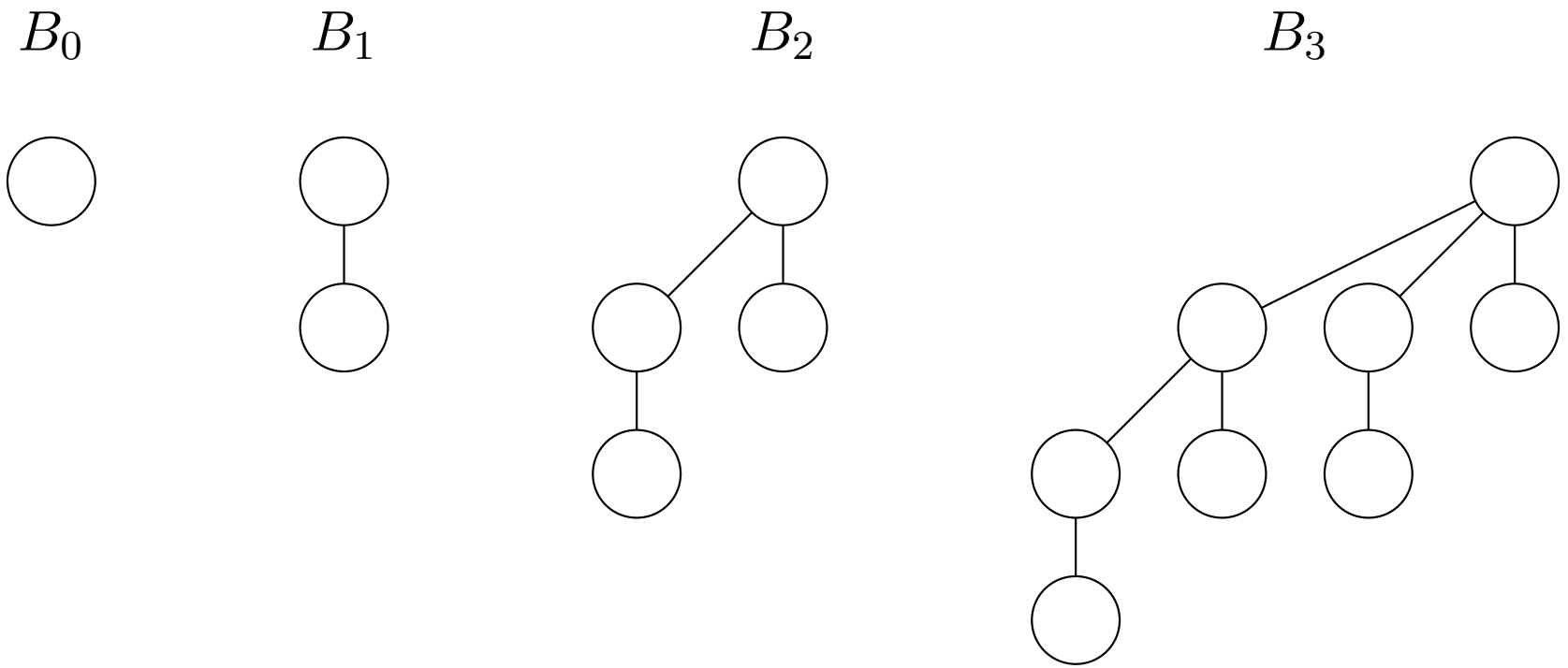
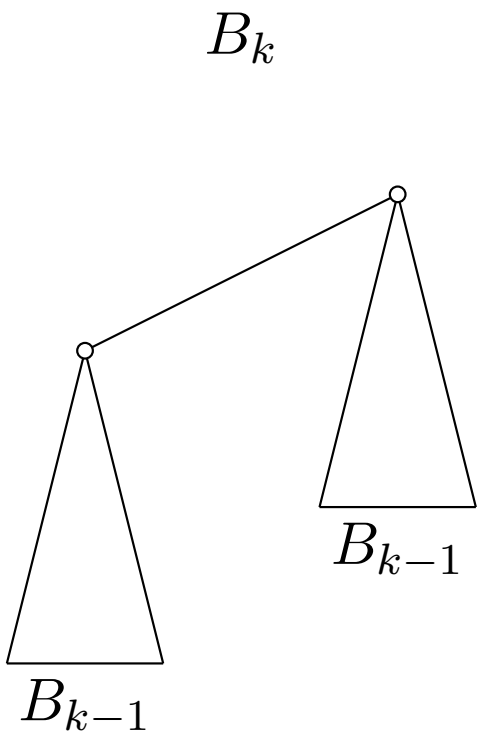
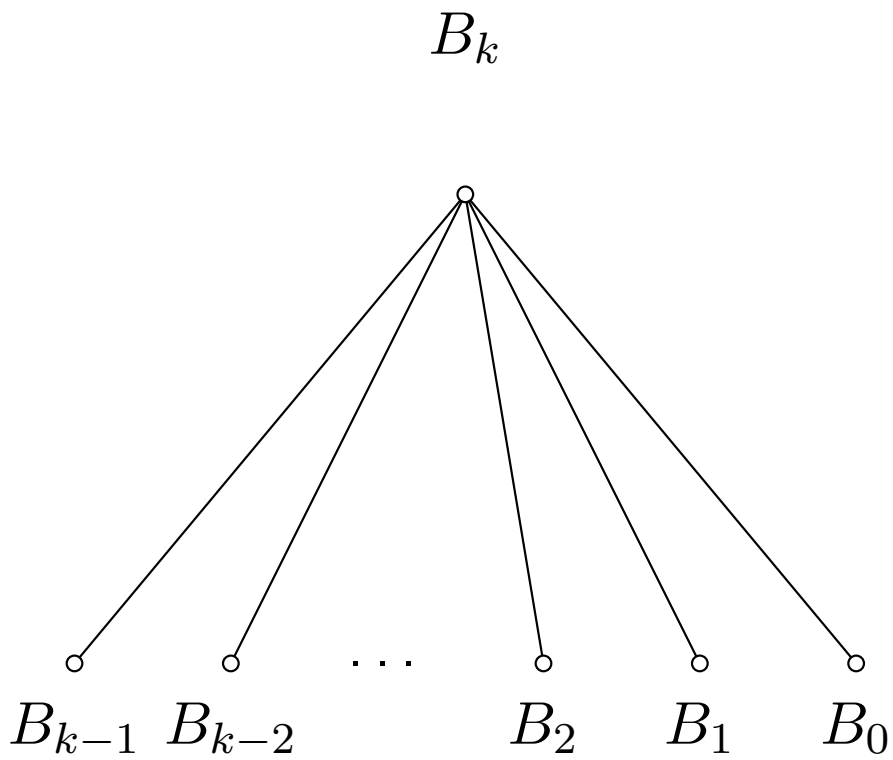
Definition 176

- Der Binomialbaum B_0 besteht aus genau einem Knoten.
- Den Binomialbaum B_k, $k \geq 1$, erhält man, indem man die Wurzel eines B_{k-1} zu einem zusätzlichen Kind der Wurzel eines zweiten B_{k-1} macht.

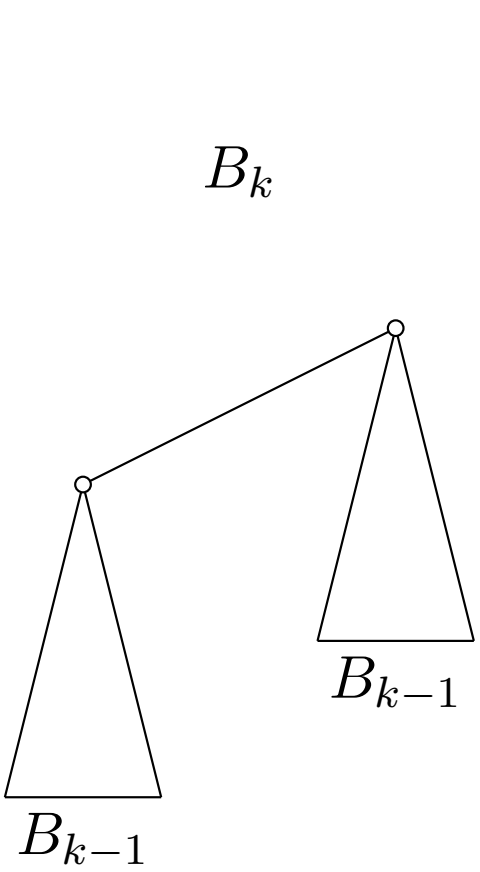


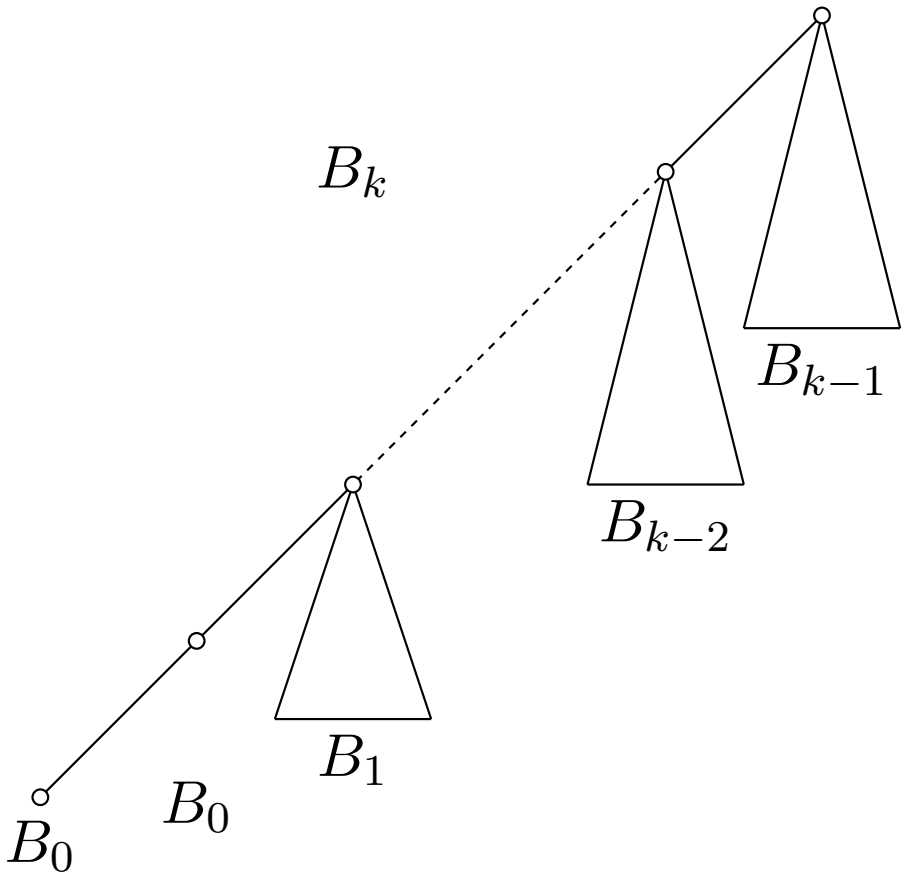
Rekursiver Aufbau des Binomialbaums B_k (rekursive Verfolgung des rechten Zweigs)





Rekursiver Aufbau des Binomialbaums B_k (rekursive Verfolgung des linken Zweigs)





Satz 177

Für den Binomialbaum B_k, $k \geq 0$, gilt:

1. *er hat Höhe k und enthält 2^k Knoten;*
2. *er enthält genau $\binom{k}{i}$ Knoten der Tiefe i, für alle $i \in \{0, \ldots, k\}$;*
3. *seine Wurzel hat k Kinder, alle anderen Knoten haben $< k$ Kinder.*

Beweis:

Der Beweis ergibt sich sofort aus dem rekursiven Aufbau des B_k und der Rekursionsformel

$$\binom{k}{i} = \binom{k-1}{i} + \binom{k-1}{i-1}$$

für Binomialkoeffizienten.

Definition 178

Ein Binomial Heap ist eine *Menge* $\mathcal{H}$ von Binomialbäumen, wobei jedem Knoten v ein Schlüssel $key(v)$ zugeordnet ist, so dass folgende Eigenschaften gelten:

1. jeder Binomialbaum $\in \mathcal{H}$ erfüllt die Heap-Bedingung und ist ein min-Heap:

$$(\forall \text{ Knoten } v, w)[v \text{ Vater von } w \Rightarrow key(v) \leq key(w)]$$

2. $\mathcal{H}$ enthält für jedes $k \in \mathbb{N}_0$ höchstens einen B_k

Für jeden Binomial Heap $\mathcal{H}$ gilt also

1. Enthält $\mathcal{H}$ n Schlüssel, $n \in \mathbb{N}$, so besteht $\mathcal{H}$ höchstens aus $\max\{1, \lceil \mathrm{ld}\, n \rceil\}$ Binomialbäumen.
2. In jedem von $\mathcal{H}$'s Binomialbäumen ist ein kleinster Schlüssel an der Wurzel gespeichert; verlinkt man daher die Wurzeln aller Binomialbäume von $\mathcal{H}$ in einer zirkulären Liste, kann ein minimaler Schlüssel in $\mathcal{H}$ durch einfaches Durchlaufen dieser Liste gefunden werden.

Korollar 179

In einem Binomial Heap mit n Schlüsseln benötigt FindMin *Zeit* $O(\log n)$.

Wir betrachten nun die Realisierung der Union-Operation. Hierbei wird vorausgesetzt, dass die Objektmengen, zu denen die Schlüssel in den beiden zu verschmelzenden Binomial Heaps $\mathcal{H}_1$ und $\mathcal{H}_2$ gehören, disjunkt sind. Es ist jedoch durchaus möglich, dass der gleiche Schlüssel für mehrere Objekte vorkommt.

1. Fall $\mathcal{H}_1$ und $\mathcal{H}_2$ enthalten jeweils nur einen Binomialbaum B_k (mit dem gleichen Index k):

In diesem Fall fügen wir den B_k, dessen Wurzel den größeren Schlüssel hat, als neuen Unterbaum der Wurzel des anderen B_k ein, gemäß der rekursiven Struktur der Binomialbäume. Es entsteht ein B_{k+1}, für den per Konstruktion weiterhin die Heap-Bedingung erfüllt ist.

2. Fall Sonst. Sei $\mathcal{H}_1 = \{B_i^1;\ i \in I_1 \subset \mathbb{N}_0\}$ und sei $\mathcal{H}_2 = \{B_i^2;\ i \in I_2 \subset \mathbb{N}_0\}$.

Wir durchlaufen dann alle Indizes $k \in [0, \max\{I_1 \cup I_2\}]$ in aufsteigender Reihenfolge und führen folgende Schritte durch:

1. falls kein Binomialbaum mit Index k vorhanden ist: **noop**
2. falls genau ein Binomialbaum mit Index k vorhanden ist, wird dieser in den verschmolzenen Binomial Heap übernommen
3. falls genau zwei Binomialbäume mit Index k vorhanden sind, werden diese gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet. Dadurch kann auch der folgende Fall eintreten!
4. falls genau drei Binomialbäume mit Index k vorhanden sind, wird einer davon in den verschmolzenen Binomial Heap übernommen; die beiden anderen werden gemäß dem 1. Fall verschmolzen; der entstehende B_{k+1} wird im nächsten Schleifendurchlauf wieder betrachtet.

Man beachte, dass nie mehr als 3 Binomialbäume mit gleichem Index auftreten können!

Bemerkung:
Es besteht eine einfache Analogie zur Addition zweier Binärzahlen, wobei das Verschmelzen zweier gleich großer Binomialbäume dem Auftreten eines Übertrags entspricht!

Lemma 180
Die Union-*Operation zweier Binomial Heaps mit zusammen* n *Schlüsseln kann in Zeit* $O(\log n)$ *durchgeführt werden.*

Die Operation Insert$(\mathcal{H}, k)$:

1. erzeuge einen neuen Binomial Heap $\mathcal{H}' = \{B_0\}$ mit k als einzigem Schlüssel
2. $\mathcal{H} :=$ Union$(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der Insert-Operation offensichtlich $O(\log n)$.

Die Operation ExtractMin($\mathcal{H}$):

1. durchlaufe die Liste der Wurzeln der Binomialbäume in $\mathcal{H}$ und finde den/einen Baum mit minimaler Wurzel
2. gib den Schlüssel dieser Wurzel zurück
3. entferne diesen Binomialbaum aus $\mathcal{H}$
4. erzeuge einen neuen Binomial Heap $\mathcal{H}'$ aus den Unterbäumen der Wurzel dieses Baums
5. $\mathcal{H} :=$ Union$(\mathcal{H}, \mathcal{H}')$

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der ExtractMin-Operation offensichtlich $O(\log n)$.

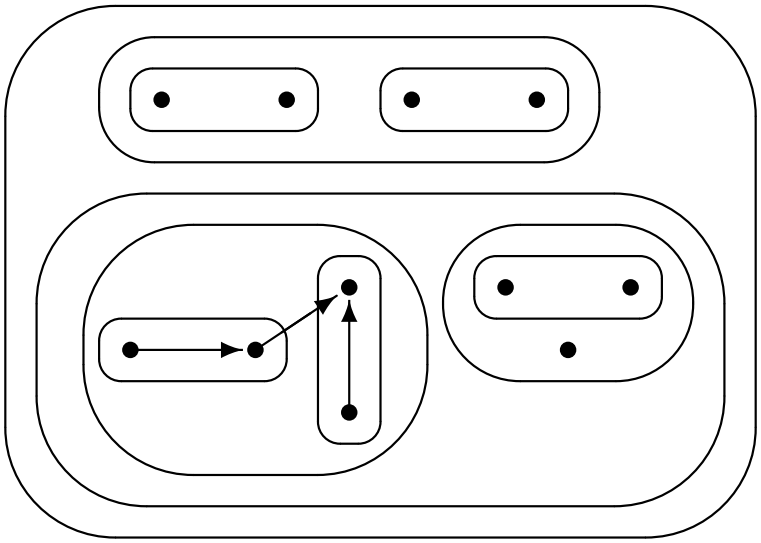
Die Operation DecreaseKey($\mathcal{H}, v, k$) (diese Operation ersetzt, falls $k < key(v)$, $key(v)$ durch k):

1. sei B_i der Binomialbaum in $\mathcal{H}$, der den Knoten v enthält
2. falls $k < key(v)$, ersetze $key(v)$ durch k
3. stelle, falls nötig, die Heap-Bedingung auf dem Pfad von v zur Wurzel von B_i wieder her, indem, solange nötig, der Schlüssel eines Knotens mit dem seines Vaters ausgetauscht wird

Falls $\mathcal{H}$ n Schlüssel enthält, beträgt die Laufzeit der DecreaseKey-Operation offensichtlich $O(\log n)$.

5. Union/Find-Datenstrukturen

5.1 Motivation

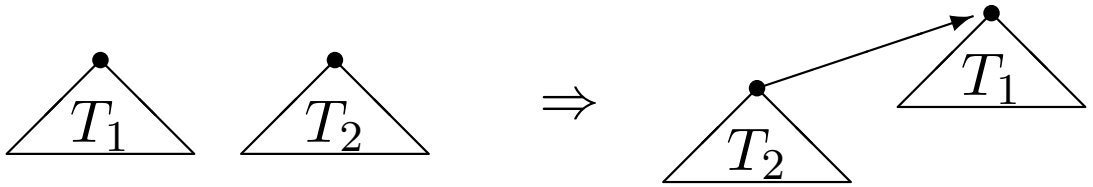


- *Union*(T_1, T_2): Vereinige T_1 und T_2
  $T_1 \cap T_2 = \emptyset$
- *Find*(x): Finde den Repräsentanten der (größten) Teilmenge, in der sich x gerade befindet.

5.2 Union/Find-Datenstruktur

5.2.1 Intrees

1. Initialisierung: $x \longrightarrow \bullet x$: Mache x zur Wurzel eines neuen (einelementigen) Baumes.
2. *Union*(T_1, T_2):



3. *Find*: Suche Wurzel des Baumes, in dem sich x befindet.

Bemerkung: Naive Implementation: worst-case-Tiefe $= n$

- Zeit für *Find* $= \Omega(n)$
- Zeit für *Union* $= \mathcal{O}(1)$

5.2.2 Gewichtete Union (erste Verbesserung)

Mache die Wurzel des kleineren Baumes zu einem Kind der Wurzel des größeren Baumes. Die Tiefe des Baumes ist dann $\mathcal{O}(\log n)$.

- Zeit für *Find* $= \mathcal{O}(\log n)$
- Zeit für *Union* $= \mathcal{O}(1)$

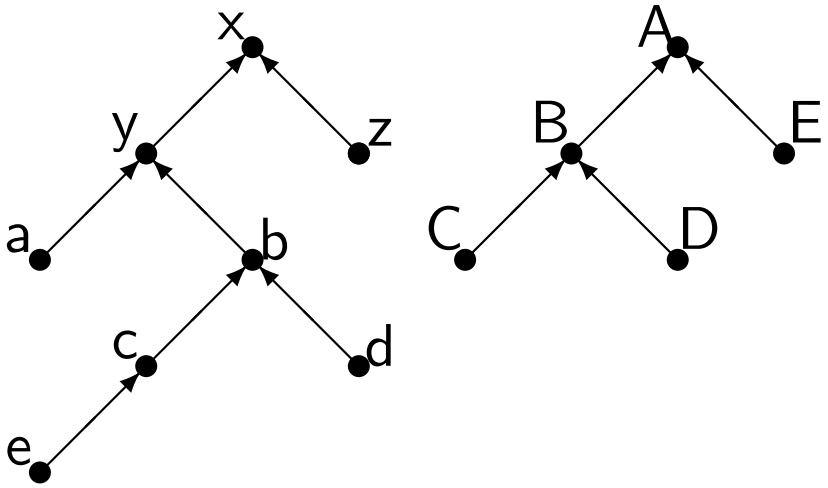
Es gilt auch: Tiefe des Baumes im worst-case:

$$\boxed{\Omega(\log n)}$$

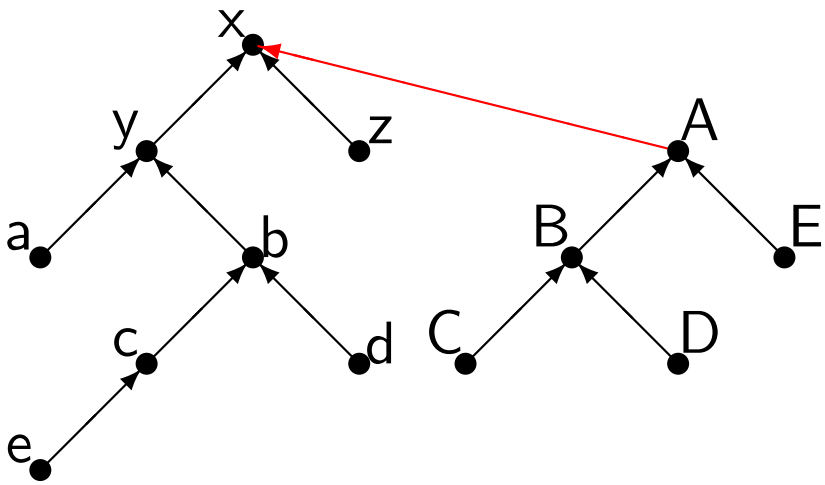
5.2.3 Pfad-Kompression mit gewichteter Union (zweite Verbesserung)

Wir betrachten eine Folge von k *Find*- und *Union*-Operationen auf einer Menge mit n Elementen, darunter $n-1$ *Union*.

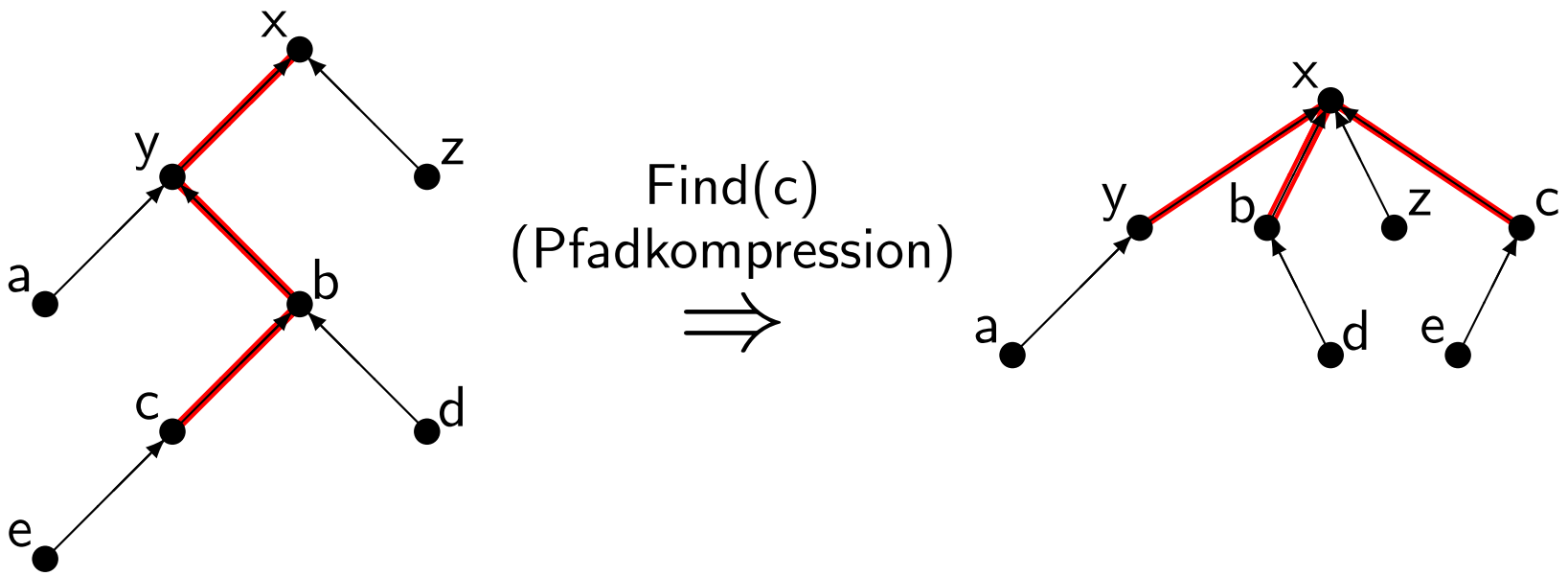
Implementierung: Gewichtete *Union* für Pfad-Kompression:



Union $\Rightarrow$



Implementierung: *Find* für Pfad-Kompression:



Bemerkung:
Nach Definition ist

$$\log^* n = \min\{i \geq 0; \underbrace{\log\log\log\ldots\log n}_{i \text{ log 's}} \leq 1\}$$

Beispiel 181

$$\log^* 0 = \log^* 1 = 0$$
$$\log^* 2 = 1$$
$$\log^* 3 = 2$$
$$\log^* 16 = 3 \qquad \text{da } 16 = 2^{2^2}$$
$$\log^* 2^{65536} = 5 \qquad \text{da } 2^{65536} = 2^{2^{2^{2^2}}}$$

Satz 182

Bei der obigen Implementierung ergibt sich eine amortisierte Komplexität von $\mathcal{O}(\log^* n)$ pro Operation.

Beweis:

(der Vollständigkeit halber, gehört nicht zum Vorlesungsstoff!)

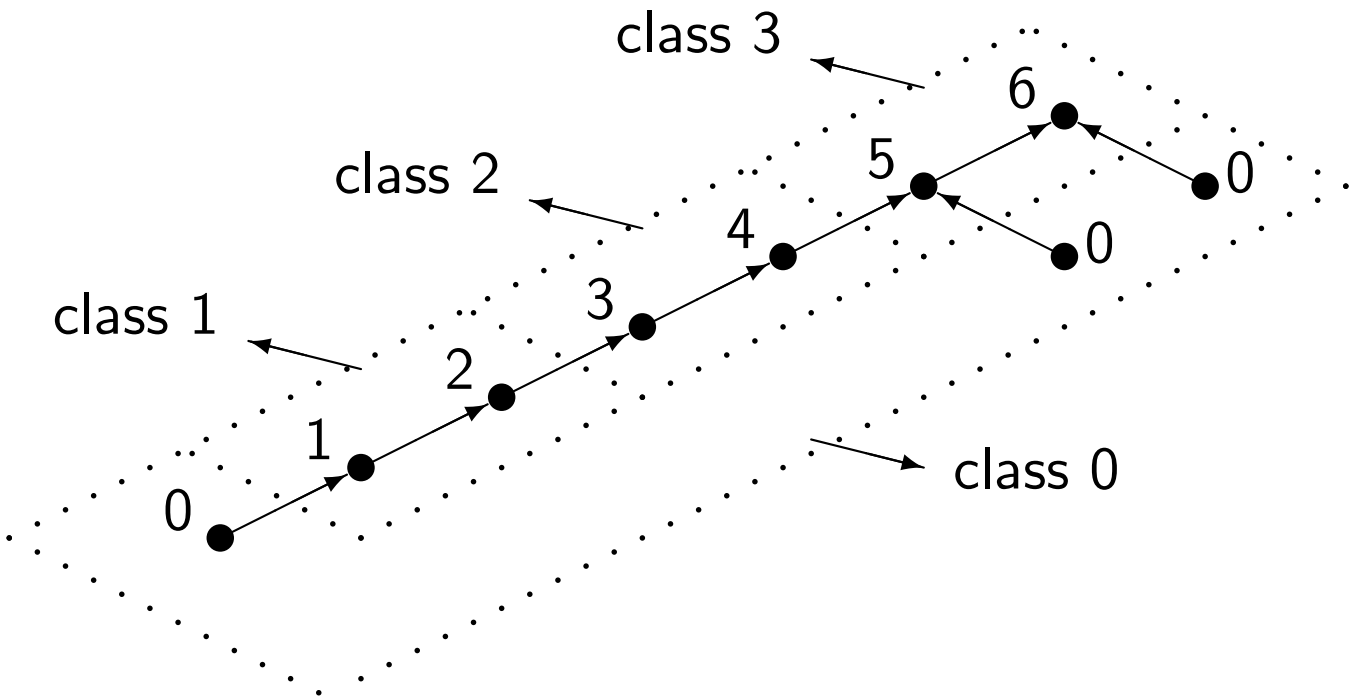
Sei T' der (endgültige) In-Baum, der durch die Folge der *Union*'s, ohne die *Find*'s, entstehen würde (also keine Pfad-Kompression).

Ordne jedem Element x drei Werte zu:

- rank(x):= Höhe des Unterbaums in T' mit Wurzel x
- class(x):= $\begin{cases} i \geq 1 & \text{falls } a_{i-1} < \text{ rank}(x) \leq a_i \text{ ist } (i \geq 1) \\ 0 & \text{falls rank}(x) = 0 \end{cases}$

  Dabei gilt: $a_0 = 0, a_i = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{i \ 2'\text{en}}$ für $i \geq 1$.

  Setze zusätzlich $a_{-1} := -1$.


class 3
class 2
class 1
class 0
0
1
2
3
4
5
6
0
0

Beweis (Forts.):

- dist(x) ist die Distanz von x zu einem Vorfahr y im momentanen Union/Find-Baum (mit Pfad-Kompression), so dass class(y) $>$ class(x) bzw. y die Wurzel des Baumes ist.

Definiere die Potenzialfunktion

$$\text{Potenzial} := c \sum_x \text{dist}(x), \quad c \text{ eine geeignete Konstante } > 0$$

Beweis (Forts.):

Beobachtungen:

i) Sei T ein Baum in der aktuellen Union/Find-Struktur (mit Pfad-Kompression), seien x, y Knoten in T, y Vater von x. Dann ist class$(x) \leq$ class(y).

ii) Aufeinander folgende *Find*(x) durchlaufen (bis auf eine) verschiedene Kanten. Diese Kanten sind (im wesentlichen) eine Teilfolge der Kanten in T' auf dem Pfad von x zur Wurzel.

Beweis (Forts.):

Amortisierte Kosten Find(x):

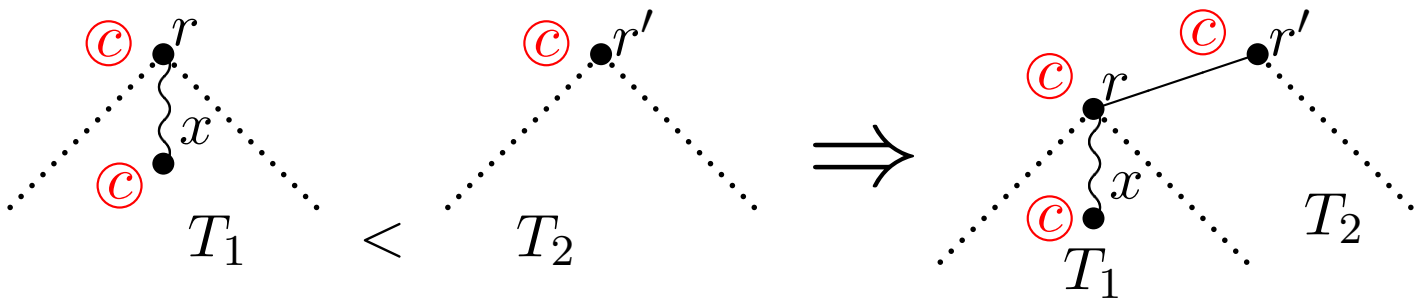
Sei $x_0 \to x_1 \to x_2 \ldots x_k = r$ der Pfad von x_0 zur Wurzel. Es gibt höchstens $\log^* n$-Kanten (x_{i-1}, x_i) mit class$(x_{i-1}) <$ class(x_i). Ist class$(x_{i-1}) =$ class(x_i) und $i < k$ (also $x_i \neq r$), dann ist dist(x_{i-1}) vor der *Find(x)*-Operation ≥ 2, nachher gleich 1.

Damit können die Kosten für alle Kanten (x_{i-1}, x_i) mit class(x_{i-1}) $=$ class(x_i) aus der Potenzialverringerung bezahlt werden. Es ergeben sich damit amortisierte Kosten

$$\boxed{\mathcal{O}(\log^* n)}$$

Beweis (Forts.):

Amortisierte Gesamtkosten aller $(n-1)$-Union's:



Die gesamte Potenzialerhöhung durch alle *Union*'s ist nach oben durch das Potenzial von T' beschränkt (Beobachtung ii).

Beweis (Forts.):

$$\begin{aligned}
\text{Potenzial}(T') &\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \text{dist}(x) \\
&\leq c \cdot \sum_{i=0}^{\log^* n} \sum_{\text{rank}(x)=j=a_{i-1}+1}^{a_i} \frac{n}{2^j} a_i \\
&\leq c \cdot n \sum_{i=0}^{\log^* n} a_i \frac{1}{2^{a_{i-1}}} = c \cdot n \sum_{i=0}^{\log^* n} 1 \\
&= \mathcal{O}(n \log^* n) \,.
\end{aligned}$$

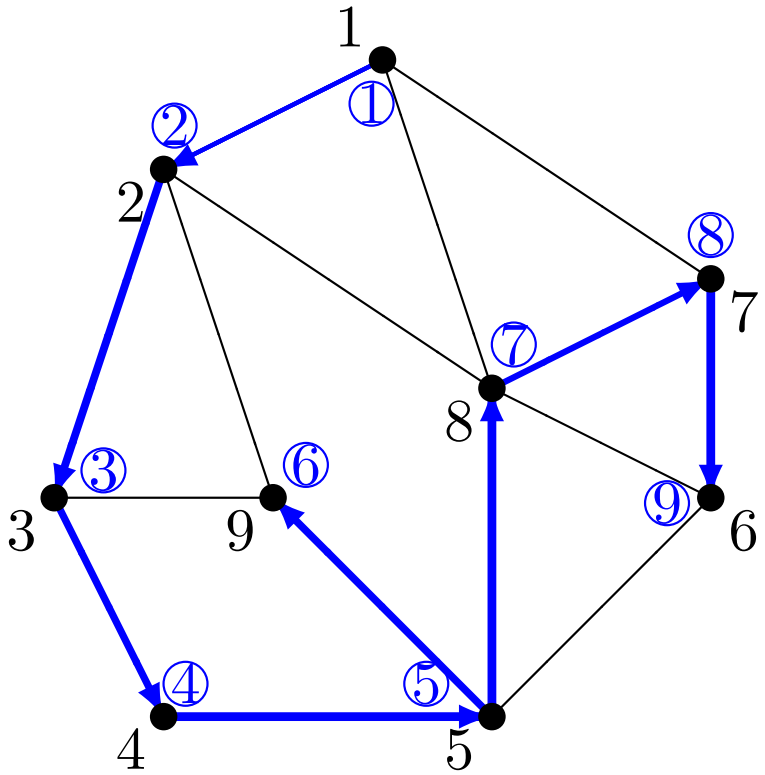
Die zweite Ungleichung ergibt sich, da alle Unterbäume, deren Wurzel x $\text{rank}(x) = j$ hat, disjunkt sind und jeweils $\geq 2^j$ Knoten enthalten. □

6. Traversierung von Graphen

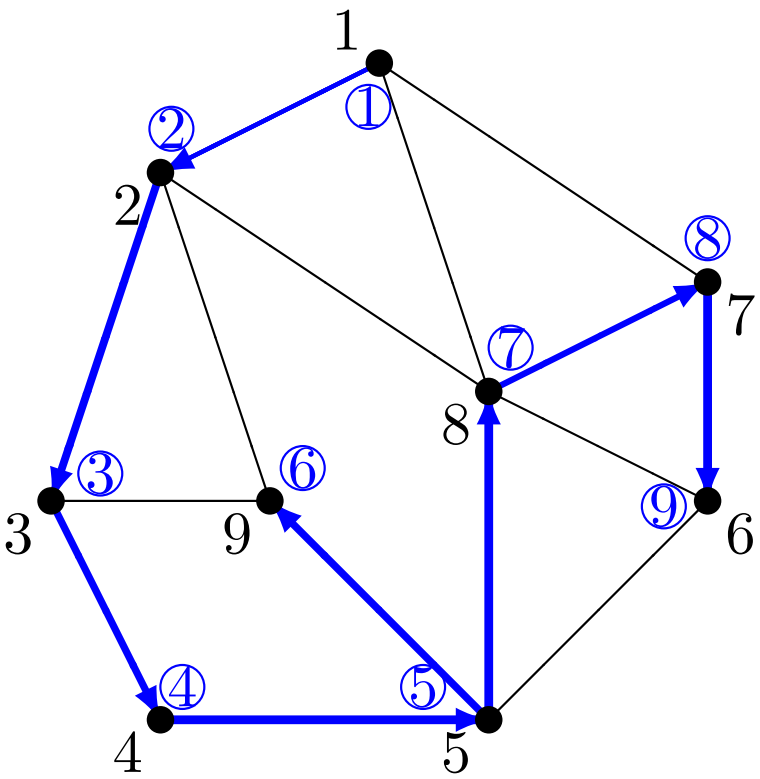
Sei $G = (V, E)$ ein ungerichteter Graph. Anhand eines Beipiels betrachten wir die zwei Algorithmen DFS (Tiefensuche) und BFS (Breitensuche).

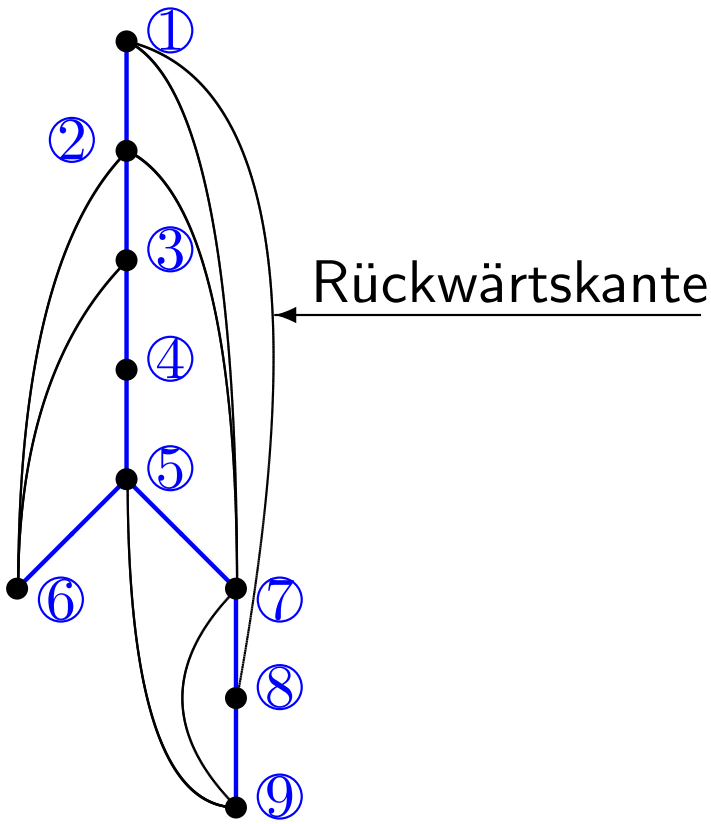
```
while ∃ unvisited v do
    r := pick (random) unvisited node
    push r onto stack
    while stack ≠ ∅ do
        v := pop top element
        if v unvisited then
            mark v visited
            push all neighbours of v onto stack
            perform operations DFS_Ops(v)
        fi
    od
od
```

Beispiel 183



Beobachtung: Die markierten Kanten bilden einen Spannbaum:

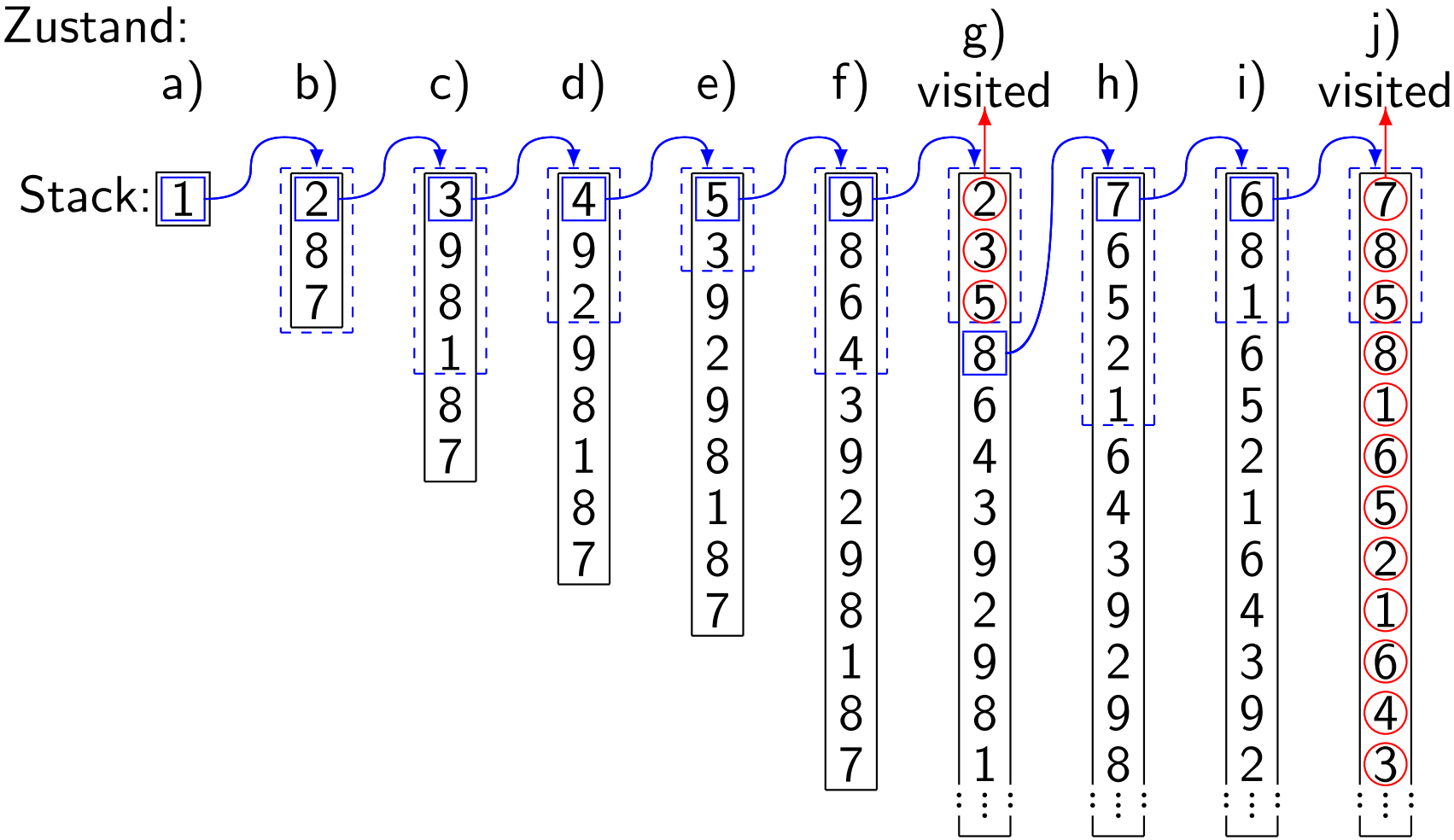




Folge der Stackzustände

: oberstes Stackelement : Nachbarn

: schon besuchte Knoten



Wir betrachten den Stackzustand:
Im Zustand g) sind die Elemente 2, 3 und 5 als visited markiert (siehe Zustände b), c) und e)). Deswegen werden sie aus dem Stack entfernt, und das Element 8 wird zum obersten Stackelement. Im Zustand j) sind alle Elemente markiert, so dass eins nach dem anderen aus dem Stack entfernt wird.