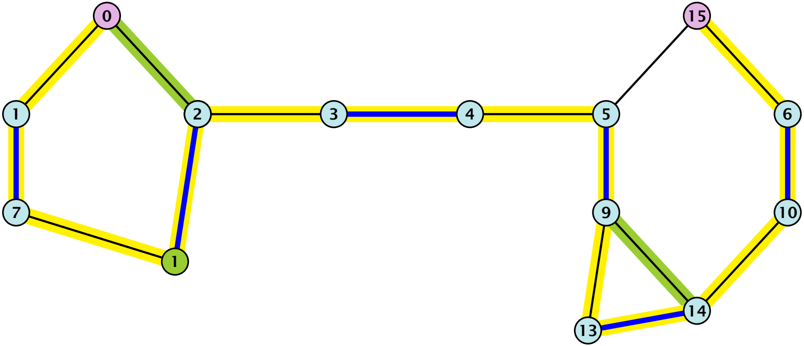
Example: Blossom Algorithm



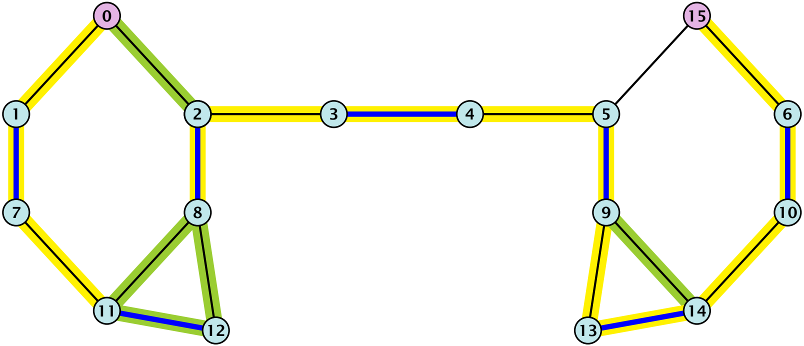
Example: Blossom Algorithm



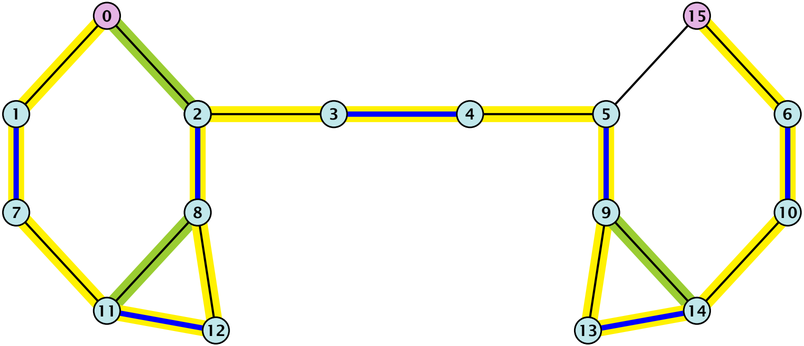
Example: Blossom Algorithm



Example: Blossom Algorithm



Example: Blossom Algorithm



Example: Blossom Algorithm