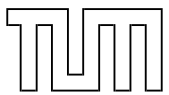
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010

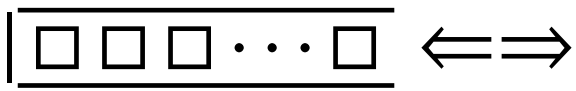


Übersicht

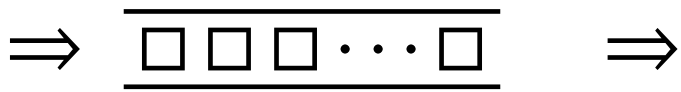
Stacks und Queues

Grundlegende sequenzbasierte Datenstrukturen:

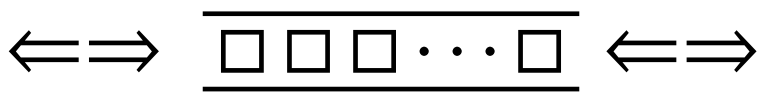
- Stack (Stapel)



- (FIFO-)Queue (Schlange)



- Deque (double-ended queue)



Stacks und Queues

Stack-Methoden:

- pushBack (bzw. push)
- popBack (bzw. pop)
- last (bzw. top)

Queue-Methoden:

- pushBack
- popFront
- first

Stacks und Queues

Warum spezielle Sequenz-Typen betrachten, wenn wir mit der bekannten Datenstruktur für Listen schon alle benötigten Operationen in $\mathcal{O}(1)$ haben?

- Programme werden lesbarer und einfacher zu debuggen, wenn spezialisierte Zugriffsmuster explizit gemacht werden.
- Einfachere Interfaces erlauben eine größere Breite von konkreten Implementationen (hier z.B. platzsparendere als Listen).
- Listen sind ungünstig, wenn die Operationen auf dem Sekundärspeicher (Festplatte) ausgeführt werden.

  Sequentielle Zugriffsmuster können bei entsprechender Implementation (hier z.B. als Arrays) stark vom Cache profitieren.

Stacks und Queues

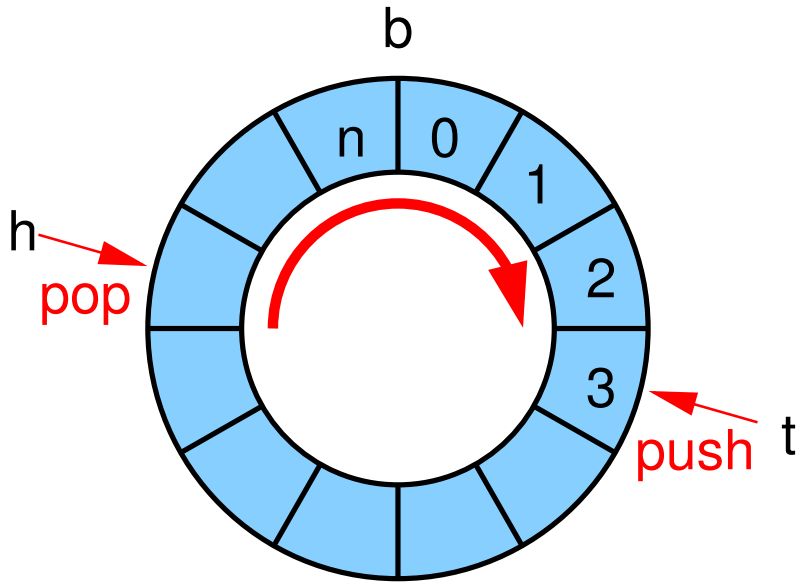
Spezielle Umsetzungen:

- Stacks mit beschränkter Größe $\Rightarrow$ bounded arrays
- unbounded stacks $\Rightarrow$ unbounded arrays
- oder: Stacks als einfach verkettete Listen
  (top of stack = front of list)
- (FIFO-)Queues: einfach verkettete Listen mit Zeiger auf letztes Element (eingefügt wird am Listenende, entnommen am Listenanfang, denn beim Entnehmen muss der Nachfolger bestimmt werden)
- Für Deques braucht man doppelt verkettete Listen, einfach verkettete reichen hier nicht.

Beschränkte FIFO-Queues

```
class BoundedFIFO<Elem> {
    Elem[] b;
    int h=0;              // erstes Element
    int t=0;        // erster freier Eintrag
}
```

- Queue besteht aus den Feldelementen $h \ldots t-1$
- Es bleibt immer mindestens ein Feldelement frei (zur Unterscheidung zwischen voller und leerer Queue)



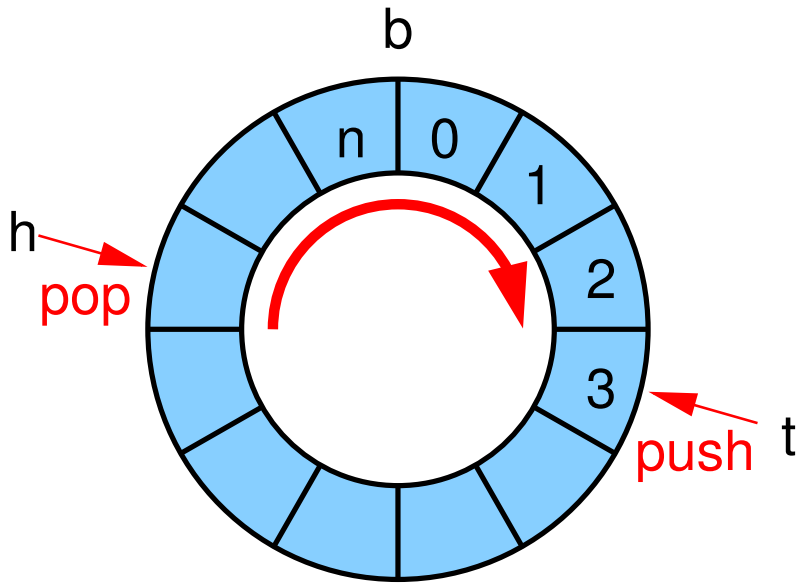
Beschränkte FIFO-Queues

```
...

boolean isEmpty() {
    return (h==t);
}

Elem first() {
    return b[h];
}

int size() {
    return (t-h+n+1)%(n+1);
}
```



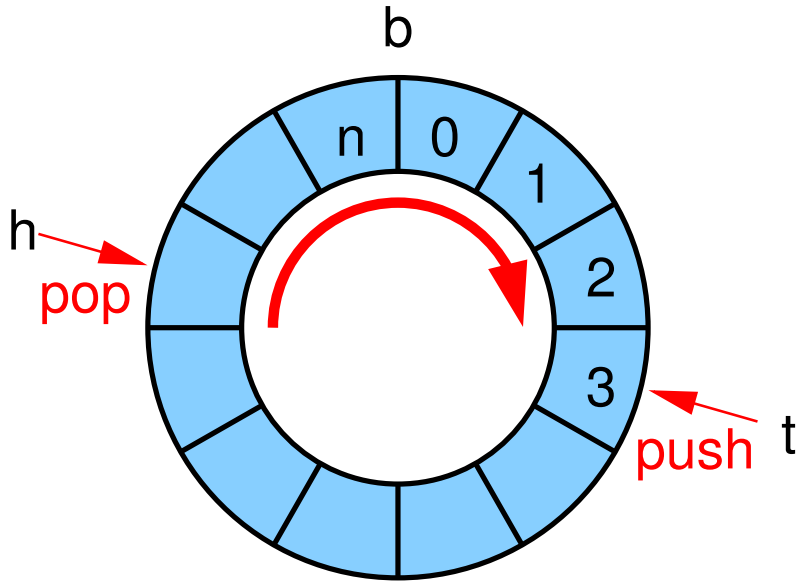
Beschränkte FIFO-Queues

```
...

void pushBack(Elem x) {
    assert(size()<n);
    b[t]=x;
    t=(t+1)%(n+1);
}

void popFront() {
    assert(!isEmpty());
    h=(h+1)%(n+1);
}

int size() {
    return (t-h+n+1)%(n+1);
}
```



Queues

- Struktur kann auch als Deque verwendet werden
- Zirkuläre Arrays erlauben auch den indexierten Zugriff [. . .]
- Bounded queues/deques können mit ähnlichen Techniken zu Unbounded queues/deques erweitert werden wie bei Unbounded arrays

Fazit Sequenzstrukturen

- Listen sind sehr flexibel beim Einfügen von Elementen in der Mitte
- Felder können in konstanter Zeit auf beliebige Elemente zugreifen
- Listen haben im Gegensatz zu Arrays kein Reallokationsproblem bei unbeschränkten Größen
- beide Datenstrukturen sind einfach, aber manchmal nicht wirklich zufriedenstellend

Beispiel: Sortierte Sequenz

Wozu?

- binäre Suche in $\log n$ Schritten möglich

Realisierung als Liste

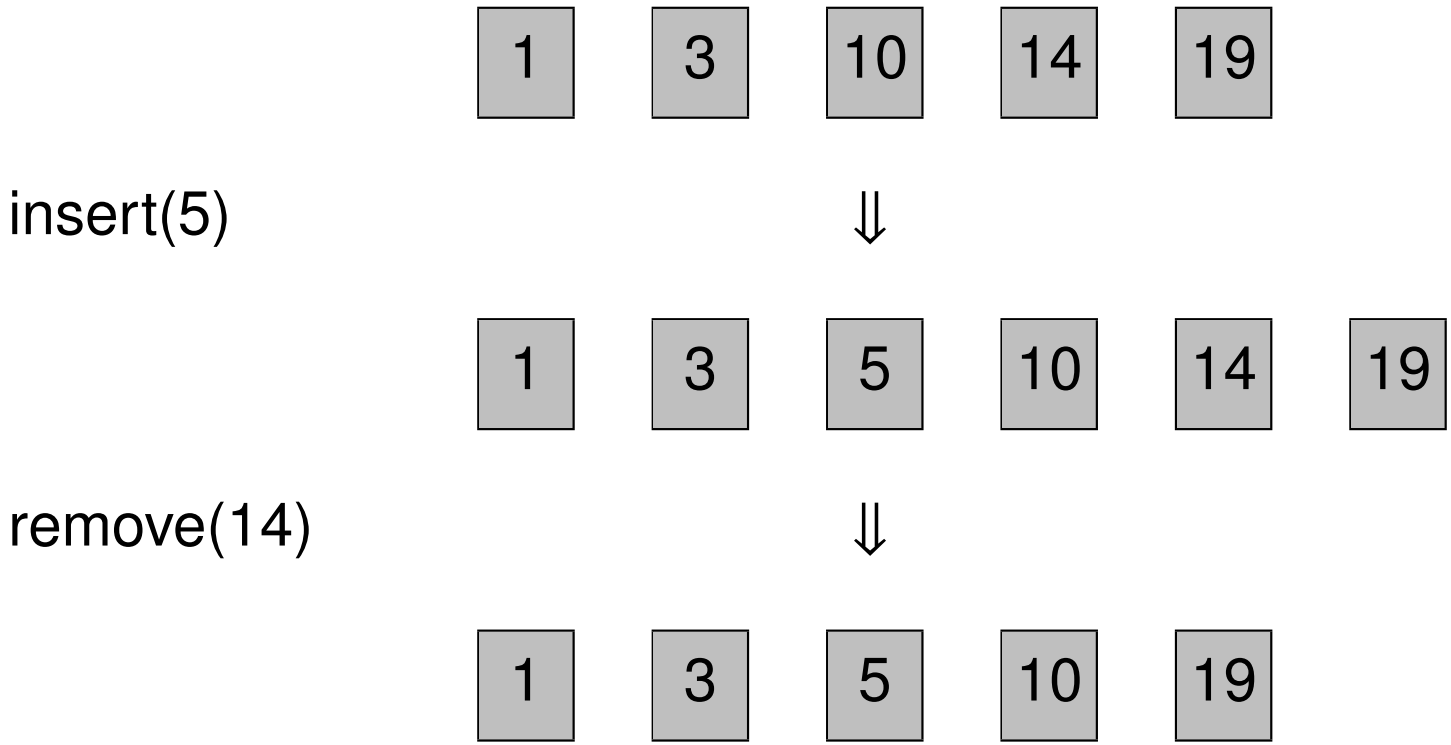
- insert, remove und find auf Sequenz der Länge n kosten im worst case $\Theta(n)$ Zeit

Realisierung als Feld

- insert und remove kosten im worst case $\Theta(n)$ Zeit
- find kann mit binärer Suche in $\log n$) im worst case realisiert werden

Beispiel: Sortierte Sequenz

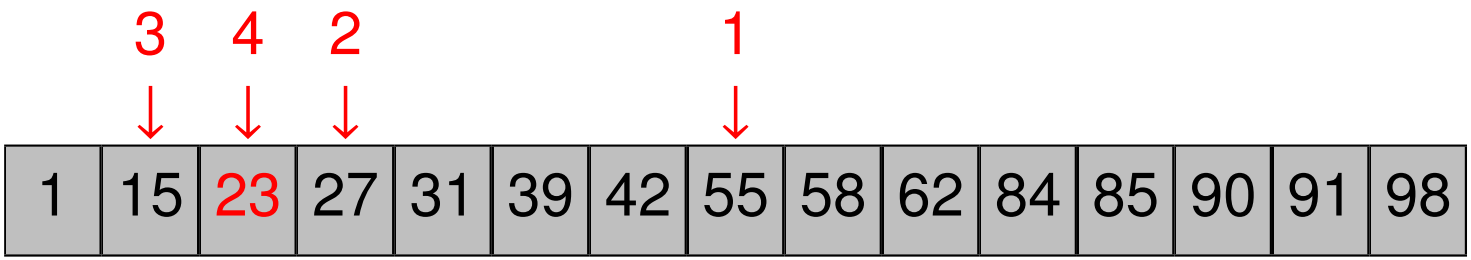
Problem:
Aufrechterhaltung der Sortierung nach jeder Einfügung / Löschung



Beispiel: Sortierte Sequenz

Warum sortierte Sequenz?
$\Rightarrow$ binäre Suche möglich

find(23):



In n-elementiger Sequenz kann ein beliebiges Element in maximal $\log_2 n$ Schritten gefunden werden

Beispiel: Sortierte Sequenz

S: sortierte Sequenz

Jedes Element e identifiziert über key(e)

Operationen:

- $\langle e_0, \ldots, e_n \rangle$.insert$(e) = \langle e_0, \ldots, e_i, e, e_{i+1}, \ldots, e_n \rangle$
  für das i mit key$(e_i) <$ key$(e) <$ key(e_{i+1})
- $\langle e_0, \ldots, e_n \rangle$.remove$(k) = \langle e_0, \ldots, e_{i-1}, e_{i+1}, \ldots, e_n \rangle$
  für das i mit key$(e_i) = k$
- $\langle e_0, \ldots, e_n \rangle$.find$(k) = e_i$
  für das i mit key$(e_i) = k$

Beispiel: Sortierte Sequenz

Realisierung als Liste:

- insert, remove und find auf Sequenz der Länge n kosten im worst case $\Theta(n)$ Zeit

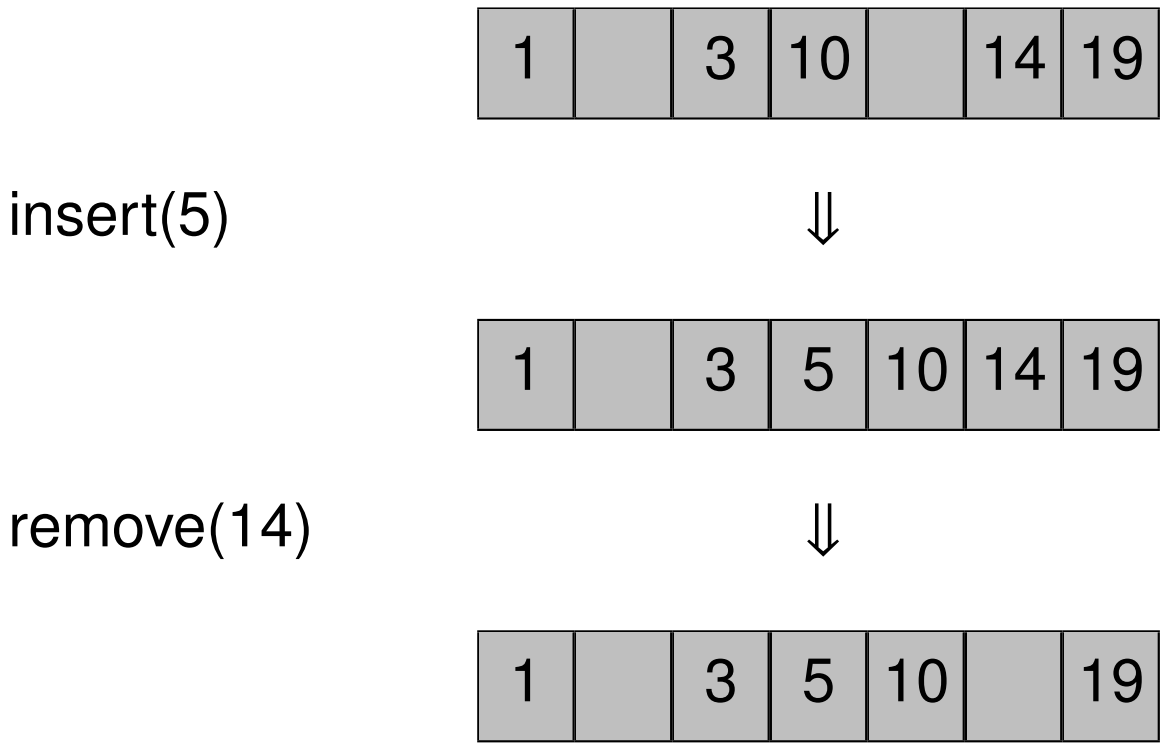
Realisierung als Feld:

- insert und remove kosten im worst case $\Theta(n)$ Zeit
- find kostet mit binärer Suche im worst case nur $\mathcal{O}(\log n)$ Zeit

Beispiel: Sortierte Sequenz

Kann man insert und remove besser mit einem Feld realisieren?

$\Rightarrow$ Vorbild Bibliothek: Lücken lassen



Beispiel: Sortierte Sequenz

Geschickte Verteilung der Lücken:

$\Rightarrow$ amortisierte Kosten für insert und remove $\Theta(\log^2 n)$

Übersicht

Wörterbuch-Datenstruktur

- S: Menge von Elementen
- Element e wird identifiziert über eindeutigen Schlüssel key(e)

Operationen:

- S.insert(Elem e): $S = S \cup \{e\}$
- S.remove(Key k): $S = S \setminus \{e\}$,
  wobei e das Element mit key(e) $==$ k ist
- S.find(Key k):
  gibt das Element $e \in S$ mit key(e) $==$ k zurück, falls es existiert, sonst null

Wörterbuch-Datenstruktur

Liste und Feld sind uns auch hier zu langsam:

Realisierung als (sortierte) Liste:

- insert, remove und find auf Sequenz der Länge n kosten im worst case $\Theta(n)$ Zeit

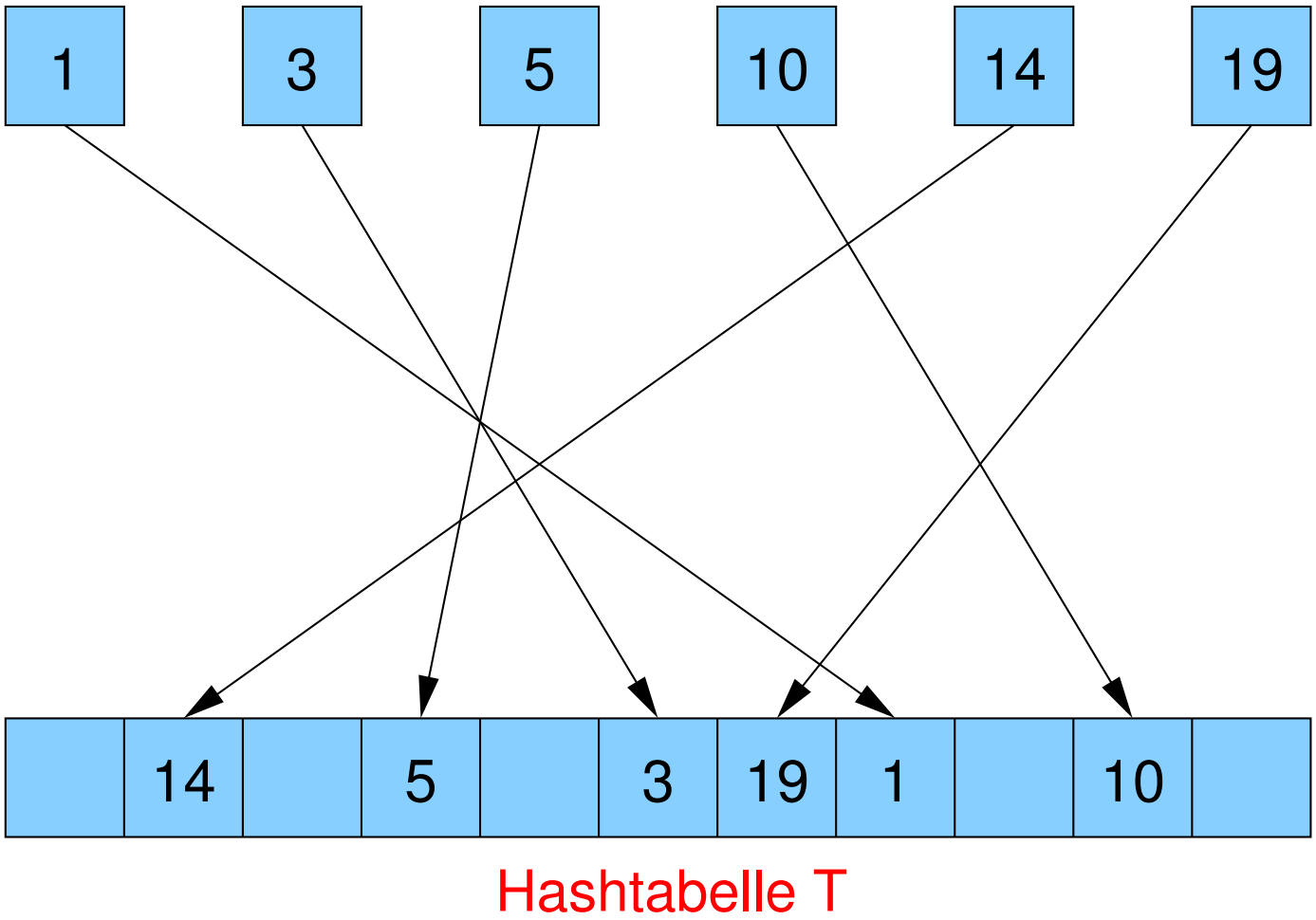
Realisierung als (sortiertes) Feld:

- insert und remove kosten im worst case $\Theta(n)$ Zeit
- find kostet mit binärer Suche im worst case nur $\mathcal{O}(\log n)$ Zeit

Hashfunktion und Hashtabelle

Hashfunktion

$h : U \mapsto T$



Hashfunktion

Anforderungen:

- schneller Zugriff (Zeiteffizienz)
- platzsparend (Speichereffizienz)
- gute Streuung bzw. Verteilung der Elemente über die ganze Tabelle
- unter optimalen Umständen können insert, remove und find in konstanter Zeit ausgeführt werden

Hashing

Annahme: perfekte Streuung

```
void insert(Elem e) {
    T[h(key(e))] = e;
}

void remove(Key k) {
    T[h(k)] = null;
}

Elem find(Key k) {
    return T[h(k)];
}
```

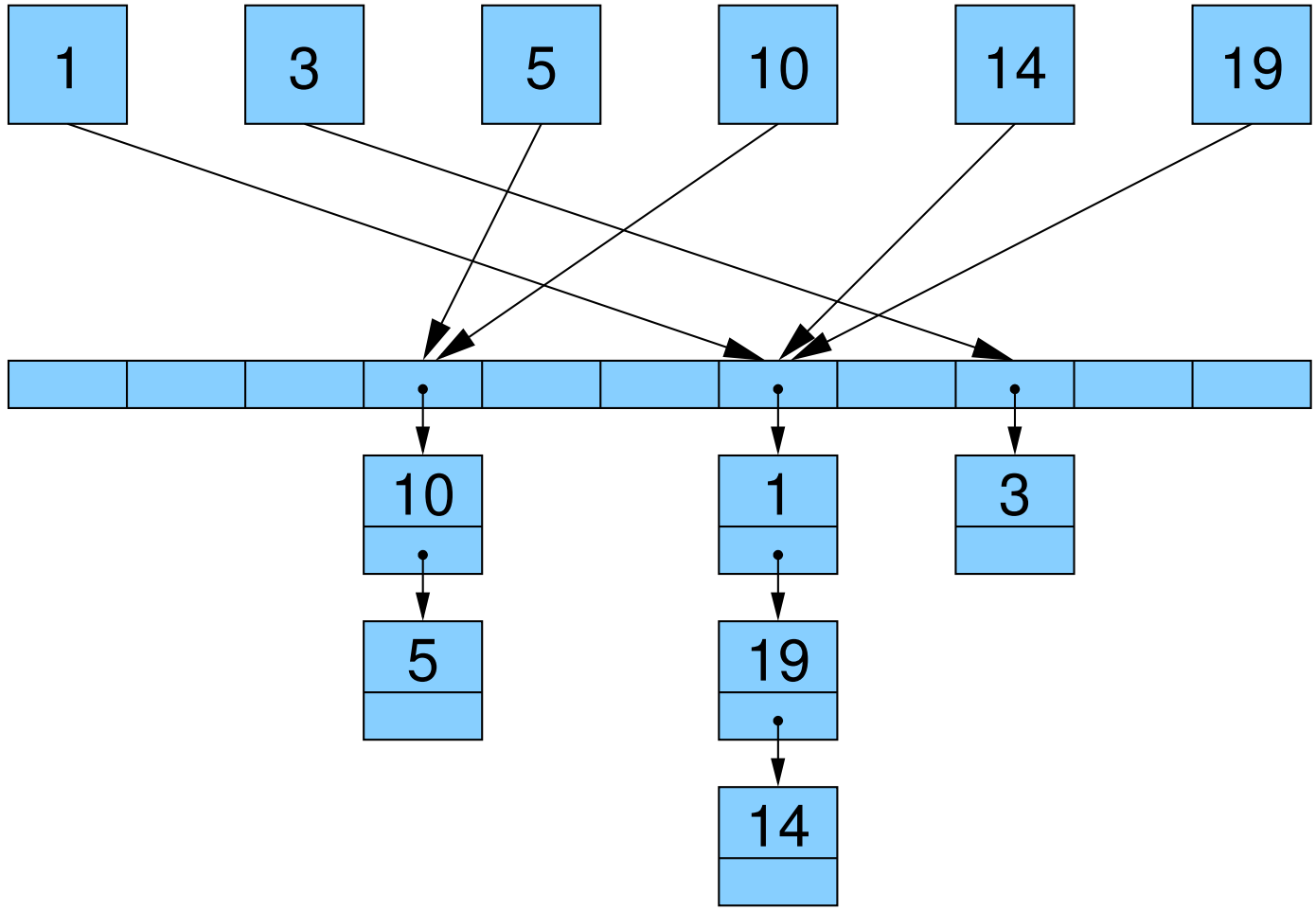
Hashing

Wörterbuch

- statisch: nur find
- dynamisch: insert, remove und find

Dynamisches Wörterbuch

Hashing with Chaining:



unsortierte verkettete Listen
(Ziel: Listen möglichst kurz)

Dynamisches Wörterbuch

Hashing with Chaining:

List<Elem>[ ] T

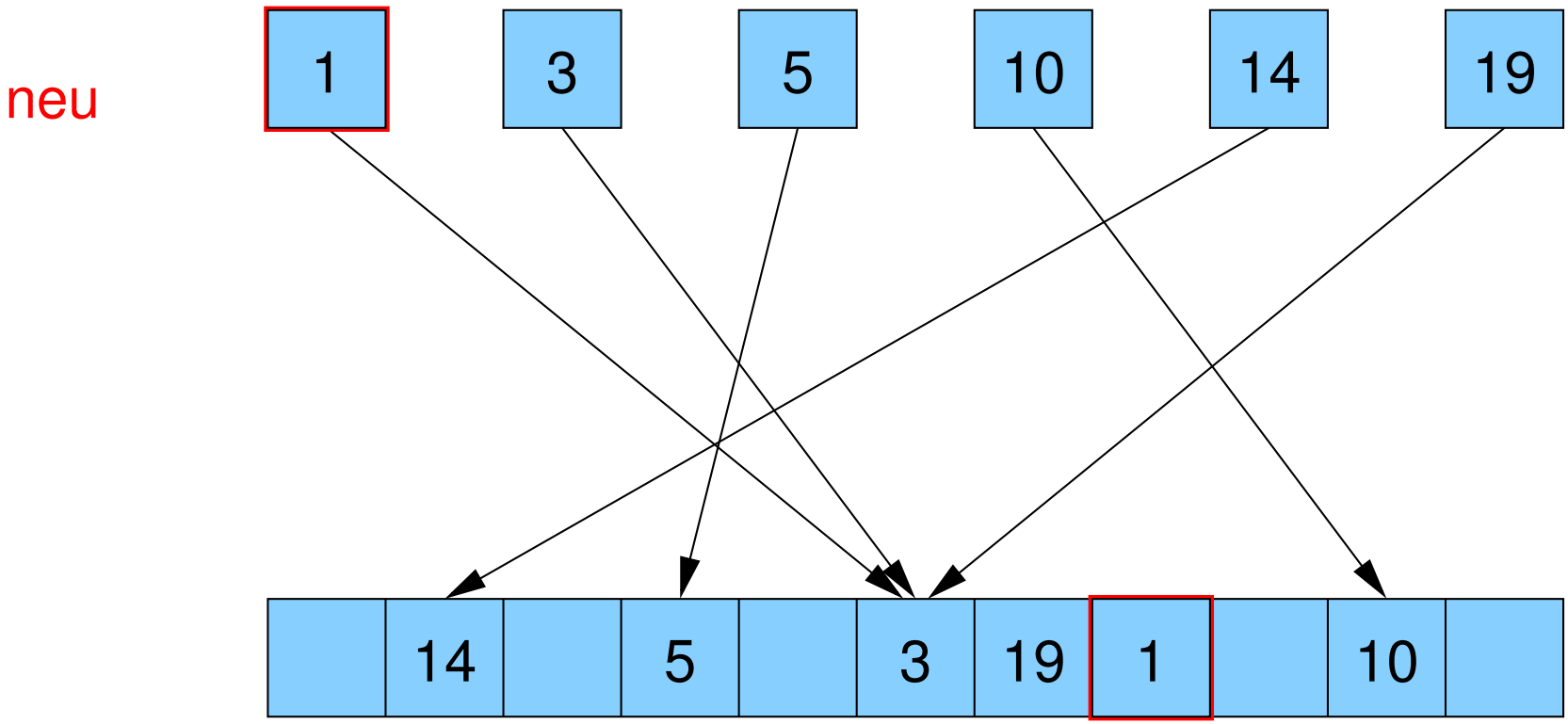
```
void insert(Elem e) {
    T[h(key(e))].insert(e);
}

void remove(Key k) {
    T[h(k)].remove(k);
}

void find(Key k) {
    T[h(k)].find(k);
}
```

Dynamisches Wörterbuch

Hashing with Linear Probing:



Speichere Element e im ersten freien Ort T[i], T[$i+1$], T[$i+2$], ... mit $i == h(\text{key}(e))$
(Ziel: Folgen besetzter Positionen möglichst kurz)

Hashing with Chaining

- Platzverbrauch: $\mathcal{O}(n+m)$
- insert benötigt konstante Zeit
- remove und find müssen u.U. eine ganze Liste scannen
- im worst case sind alle Elemente in dieser Liste

$\Rightarrow$ im worst case ist Hashing with chaining nicht besser als eine normale Liste

Hashing with Chaining

Gibt es Hashfunktionen, die garantieren, dass alle Listen kurz sind?

- nein, für jede Hashfunktion gibt es eine Menge von mindestens N/m Schlüsseln, die demselben Tabelleneintrag zugeordnet werden
- meistens ist $n < N/m$ und in diesem Fall kann die Suche immer zum Scan aller Elemente entarten

$\Rightarrow$ Auswege

- Average-case-Analyse
- Randomisierung
- Änderung des Algorithmus (z.B. abhängig von aktuellen Schlüsseln)

Hashing with Chaining

Betrachte als Hashfunktionsmenge die Menge aller Funktionen, die die Schlüsselmenge (mit Kardinalität N) auf die Zahlen $0, \ldots, m-1$ abbilden

Satz

Falls n Elemente in einer Hashtabelle T der Größe m mittels einer zufälligen Hashfunktion h gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $\mathcal{O}(1 + n/m)$.

Unrealistisch: es gibt m^N solche Funktionen und demzufolge braucht man $N \log m$ Bits, um eine Funktion zu spezifizieren

$\Rightarrow$ widerspricht dem Ziel, den Speicherverbrauch von N auf n zu senken!

Hashing with Chaining

Beweis.

- Betrachte feste Position i
- Zugriffszeit ist $O(1 + \text{Länge der Liste } T[i])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$
- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_e X_e$
- Erwartete Listenlänge

$$\begin{aligned} \mathbb{E}[X] &= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\ &= \sum_{e \in S} \Pr[X_e = 1] = \sum_{e \in S} 1/m = n/m \end{aligned}$$

□

Hashing with Chaining

Wie konstruiert man zufällige Hashfunktionen?

Definition

Sei c eine positive Konstante.

Eine Familie H von Hashfunktionen auf $\{0, \ldots, m-1\}$ heißt c-universell, falls für jedes Paar $x \neq y$ von Schlüsseln gilt, dass

$$\left|\{h \in H : \ h(x) = h(y)\}\right| \leq \frac{c}{m}|H|$$

D.h. für eine zufällige Hashfunktion $h \in H$ gilt

$$\Pr[h(x) = h(y)] \leq \frac{c}{m}$$

Hashing with Chaining

Satz

Falls n Elemente in einer Hashtabelle T der Größe m mittels einer zufälligen Hashfunktion h aus einer c-universellen Familie gespeichert werden, dann ist die erwartete Laufzeit von remove bzw. find in $O(1 + c \cdot n/m)$.

Beweis.

- Betrachte festen Schlüssel k
- Zugriffszeit ist $O(1 +$ Länge der Liste $T[h(k)])$
- Zufallsvariable $X_e \in \{0, 1\}$ für jedes $e \in S$ zeigt an, ob e auf die gleiche Position wie k gehasht wird

□

Hashing with Chaining

Beweis.

- $X_e = 1 \Leftrightarrow h(\text{key}(e)) = i$
- Listenlänge $X = \sum_e X_e$
- Erwartete Listenlänge

$$\begin{aligned}
\mathbb{E}[X] &= \mathbb{E}\left[\sum_{e \in S} X_e\right] \\
&= \sum_{e \in S} \mathbb{E}[X_e] = \sum_{e \in S} 0 \cdot \Pr[X_e = 0] + 1 \cdot \Pr[X_e = 1] \\
&= \sum_{e \in S} \Pr[X_e = 1] \leq \sum_{e \in S} c/m = n \cdot c/m
\end{aligned}$$