

Proseminar  
Effiziente Algorithmen  
Kapitel 2: Datenstrukturen

Prof. Dr. Christian Scheideler  
WS 2021

# Übersicht

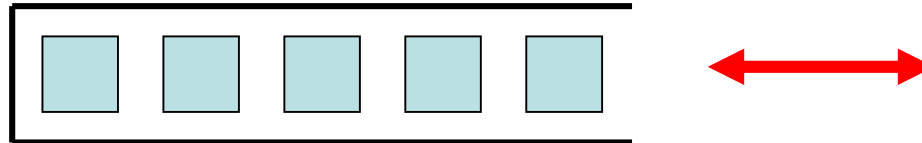
- Stacks
- Queues
- Suchbäume
- Hashtabellen
- Priority Queues
- Mengen
- Probleme im Bereich Datenstrukturen

# Stacks (Stapel) und Queues (Schlangen)

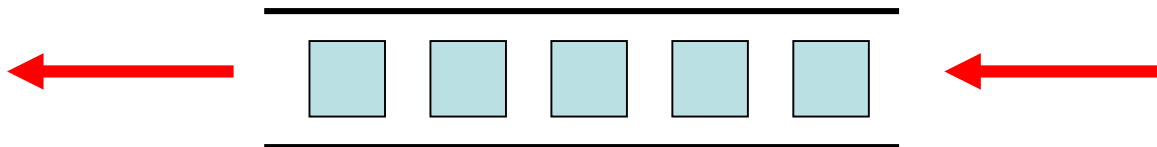
---

## Definition:

1. **Stacks (Stapel)** sind eine Datenstruktur, die die LIFO (last-in-first-out) Regel implementiert.  
Bei der LIFO Regel soll das zuletzt eingefügte Objekt entfernt werden.



2. **Queues (Schlangen)** sind eine Datenstruktur, die die FIFO (first-in-first-out) Regel implementiert.  
Bei der FIFO Regel soll das am längsten in der Menge befindliche Objekt entfernt werden.



# Stacks

---

- Einfügen eines Objekts wird bei Stacks **Push** genannt.
  - Entfernen des zuletzt eingefügten Objekts wird **Pop** genannt.
  - Zusätzliche Hilfsoperation ist **Stack-Empty**, die überprüft, ob ein Stack leer ist.
  - Stack mit maximal  $n$  Objekten wird realisiert durch ein Array  $S[1...n]$  mit einer zusätzlichen Variablen  $top[S]$ , die den Index des zuletzt eingefügten Objekts speichert.
  - Maximale Anzahl Objekte a priori nicht bekannt: verwende **dynamisches Array** für  $S$ .
-

# Queues

---

- Einfügen eines Objekts wird **Enqueue** genannt.
  - Entfernen des am längsten in der Queue befindlichen Objekts wird **Dequeue** genannt.
  - Zusätzliche Hilfsoperation ist **Queue-Empty**, die überprüft, ob eine Queue leer ist.
  - Queue mit maximal  $n-1$  Objekten wird realisiert durch ein Feld  $Q[1\dots n]$  mit zusätzlichen Variablen  $head[Q]$ ,  $tail[Q]$ .  
(Maximum nicht bekannt: verwende dynamisches Feld.)
  - $head[Q]$ : Position des am längsten in Queue befindlichen Objekts
  - $tail[Q]$ : erste freie Position.
  - Alle Indexberechnungen modulo  $n (+1)$ , betrachten Array kreisförmig.  
Auf Position  $n$  folgt wieder Position  $1$ .
-

# Binäre Suchbäume

---

## Binäre Suchbäume

- Verwende Binärbaum
- Speichere Schlüssel „geordnet“

## Binäre Suchbaumeigenschaft:

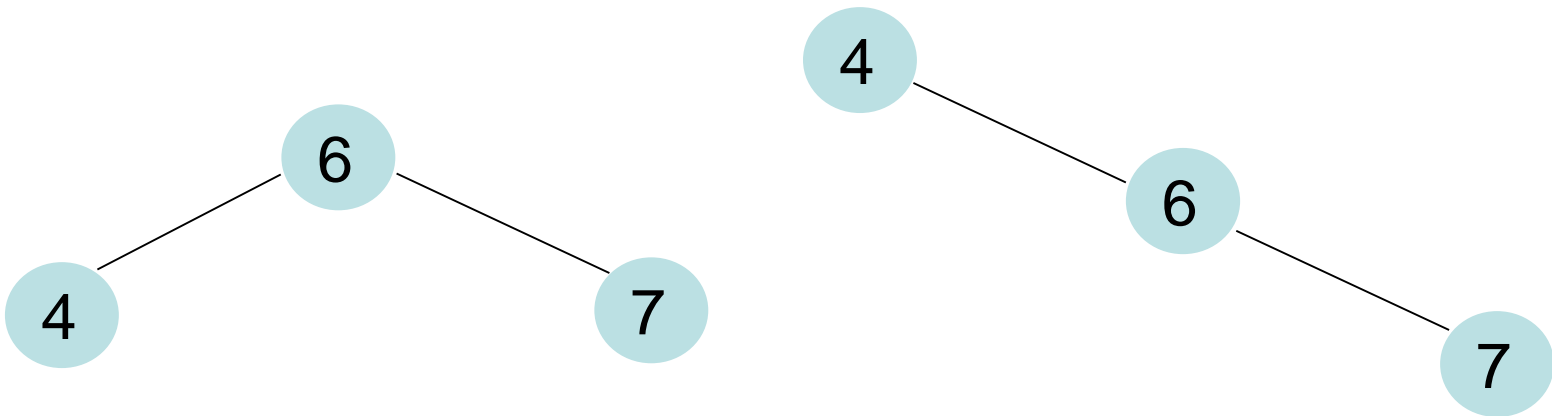
- Sei  $x$  Knoten im binären Suchbaum
- Ist  $y$  Knoten im **linken Unterbaum** von  $x$ , dann gilt  $\text{key}[y] \leq \text{key}[x]$
- Ist  $y$  Knoten im **rechten Unterbaum** von  $x$ , dann gilt  $\text{key}[y] \geq \text{key}[x]$

# Binäre Suchbäume

---

## Unterschiedliche Suchbäume

- Beispiel: Schlüsselmenge 4,6,7

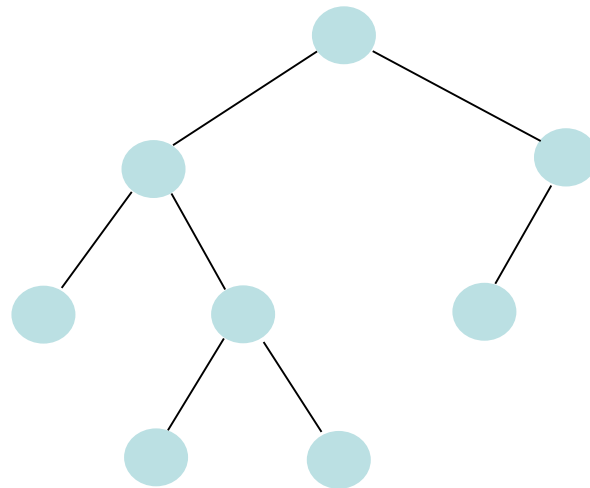


# Balancierte binäre Suchbäume

---

## AVL-Bäume

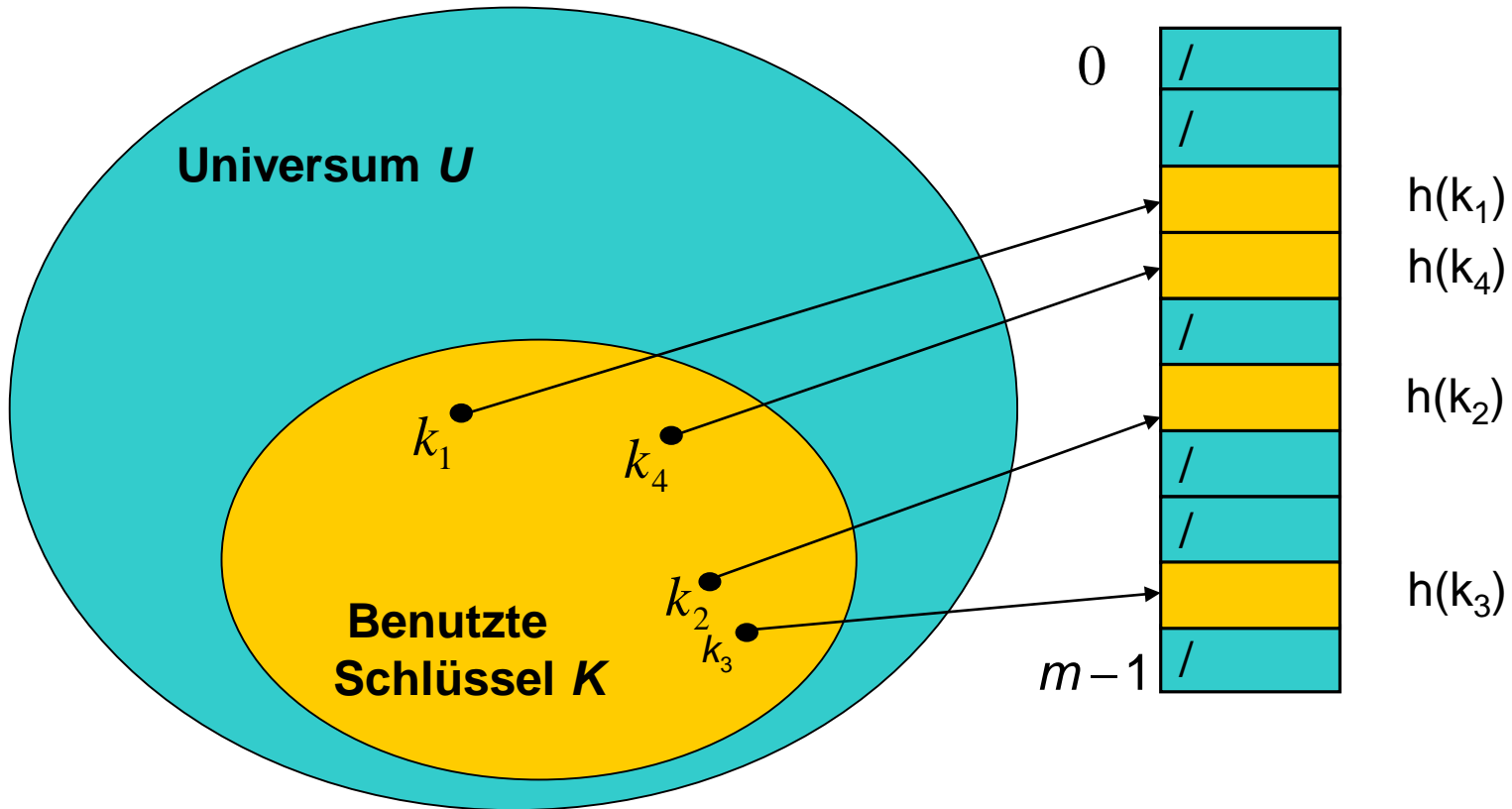
- Ein Baum heißt AVL-Baum, wenn **für jeden** Knoten gilt: Die Höhe seines linken und rechten Teilbaums unterscheidet sich **höchstens um 1**.





# Hashing (ohne Kollisionen)

---



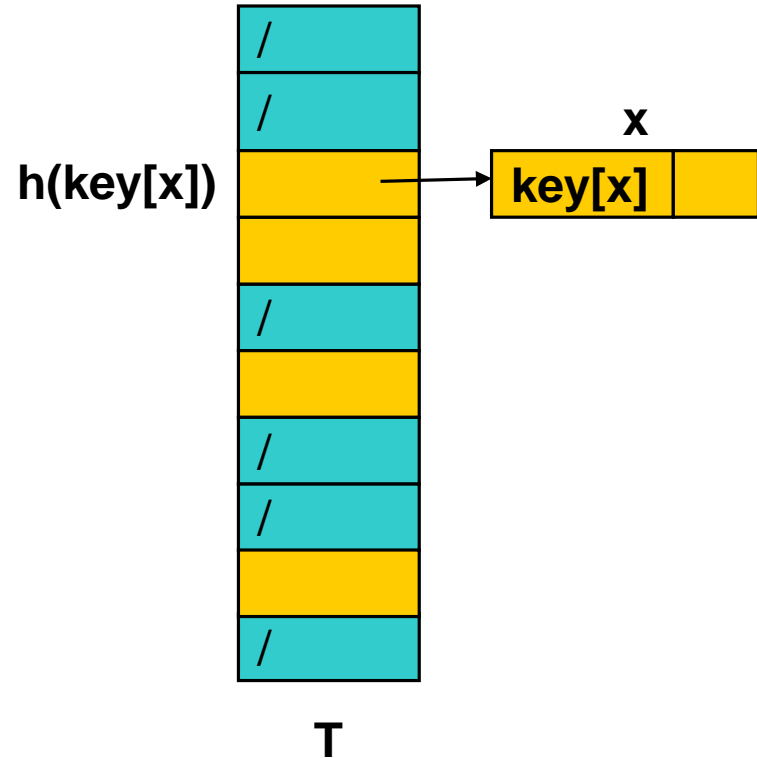
# Hashing (ohne Kollisionen)

---

Insert( $T, x$ ):  
 $T[h(\text{key}[x])] \leftarrow x$

Remove( $T, k$ ):  
if  $\text{key}[T[h(k)]] = k$  then  
 $T[h(k)] \leftarrow \text{NIL}$

Search( $T, k$ ):  
if  $\text{key}[T[h(k)]] = k$  then  
return  $T[h(k)]$   
else  
return  $\text{NIL}$



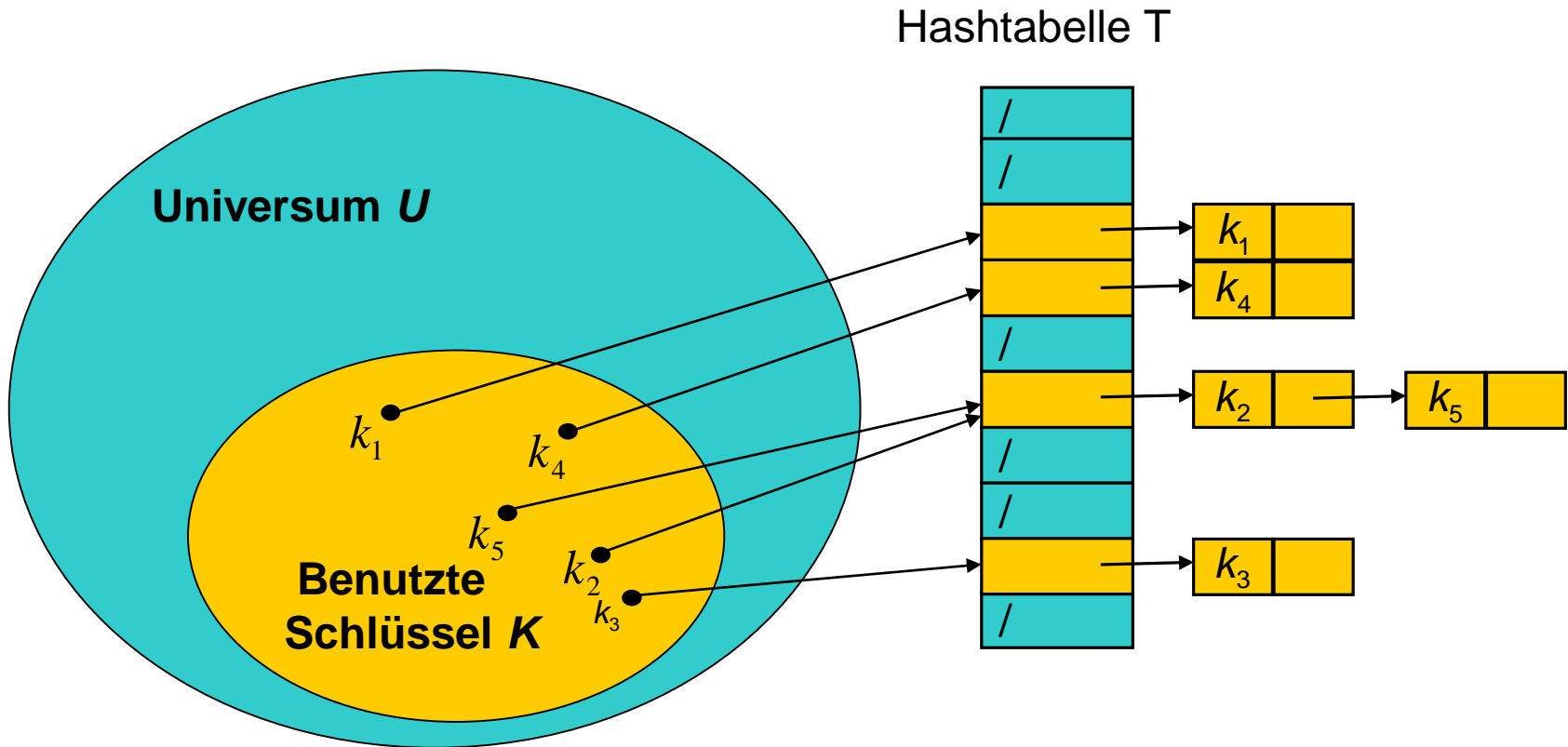
# Maßnahmen zur Kollisionsauflösung

---

Mögliche Strategien:

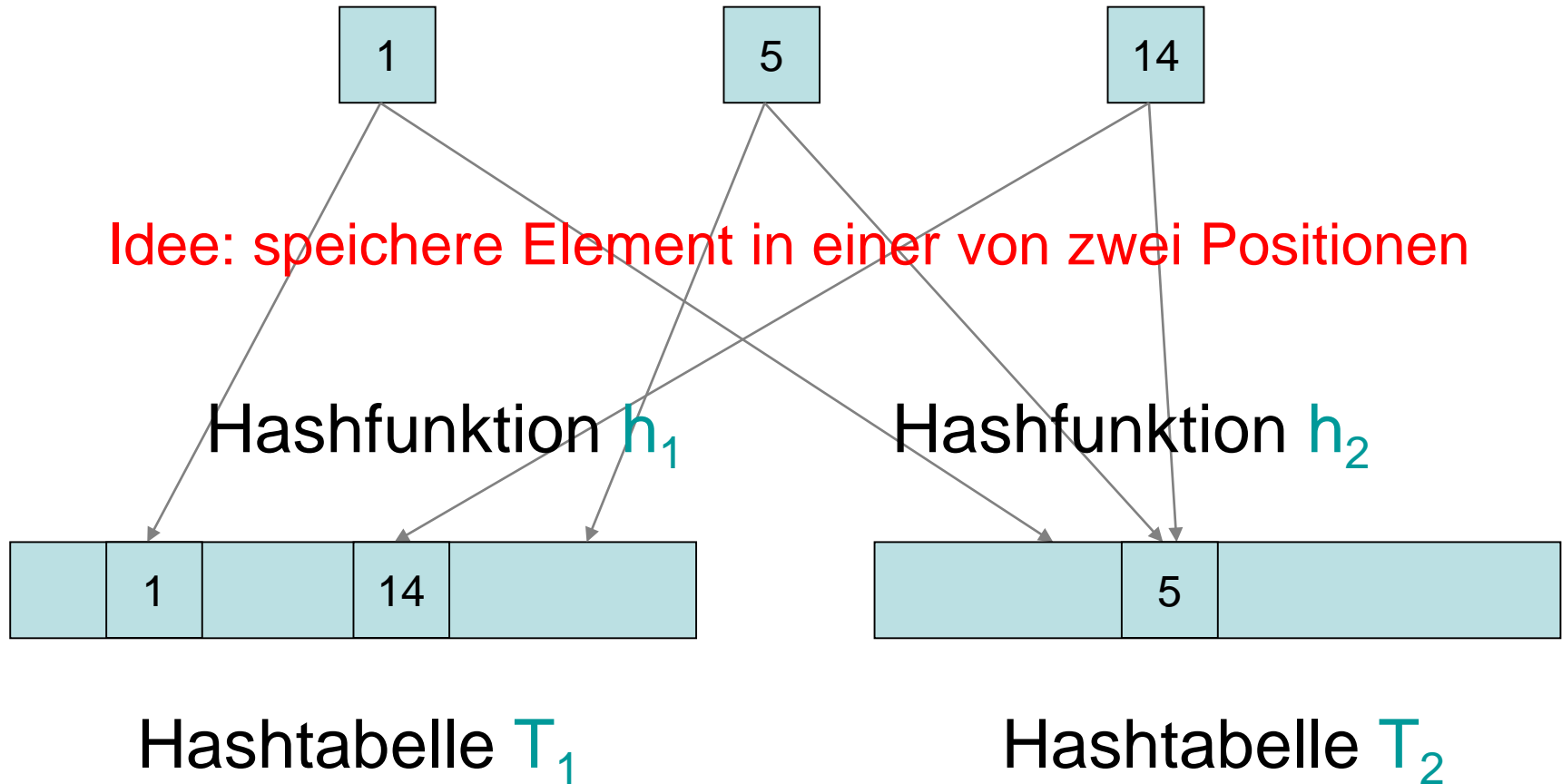
- **Geschlossene Adressierung**
    - Kollisionsauflösung durch Listen
  - **Offene Adressierung**
    - Lineares/Quadratisches Hashing: es wird nach der nächsten freien Stelle in T gesucht
  - **Kuckuckshashing**: geschickte Verwendung von zwei Hashfunktionen
-

# Hashing mit Listen - Illustration



# Kuckuckshashing

---



# Kuckuckshashing

---

$T_1, T_2$ : Hashtabellen der Größe  $m$

Search( $T_1, T_2, k$ ):

if  $\text{key}[T_1[h_1(k)]] = k$  then return  $T_1[h_1(k)]$

if  $\text{key}[T_2[h_2(k)]] = k$  then return  $T_2[h_2(k)]$

return NIL

Remove( $T_1, T_2, k$ ):

if  $\text{key}[T_1[h_1(k)]] = k$  then  $T_1[h_1(k)] \leftarrow \text{NIL}$

if  $\text{key}[T_2[h_2(k)]] = k$  then  $T_2[h_2(k)] \leftarrow \text{NIL}$

---

# Kuckuckshashing

---

```
Insert( $T_1, T_2, x$ ):  
  // key[x] schon in Hashtabelle?  
  if  $T_1[h_1(\text{key}[x])]=\text{NIL}$  or  $\text{key}[T_1[h_1(\text{key}[x])]]=\text{key}[x]$  then  
     $T_1[h_1(\text{key}[x])]\leftarrow x$ ; return  
  if  $T_2[h_2(\text{key}[x])]=\text{NIL}$  or  $\text{key}[T_2[h_2(\text{key}[x])]]=\text{key}[x]$  then  
     $T_2[h_2(\text{key}[x])]\leftarrow x$ ; return  
  // nein, dann einfügen  
  while true do  
     $x \leftrightarrow T_1[h_1(\text{key}[x])]$  // tausche x mit Pos. in  $T_1$   
    if  $x=\text{NIL}$  then return  
     $x \leftrightarrow T_2[h_2(\text{key}[x])]$  // tausche x mit Pos. in  $T_2$   
    if  $x=\text{NIL}$  then return
```

Oder maximal  $d \cdot \log n$  oft, wobei Konstante  $d$  so gewählt ist, dass die Wahrscheinlichkeit eines Erfolgs unter der einer Endlosschleife liegt. Bei Misserfolg kompletter Rehash mit neuen, zufälligen  $h_1, h_2$

---

# Priority Queue

---

**M**: Menge von Elementen

Jedes Element **e** identifiziert über **key(e)**.

Operationen:

- **insert(M,e)**:  $M := M \cup \{e\}$
  - **min(M)**: gib  $e \in M$  mit minimalem **key(e)** aus
  - **deleteMin(M)**: wie **min(M)**, aber zusätzlich  $M := M \setminus \{e\}$ , für **e** mit minimalem **key(e)**
-



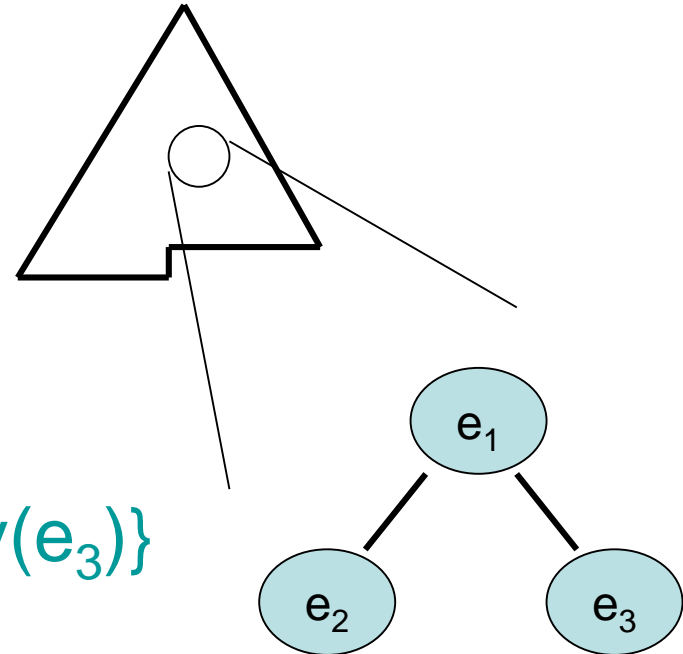
# Priority Queue als Heap

---

Idee: organisiere Daten im binären Baum

Bewahre zwei Invarianten:

- **Form-Invariante:** vollst. Binärbaum bis auf unterste Ebene



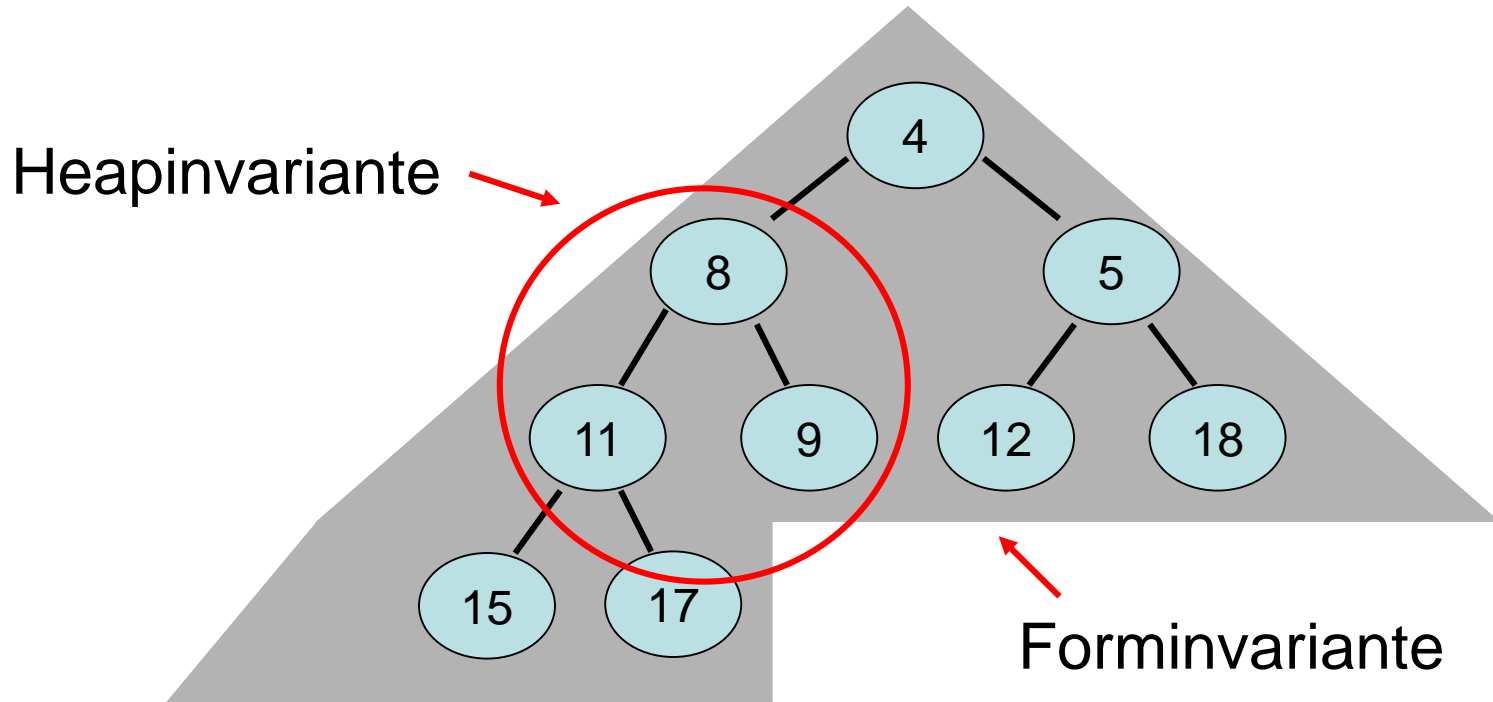
- **Heap-Invariante:**

$$\text{key}(e_1) \leq \min\{\text{key}(e_2), \text{key}(e_3)\}$$

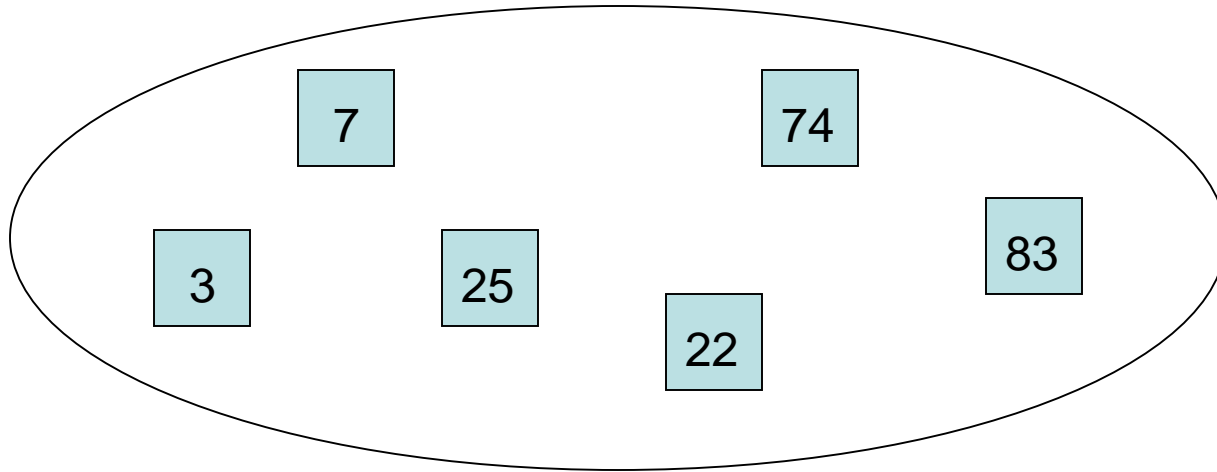
# Heap

---

Beispiel:



# Mengen

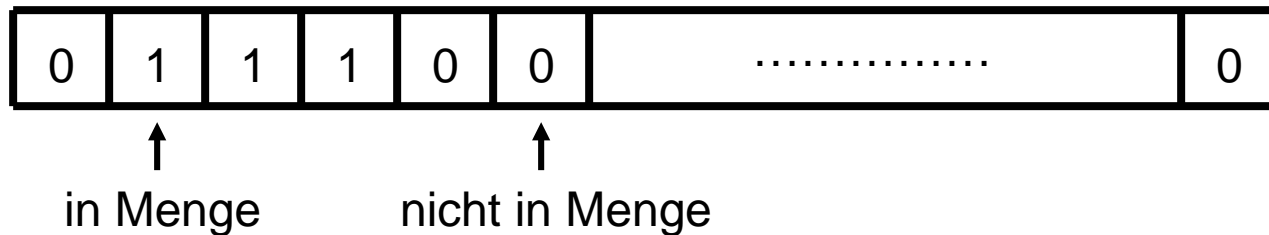


## Operationen:

- $\text{Member}(x,S)$ : testet, ob  $x$  ein Mitglied von  $S$  ist
- $\text{Union}(A,B)$ : gibt die Vereinigung von  $A$  und  $B$  zurück
- $\text{Intersection}(A,B)$ : gibt den Schnitt von  $A$  und  $B$  zurück
- $\text{Insert/Delete}(x,S)$ : füge  $x$  in  $S$  ein / lösche  $x$  aus  $S$

# Mengen

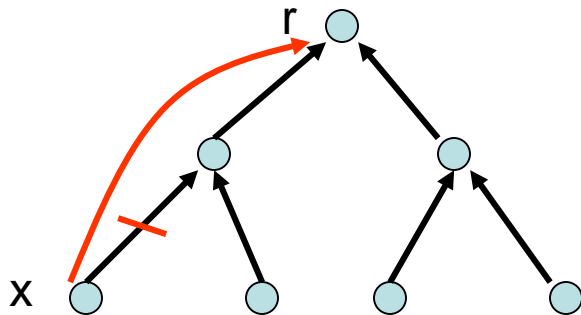
- Implementierung als Array: (vorzuziehen, falls Wertebereich klein)



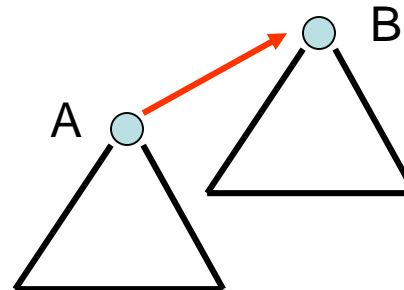
- Implementierung als Hashtabelle
- Implementierung als Suchbaum

# Mengen

- Operationen:
  - Union(A,B): gibt  $A \cup B$  zurück
  - Find(x): liefert Menge A von Element x
- Implementierung als Baum:  
Menge identifiziert durch Wurzel des Baums



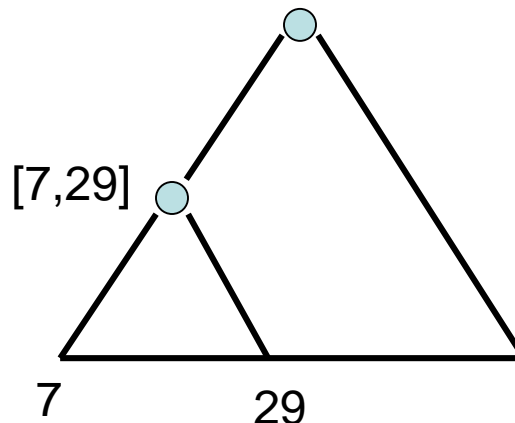
**Find(x):** biege Zeiger von x und Vorfahren auf r, gib r aus



**Union(A,B):** verbinde flacheren Baum (A) mit tieferem (B)

# Mengen

- Operationen:
  - Intersection(A,B): gibt  $A \cap B$  zurück
- Implementierung als Suchbaum:  
Füge Intervall  $[\min(T(v)), \max(T(v))]$  zu jedem Knoten  $v$  hinzu, wobei  $T(v)$  den Teilbaum mit Wurzel  $v$  angibt.  
Dann können über diese Intervalle effizient gemeinsame Elemente ermittelt werden.



# Probleme

- 10038: Jolly Jumpers
- 10591: Happy Number
- 484: The Department of Redundancy Department
- 496: Simply Subsets
- 10107: What is the Median?
- 1136: Help R2-D2!
- 10125: Sumsets

Hausaufgabe:

- 443: Humble Numbers