

---

# Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

**Universität zu Lübeck**

**Institut für Informationssysteme**

Stefan Werner (Übungen)

sowie viele Tutoren



# Danksagung

---

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus der Vorlesung „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 2: Priority Queues) gehalten von Christian Scheideler an der TUM

<http://www14.in.tum.de/lehre/2008WS/ea/index.html.de>

# Prioritätswarteschlangen

---

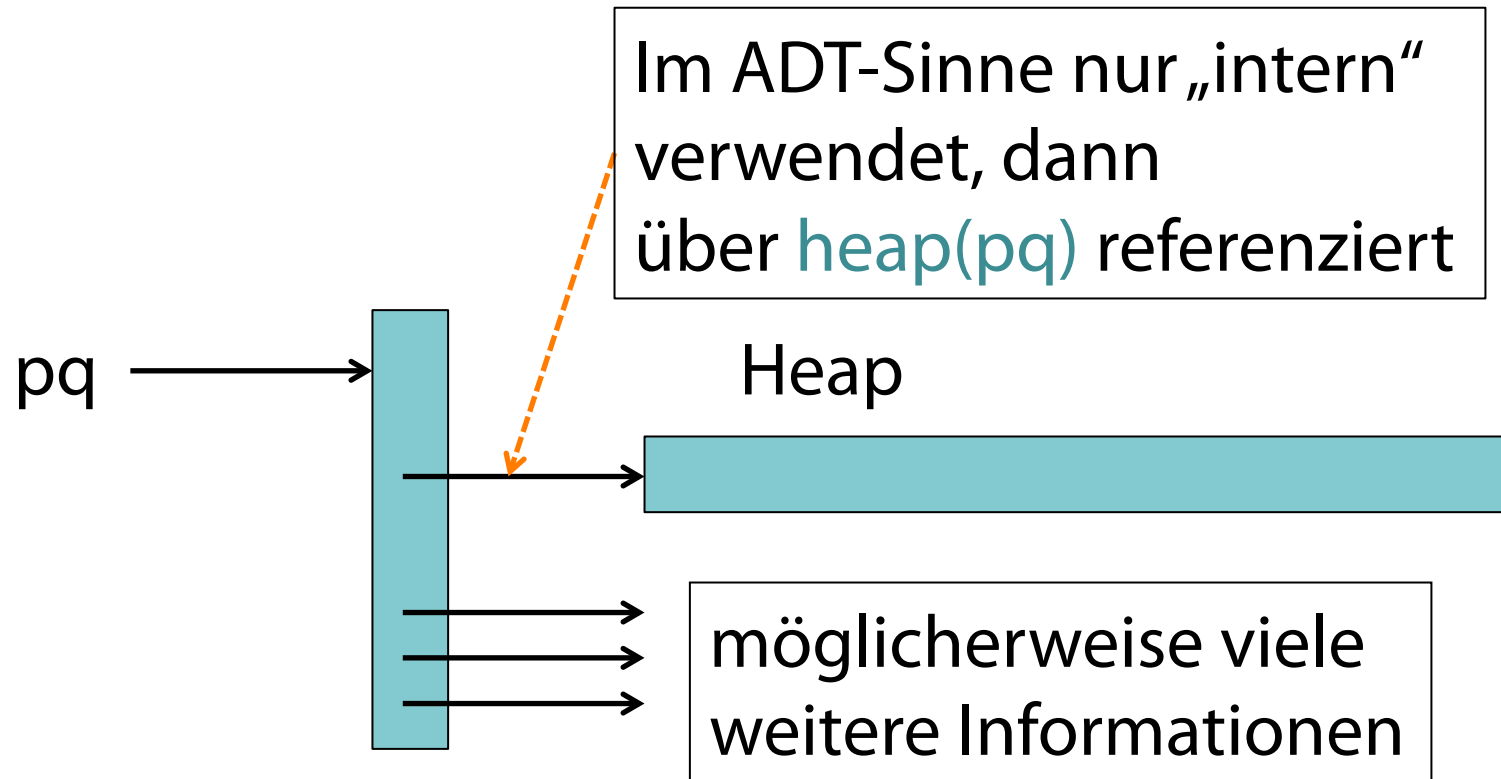
Geg.:  $\{e_1, \dots, e_n\}$  eine Menge von Elementen

Ann.: Priorität eines jeden Elements  $e$  wird identifiziert über die Funktion  $key$

Bekannte Operationen:

- function **build**( $\{e_1, \dots, e_n\}$ ) liefert neue Warteschlange
- procedure **insert**( $e, pq$ ) fügt Element  $e$  mit Priorität  $key(e)$  in  $pq$  ein, verändert  $pq$  sofern  $e$  noch nicht in  $pq$
- function **min**( $pq$ ) gibt Element mit minimalem  $key(e)$  zurück
- procedure **deleteMin**( $pq$ ): löscht das minimale Element in  $pq$ , sofern vorhanden, und  $pq$  wird verändert, wenn etwas gelöscht wird

# Prioritätswarteschlangen als ADTs



# Erweiterte Prioritätswarteschlangen

---

## Zusätzliche Operationen:

- **procedure delete**( $e, pq$ ) löscht  $e$  aus  $pq$ , falls vorhanden, verändert ggf.  $pq$
- **procedure decreaseKey**( $e, pq, \Delta$ ):  $key(e) := key(e) - \Delta$ , verändert evtl.  $pq$
- **procedure merge**( $pq, pq'$ ) fügt  $pq$  und  $pq'$  zusammen, verändert ggf.  $pq$  und auch  $pq'$

# Prioritätswarteschlangen

---

- Einfache Realisierung mittels unsortierter Liste:
  - build: Zeit  $O(n)$
  - insert:  $O(1)$
  - min, deleteMin:  $O(n)$
- Realisierung mittels sortiertem Feld:
  - build: Zeit  $O(n \log n)$  (sortieren)
  - insert:  $O(n)$  (verschiebe Elemente in Feld)
  - min, deleteMin:  $O(1)$  (mit Anfangszeiger)

**Bessere Struktur als Liste oder Feld möglich!**

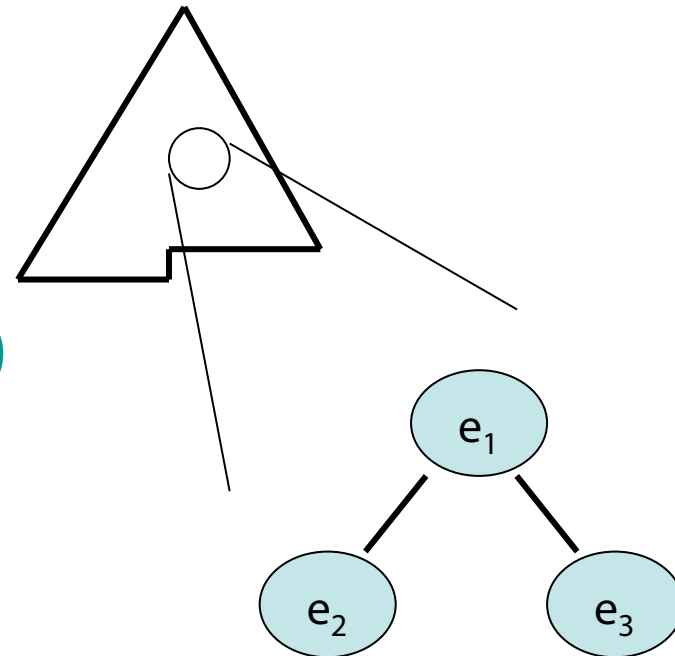
# Binärer Heap (Wiederholung)

Idee: verwende binären Baum statt Liste

Bewahre zwei Invarianten:

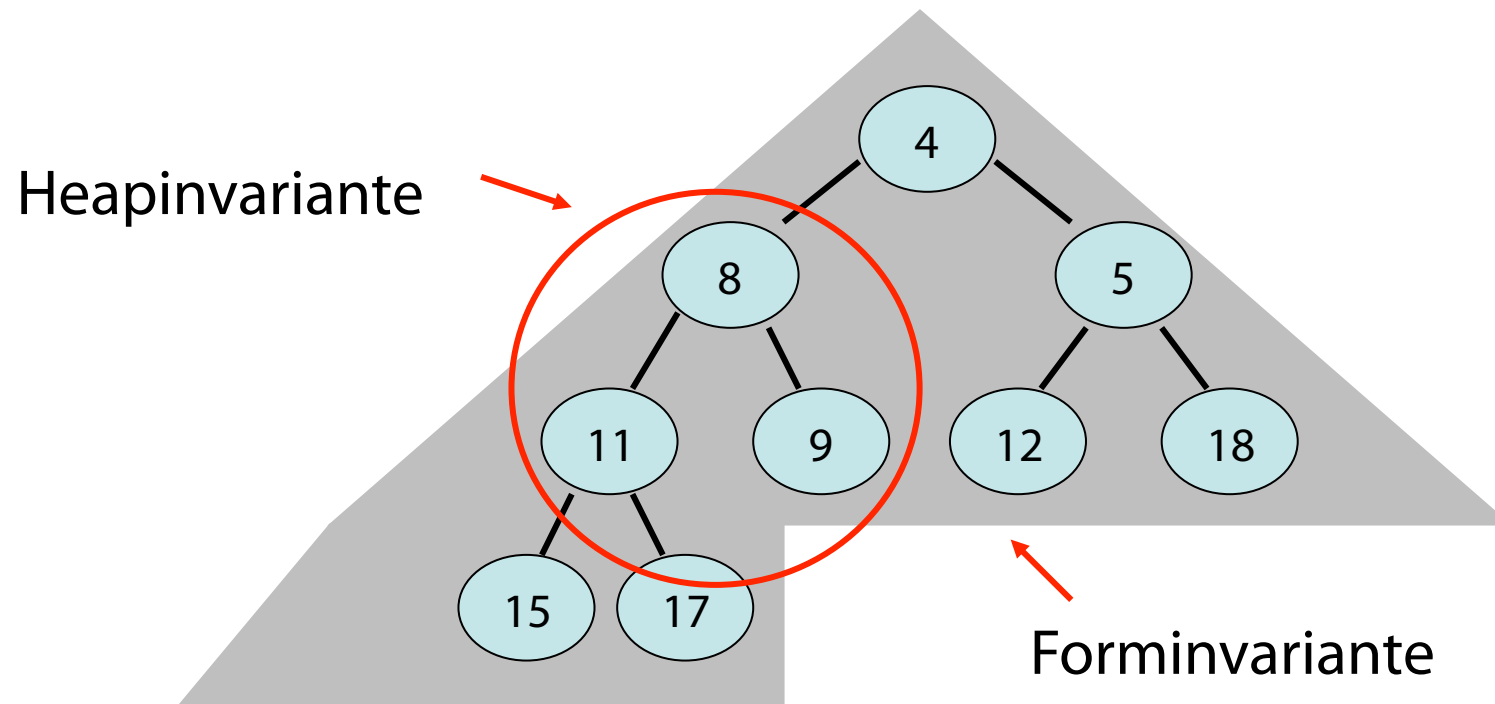
- **Form-Invariante:**  
vollst. Binärbaum bis auf unterste Ebene
- **(Min)Heap-Invariante:**

$\text{key}(e_1) \leq \min( \{ \text{key}(e_2), \text{key}(e_3) \} )$   
für die Kinder  $e_2$  und  $e_3$  von  $e_1$



# Binärer Heap (Wiederholung)

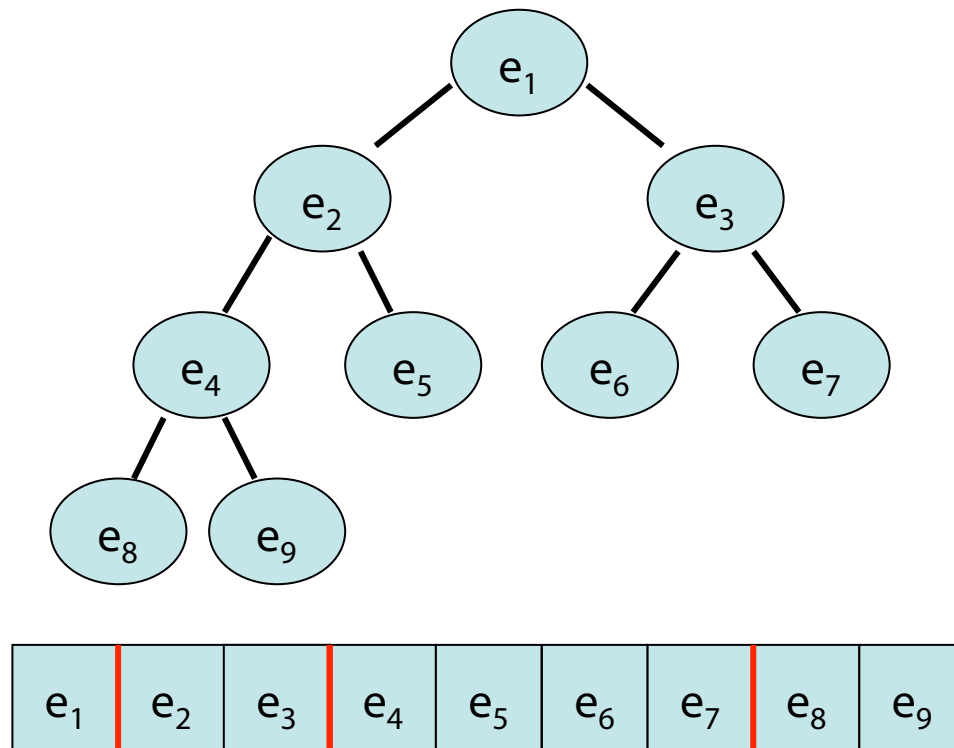
Beispiel:





# Binärer Heap (Wiederholung)

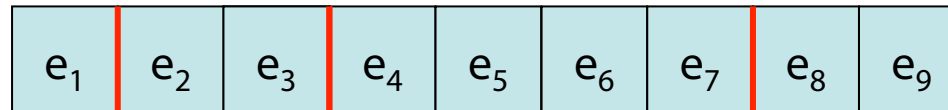
Realisierung eines Binärbaums als Feld:



# Binärer Heap (Wiederholung)

---

Realisierung eines Binärbaums als Feld:

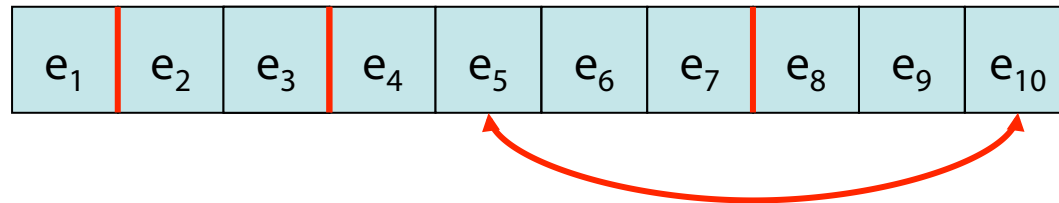


- **H**: Array [1..n]
- Kinder von **e** in  $H[i]$ : in  $H[2i], H[2i+1]$
- **Form-Invariante**:  $H[1], \dots, H[k]$  besetzt für  $k \leq n$
- **Heap-Invariante**:  
$$\text{key}(H[i]) \leq \min( \{ \text{key}(H[2i]), \text{key}(H[2i+1]) \} )$$

# Binärer Heap (Wiederholung)

---

Realisierung eines Binärbaums als Feld:



insert(e, pq): Sei H das Trägerfeld von pq ( $H = \text{heap}(pq)$ )

- **Form-Invariante:**  $n := n + 1$ ;  $H[n] := e$
- **Heap-Invariante:** vertausche e mit Vater bis  $\text{key}(H[\lfloor k/2 \rfloor]) \leq \text{key}(e)$  für e in  $H[k]$  oder e in  $H[1]$

# Insert Operation

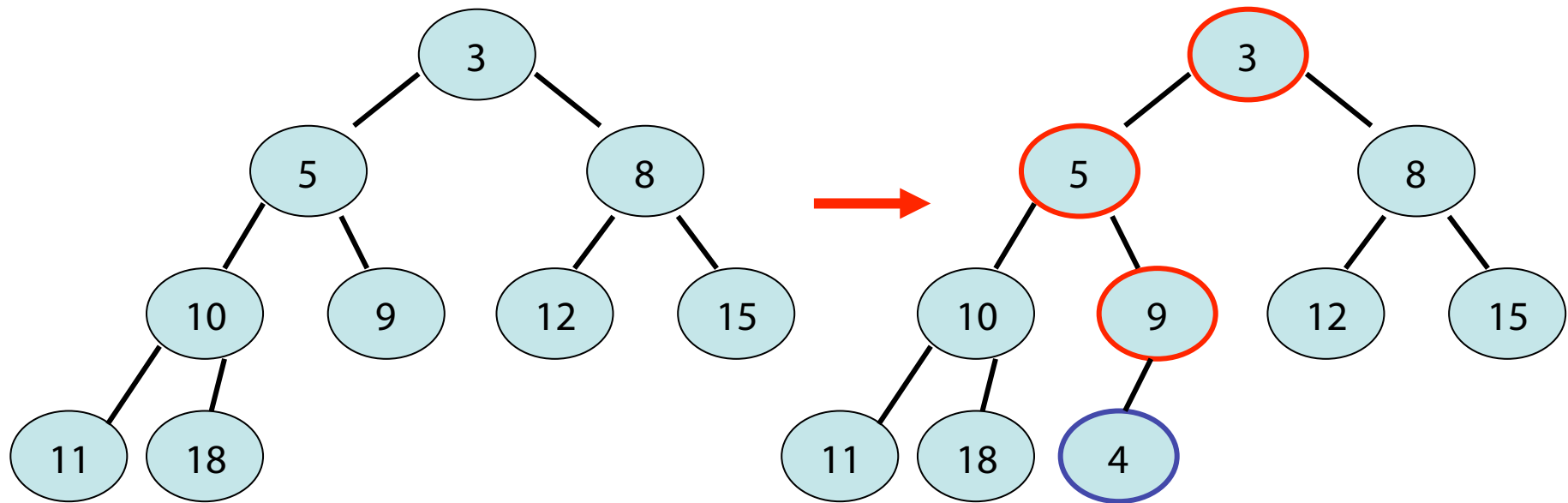
---

```
Procedure insert(e, pq)  
  H:=heap(pq); n:=n+1; H[n]:=e  
  siftUp(n, H)
```

```
Procedure siftUp(i, H)  
  while i>1 and key( H[⌊i/2⌋] ) > key( H[i] ) do  
    temp := H[i]; H[i] := H[⌊i/2⌋]; H[⌊i/2⌋] := temp;  
    i:=⌊i/2⌋
```

Laufzeit:  $O(\log n)$

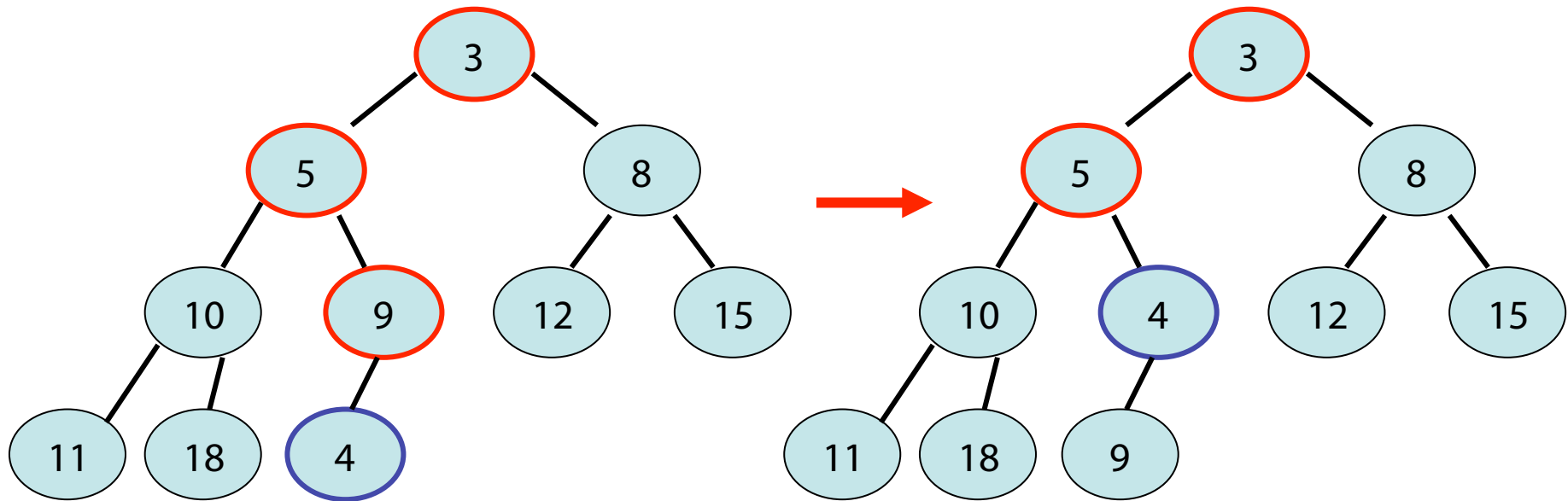
# Insert Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

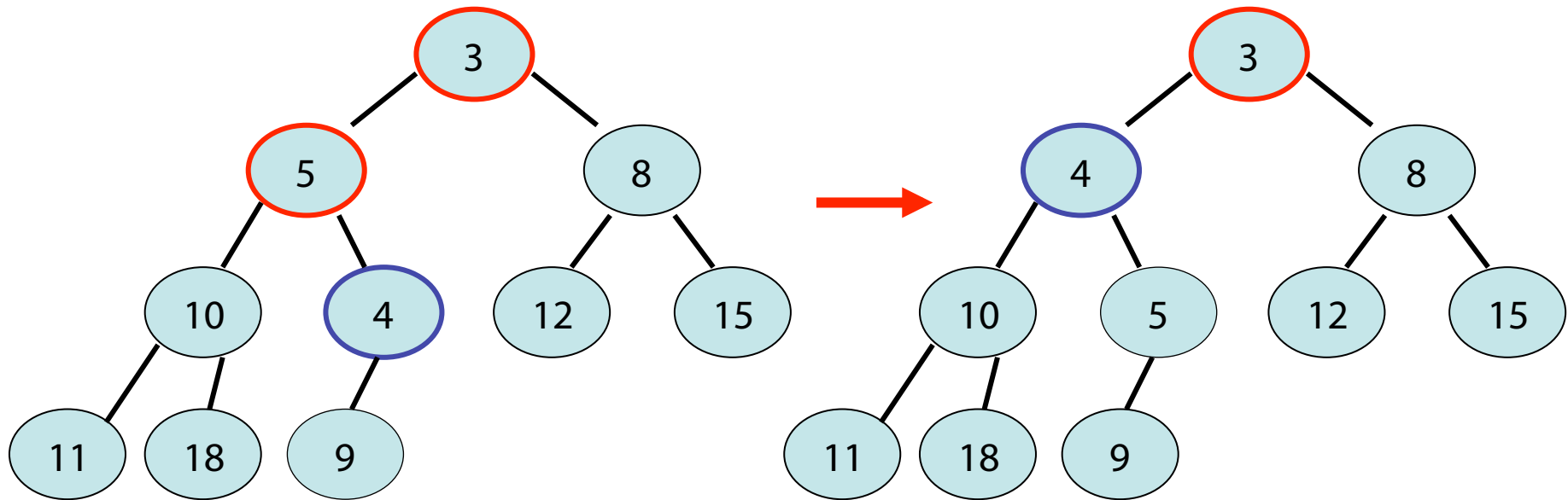
# Insert Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

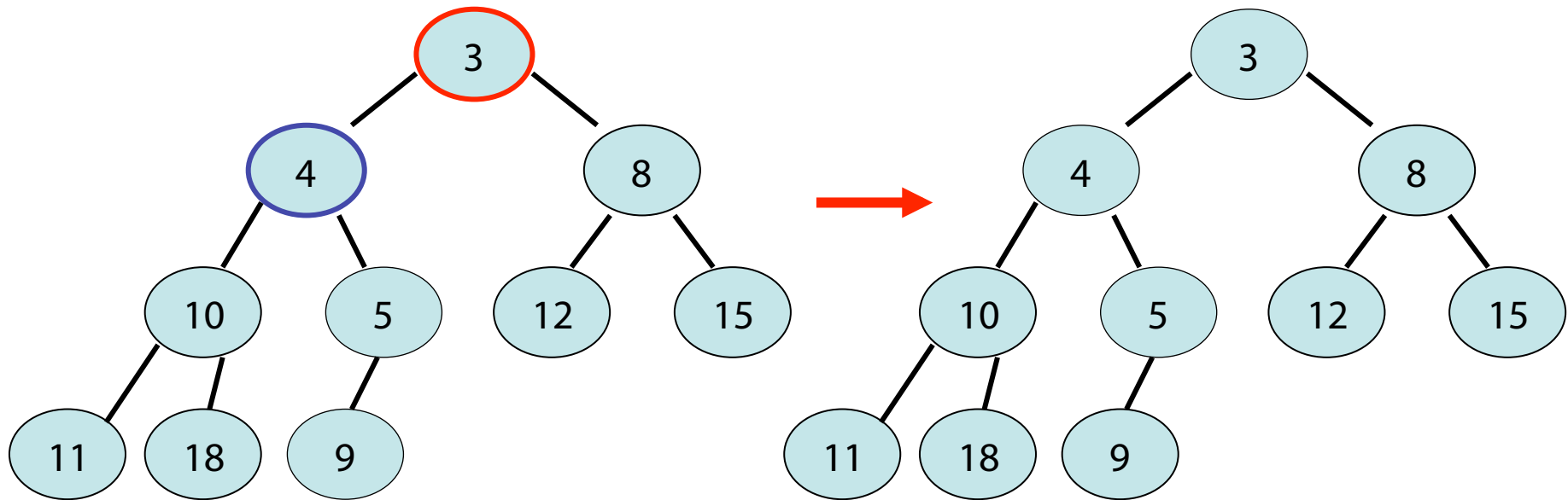
# Insert Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

# Insert Operation - Korrektheit



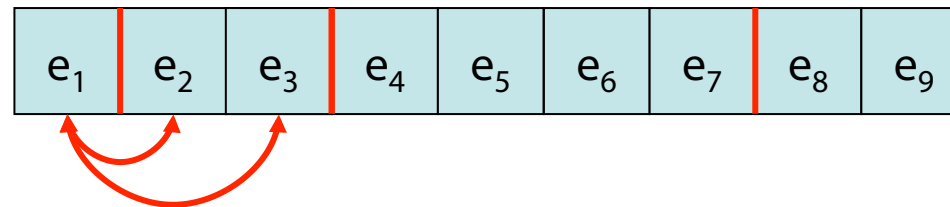
Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen



# Binärer Heap

---



deleteMin(pq):

- **Form-Invariante:**  $H[1] := H[n]; n := n - 1$
- **Heap-Invariante:** starte mit Element  $e$  in  $H[1]$ .  
Vertausche  $e$  mit Kind mit min Schlüssel bis  
 $H[k] \leq \min(\{ H[2k], H[2k+1] \})$  für Position  $k$  von  $e$   
oder  $e$  in Blatt

# Binärer Heap

---

function `deleteMin`(pq):

`H:=heap(pq); e:=H[1]; H[1]:=H[n]; n:=n-1`

`siftDown(1, H)`

return `e`

Laufzeit:  $O(\log n)$

procedure `siftDown`(`i, H`)

while `2i ≤ n` do

if `2i+1 > n` then `m:=2i` // `m: Pos. des min. Kindes`

else

if `key(H[2i]) < key(H[2i+1])`

then `m:=2i`

else `m:=2i+1`

if `key(H[i]) ≤ key(H[m])`

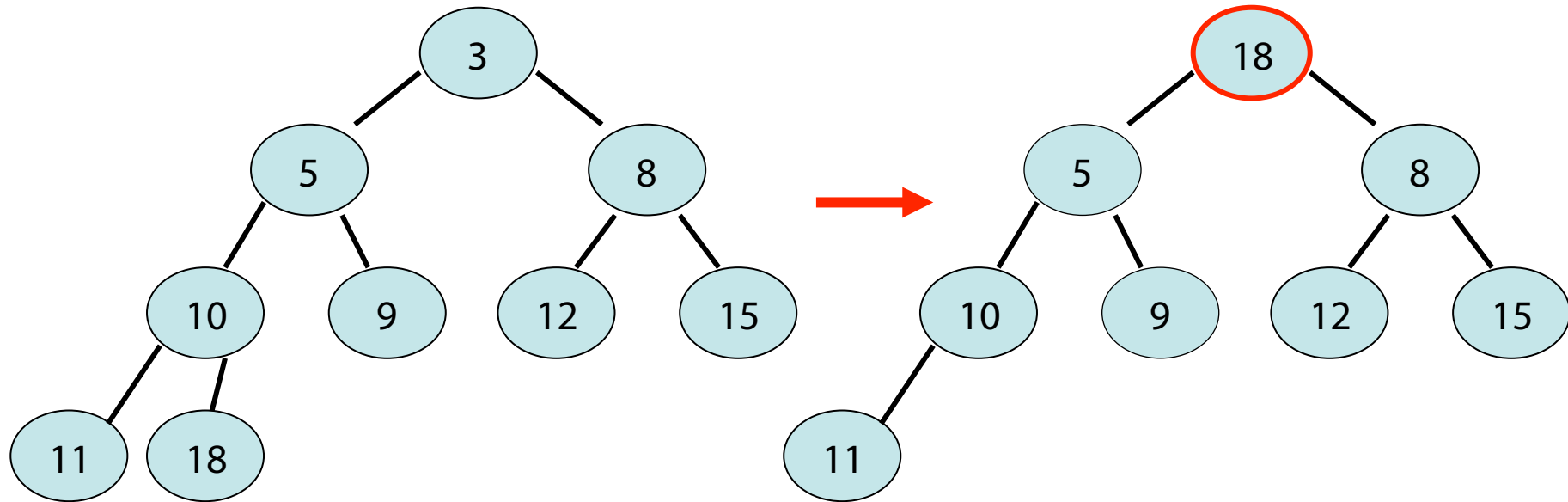
then exit // Heap-Inv gilt

`temp := H[i]; H[i] := H[m]; H[m] := temp;`

`i:=m`



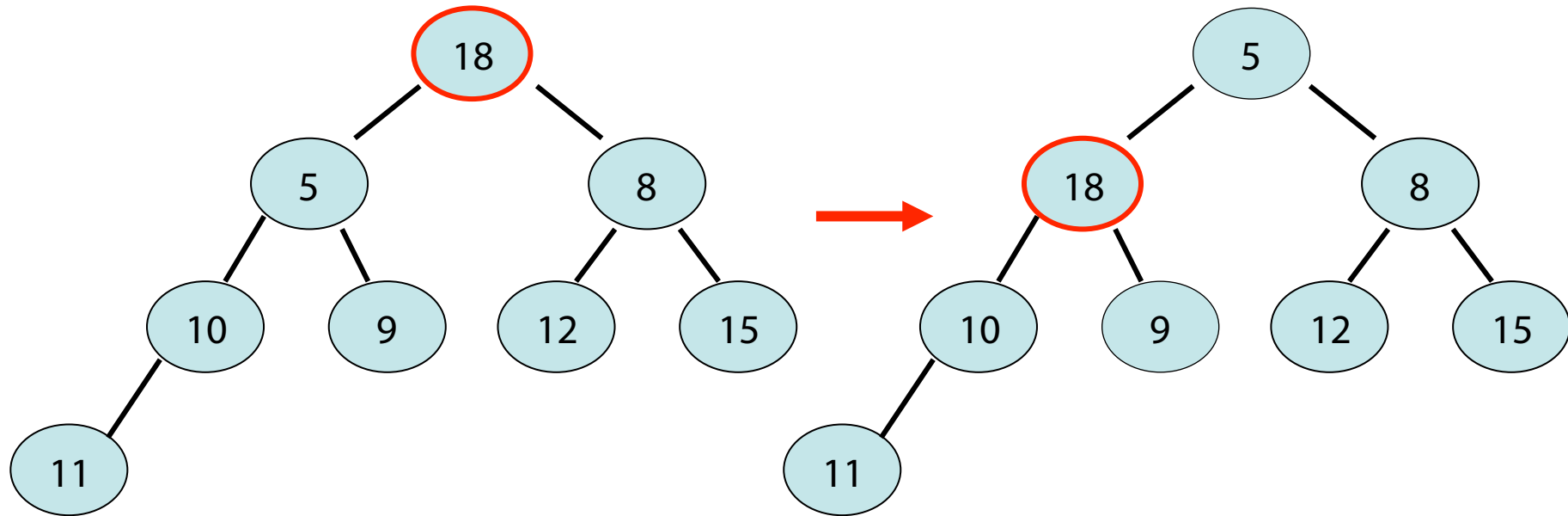
# deleteMin Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

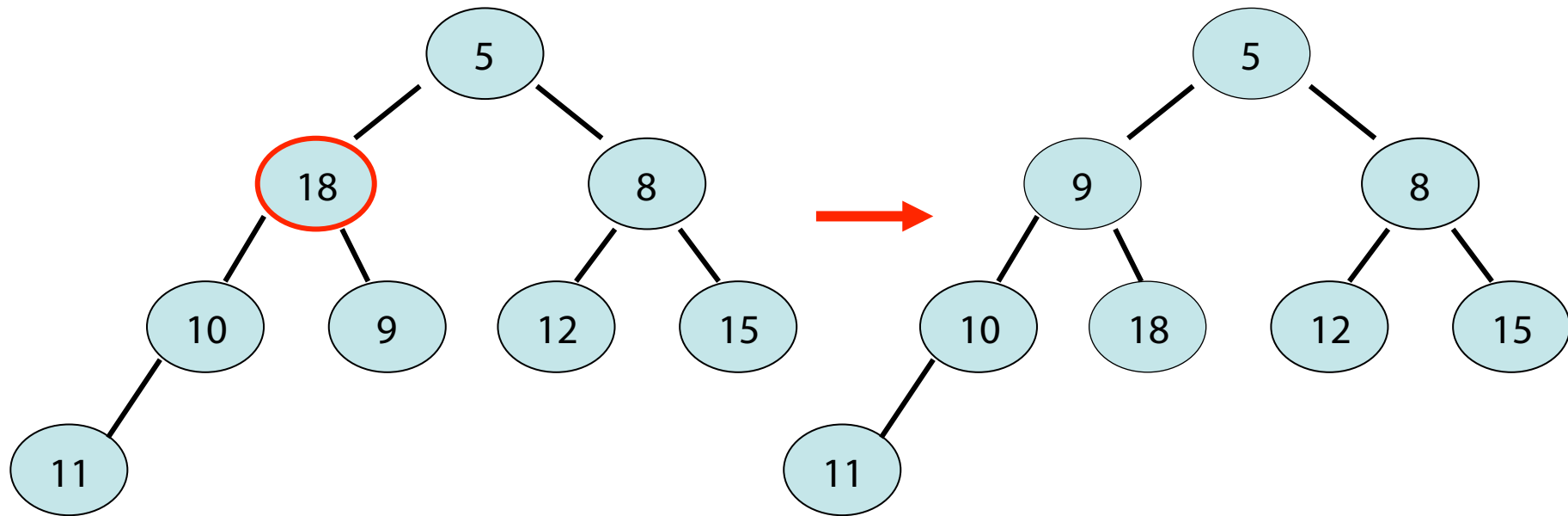
# deleteMin Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

# deleteMin Operation - Korrektheit



Invariante:  $H[k]$  ist minimal für Teilbaum von  $H[k]$

 : Knoten, die Invariante eventuell verletzen

# Binärer Heap

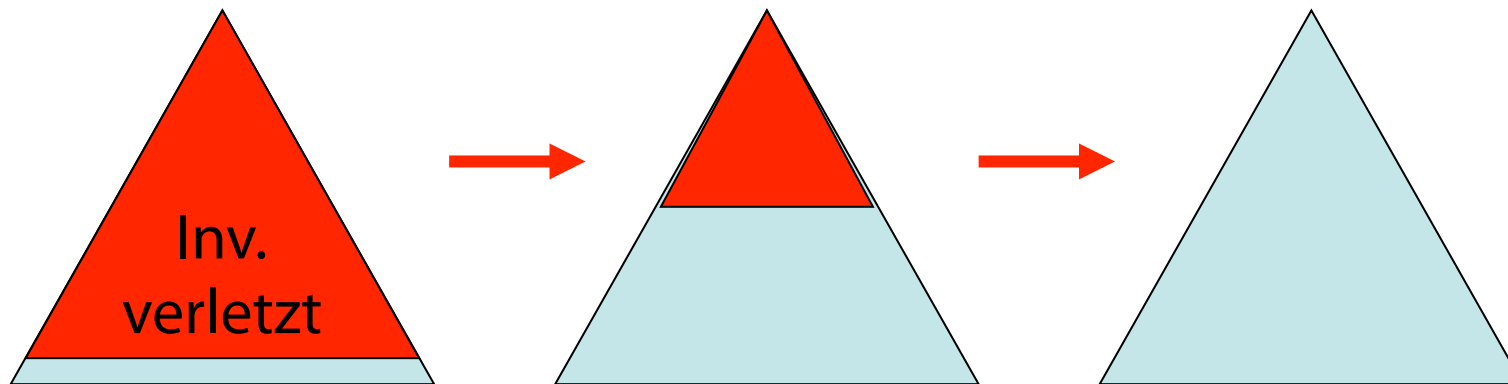
---

$\text{build}(\{e_1, \dots, e_n\})$ :

- Naive Implementierung: über  $n$   $\text{insert}(e)$ -Operationen.  
Laufzeit  $O(n \log n)$
- Bessere Implementierung:  
Setze  $H[i] := e_i$  für alle  $i$ . Rufe  $\text{siftDown}(i)$  auf für  $i = \lfloor n/2 \rfloor$  runter bis  $1$  (d.h. von der vorletzten Ebene hoch bis zur obersten Ebene)

# Binärer Heap: Operation build

Setze  $H[i]:=e_i$  für alle  $i$ . Rufe  $\text{siftDown}(i, H)$  für  $i=\lfloor n/2 \rfloor$  runter bis 1 auf.



Invariante: Für alle  $j>i$ :  $H[j]$  minimal für Teilbaum von  $H[j]$

Aufwand? Sicher  $O(n \log n)$ , siehe vorige Überlegungen

Unnötig pessimistisch (besser gesagt: asymptotisch nicht eng)

# Aufwand für build

- Die Höhe des Baumes, in den eingesiebt wird, nimmt zwar von unten nach oben zu, ...
- ... aber für die meisten Knoten ist die Höhe „klein“ (die meisten Knoten sind unten)
- Ein n-elementiger Heap hat Höhe  $\log n$  ...
- ... und maximal  $\lceil n/2^{h+1} \rceil$  viele Knoten (Teilbäume) mit Höhe h
- siftDown, aufgerufen auf Ebene h, braucht h Schritte
- Der Aufwand für build ist also

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

- wobei wir  $x = 1/2$  setzen in  $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$   
for  $|x| < 1$ .

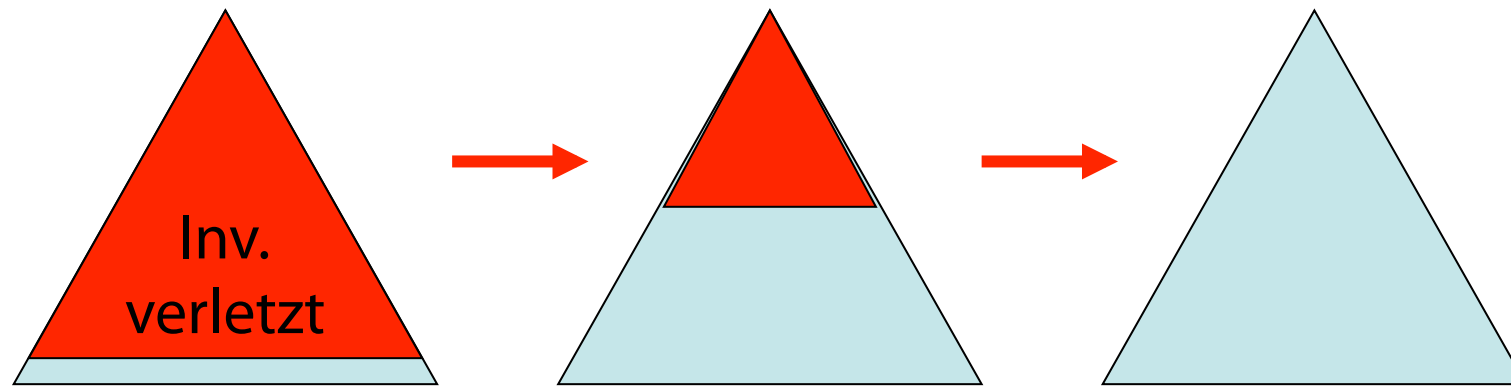
Ergibt sich aus der Ableitung der geometrischen Reihe mit  $a_0 = 1$ :

$$\sum_{k=0}^{\infty} a_0 q^k = \frac{a_0}{1-q}$$



# Binärer Heap: Operation build

Setze  $H[i] := e_i$  für alle  $i$ . Rufe  $\text{siftDown}(i, H)$  für  $i = \lfloor n/2 \rfloor$  runter bis 1 auf.



Invariante: Für alle  $j > i$ :  $H[j]$  minimal für Teilbaum von  $H[j]$

Aufwand ist gekennzeichnet durch eine Funktion in  $O(n)$

# Binärer Heap

---

## Laufzeiten:

- build:  $O(n)$
- insert:  $O(\log n)$
- min:  $O(1)$
- deleteMin:  $O(\log n)$

# Erweiterte Prioritätswarteschlange

---

Zusätzliche Operationen:

- `delete(e, pq)`
- `decreaseKey(e, pq,  $\Delta$ )`
- `merge(pq, pq')`

Delete und `decreaseKey` in Zeit  $O(\log n)$  in Heap (wenn Position von `e` bekannt), aber `merge` ist **teuer** ( $\Theta(n)$  Zeit)!

# Binomial-Heap zum schnellen Verschmelzen

Binomial-Heap basiert auf sog. Binomial-Bäumen

Binomial-Baum muss erfüllen:

- **Form-Invariante** ( $r$ : Rang):

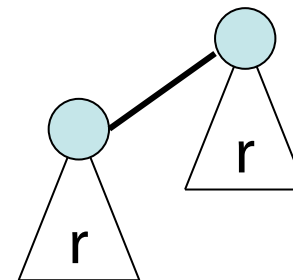
$r=0$



$r=1$



$r \rightarrow r+1$



- **Heap-Invariante:**  $\text{key}(\text{Vater}) \leq \text{key}(\text{Kinder})$

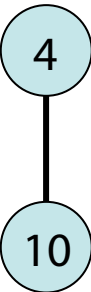
# Binomial-Heap

Beispiel für korrekte Binomial-Bäume:

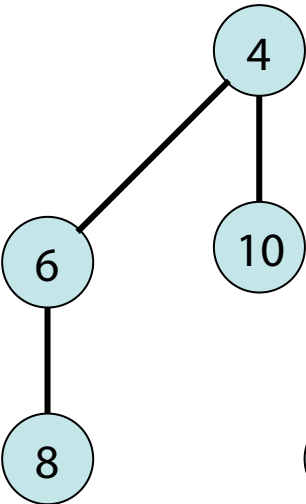
r=0



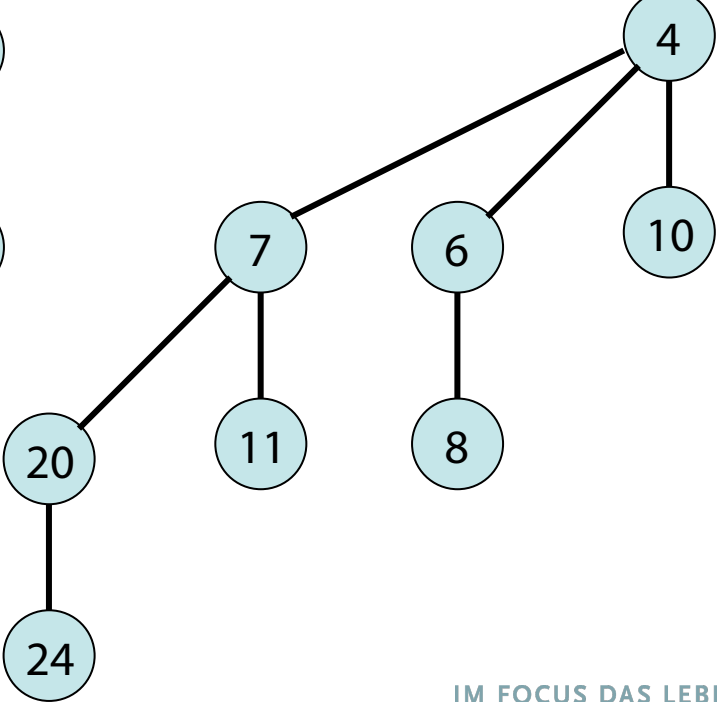
r=1



r=2



r=3



# Binomial-Heap

Eigenschaften von Binomial-Bäumen:

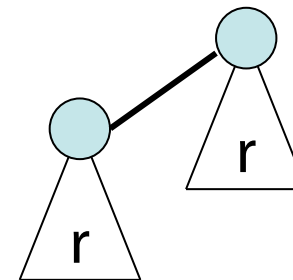
$r=0$



$r=1$



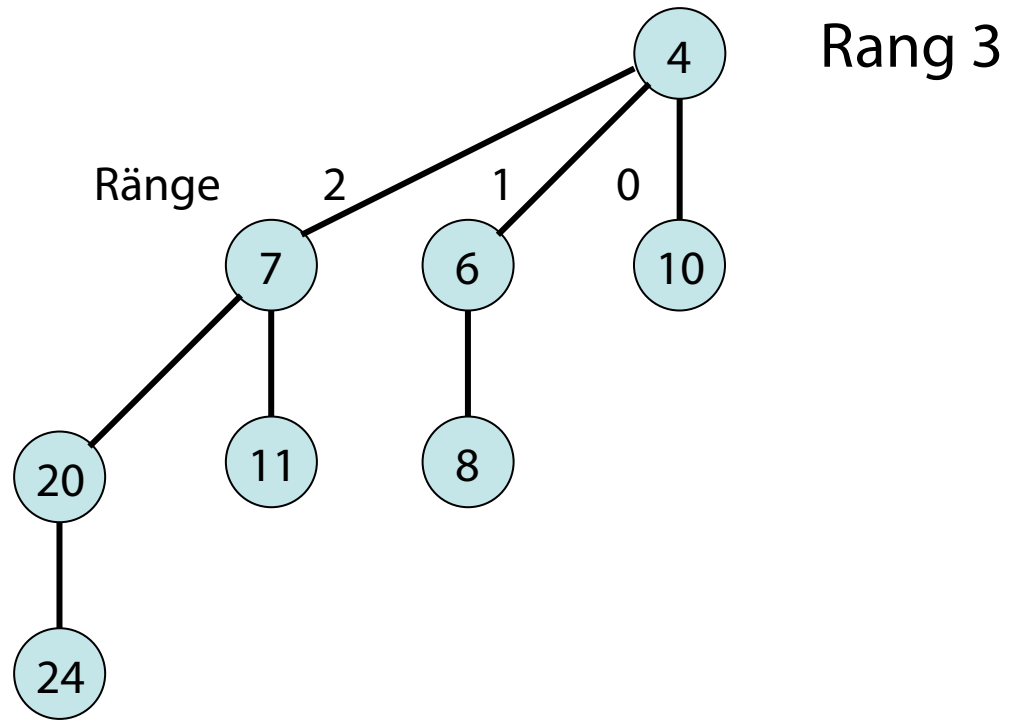
$r \rightarrow r+1$



- $2^r$  Knoten
- maximaler Grad  $r$  (bei Wurzel)
- Wurzel weg: **Zerfall** in Binomial-Bäume mit Rang  $0$  bis  $r-1$

# Binomial-Heap

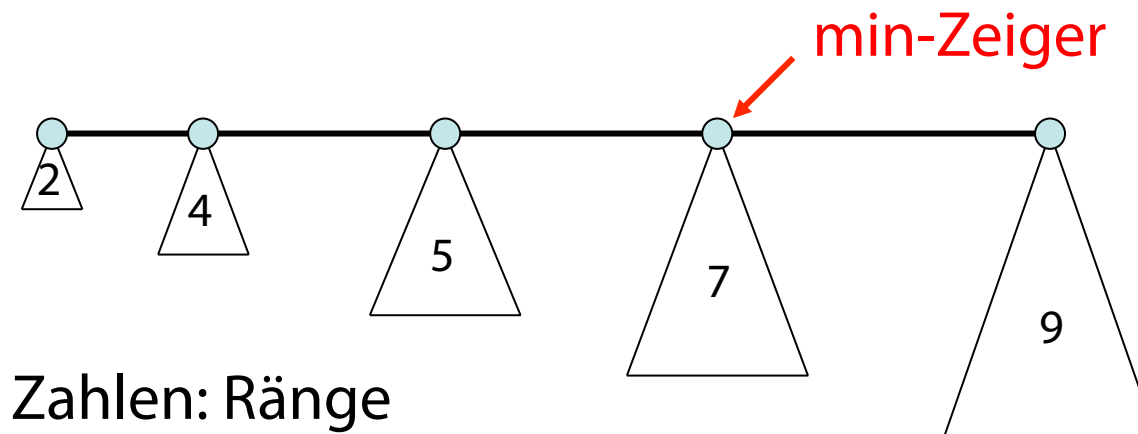
Beispiel für Zerfall in Binomial-Bäume mit Rang 0 bis  $r-1$



# Binomial-Heap

## Binomial-Heap:

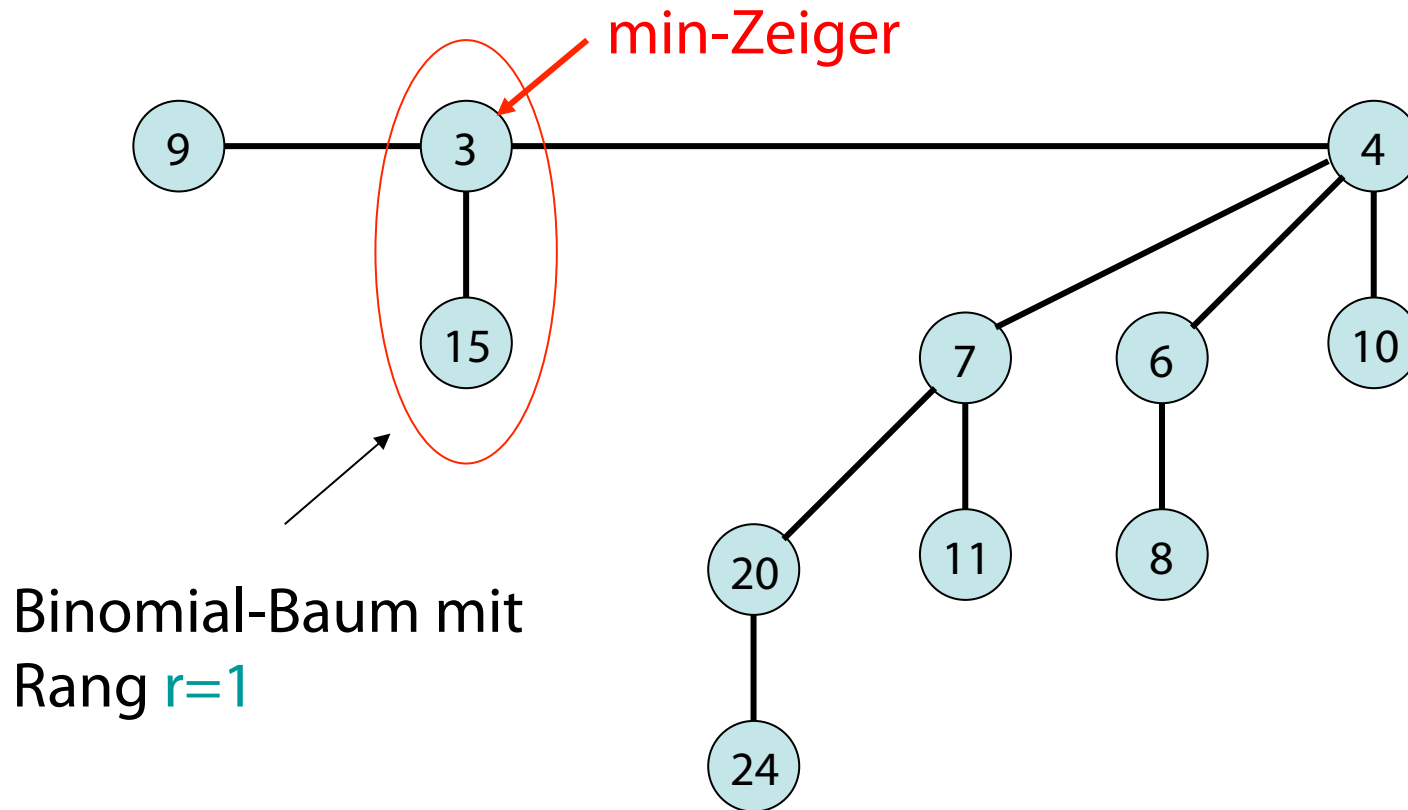
- verkettete Liste von Binomial-Bäumen
- Pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem key





# Binomial-Heap

Beispiel eines korrekten Binomial-Heaps:



Binomial-Baum mit  
Rang  $r=1$

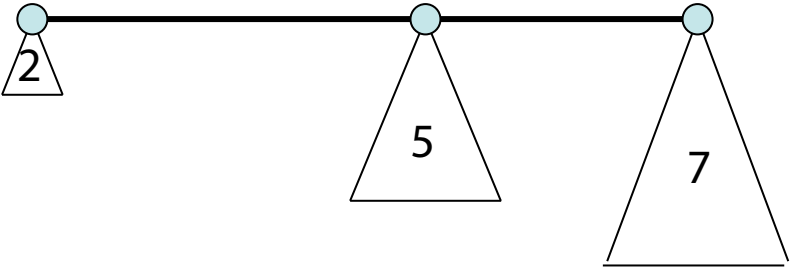
# Anzahl der Bäume auf der Kette

---

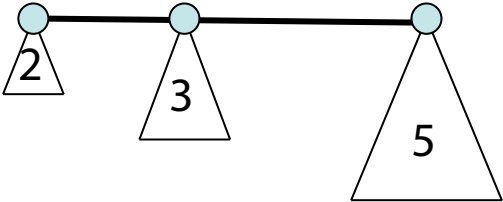
- Binomial-Heap-Invariante:  
Pro Rang maximal 1 Binomial-Baum
- Was heißt das?
- Für  $n$  Knoten können höchstens  $\log n$  viele Binomialbäume in der Kette vorkommen  
(dann müssen alle Knoten untergebracht sein)

# Binomial-Heap

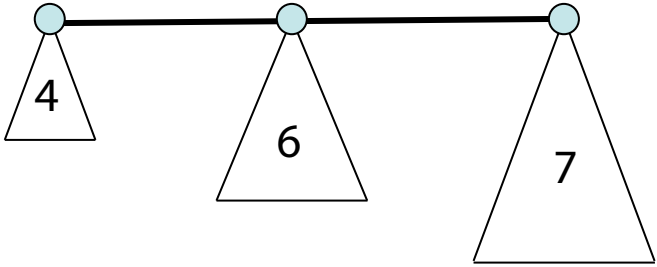
Merge von Binomial-Heaps  $H_1$  und  $H_2$ :



$H_1$



$H_2$



wie Binäraddition

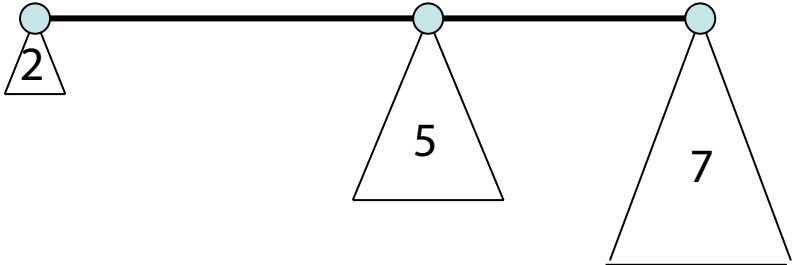
10100100

+ 101100

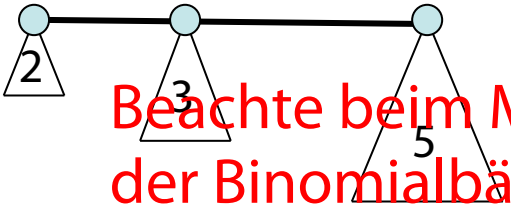
---

11010000

# Beispiel einer Merge-Operation

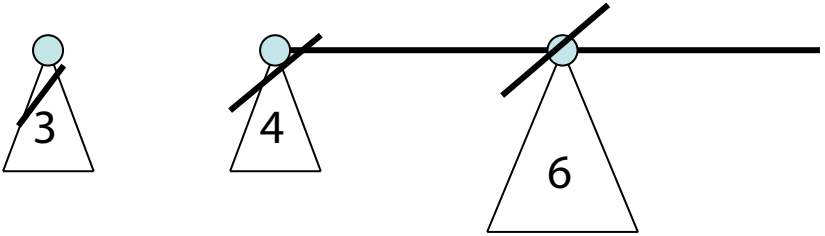


H<sub>1</sub>



Beachte beim Mergen  
der Binomialbäume die  
Heap-Eigenschaft!

H<sub>2</sub>



Zahlen geben  
die Ränge an

Ergebnis-Heap

# Operationen auf Binomial-Heaps

---

Sei  $B_i$ : Binomial-Baum mit Rang  $i$

- $\text{merge}(pq, pq')$ : Aufwand für Merge-Operation:  $O(\log n)$
- $\text{insert}(e, pq)$ : Merge mit  $B_0$ , Zeit  $O(\log n)$
- $\text{min}$ : spezieller Zeiger, Zeit  $O(1)$
- $\text{deleteMin}$ : sei Minimum in Wurzel von  $B_i$ , Löschen von Minimum:  $B_i \rightarrow B_0, \dots, B_{i-1}$ . Diese zurückmergen in Binomial-Heap. Zeit dafür  $O(\log n)$ .

# Binomial-Heap

---

- $\text{decreaseKey}(e, pq, \Delta)$ :  $\text{siftUp}$ -Operation in Binomial-Baum von  $e$  und aktualisierte min-Zeiger. Zeit  $O(\log n)$
- $\text{delete}(e, pq)$ : (min-Zeiger zeigt nicht auf  $e$ ) setze  $\text{key}(e) := -\infty$  und wende  $\text{siftUp}$ -Operation auf  $e$  an, bis  $e$  in der Wurzel; dann weiter wie bei  $\text{deleteMin}$ .  
Zeit  $O(\log n)$

# Anwendungen für log-n-merge

---

- Lastumverteilung
  - Delegation der Aufträge für einen Prozessor an einen anderen (evtl. schnelleren) Prozessor
- Reduce-Operation, Mischung von parallel ermittelten Ergebnissen, jeweils mit Bewertung bzw. Sortierung

# Zusammenfassung

---

Laufzeit	Binärer-Heap	Binomial-Heap
insert	$O(\log n)$	$O(\log n)$
min	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$
delete	$O(\log n)$	$O(\log n)$
decreaseKey	$O(\log n)$	$O(\log n)$
merge	$O(n)$	$O(\log n)$



# Datenstruktur Fibonacci-Heap

---

- Baut auf Binomial-Bäumen auf, aber erlaubt **lazy merge** und **lazy delete**.
- **Lazy merge**: keine Verschmelzung von Binomial-Bäumen gleichen Ranges bei merge, nur Verkettung der Wurzellisten
- **Lazy delete**: erzeugt unvollständige Binomial-Bäume

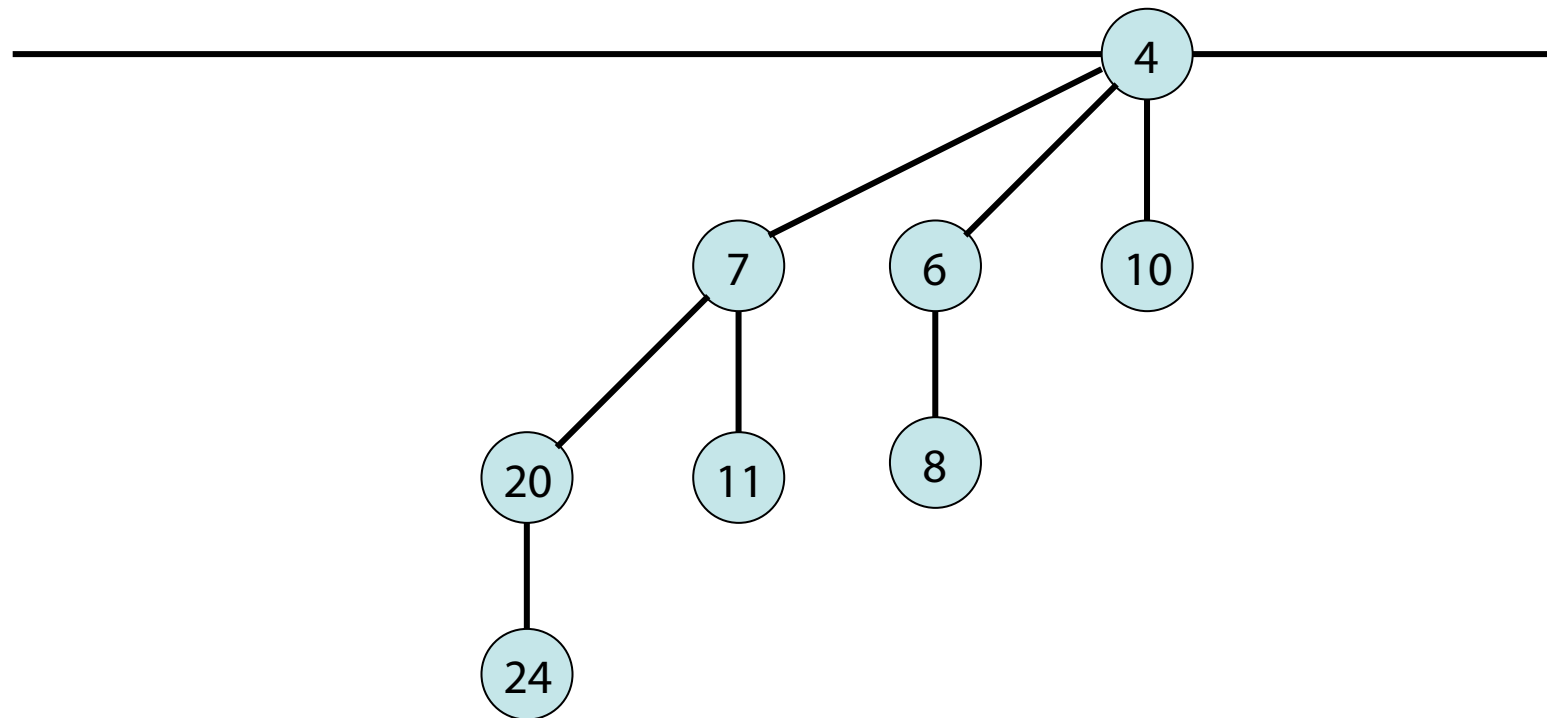
Der Name rührt von der Analyse der Datenstruktur her, bei der Fibonacci-Zahlen eine große Rolle spielen (wird nachher deutlich)

Michael L. Fredman, Robert E. Tarjan: Fibonacci heaps and their uses in improved network optimization algorithms. In: Journal of the ACM. 34, Nr. 3, S. 596–615, 1987

# Fibonacci-Heap

Baum in Binomial-Heap: Zentrale Invariante

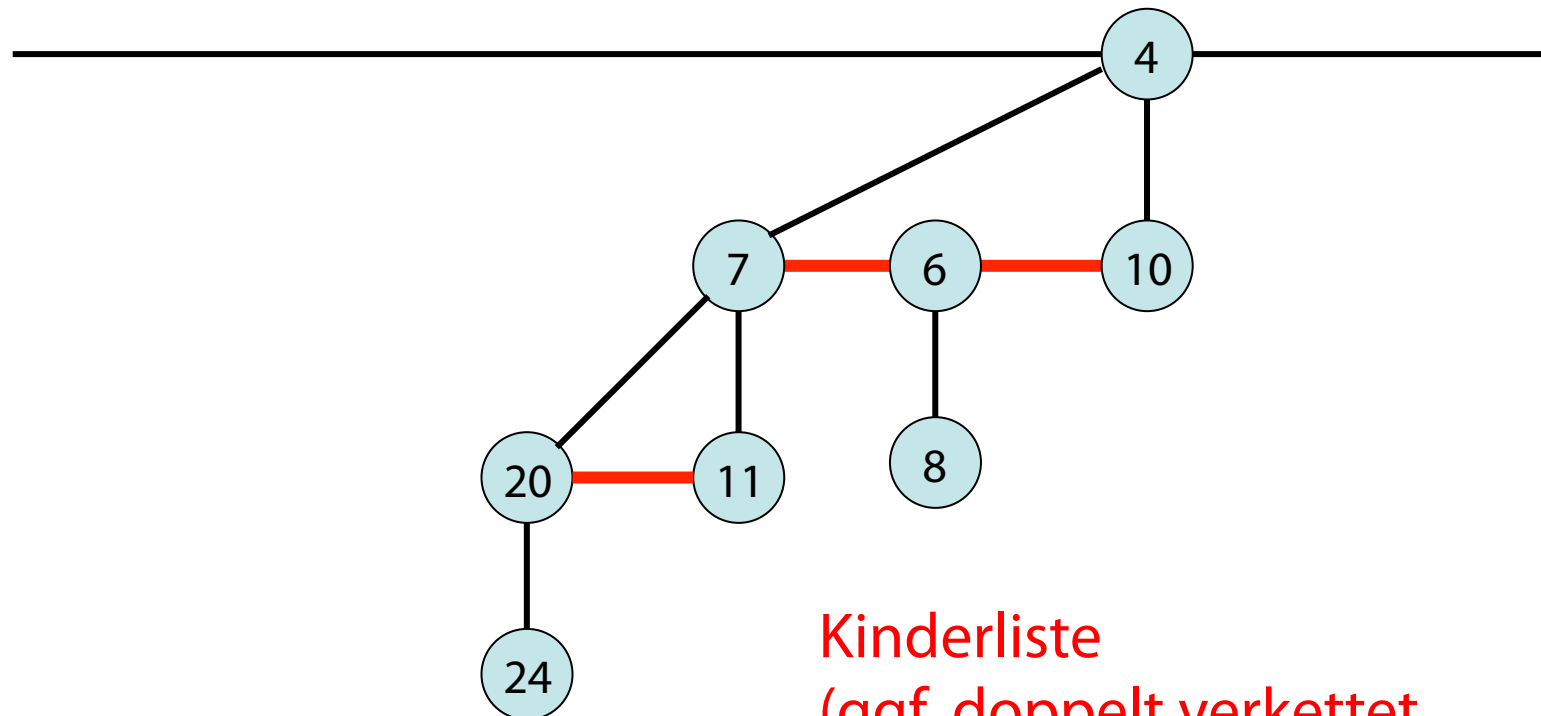
Rang = Anzahl der Kinder des Wurzelknotens



# Fibonacci-Heap: Anpassung der Struktur

Baum in Fibonacci-Heap: Zentrale Invariante

**Rang = Anzahl der Kinder des Wurzelknotens**



**Kinderliste**  
(ggf. doppelt verkettet  
dito für Wurzelliste)

# Fibonacci-Heap

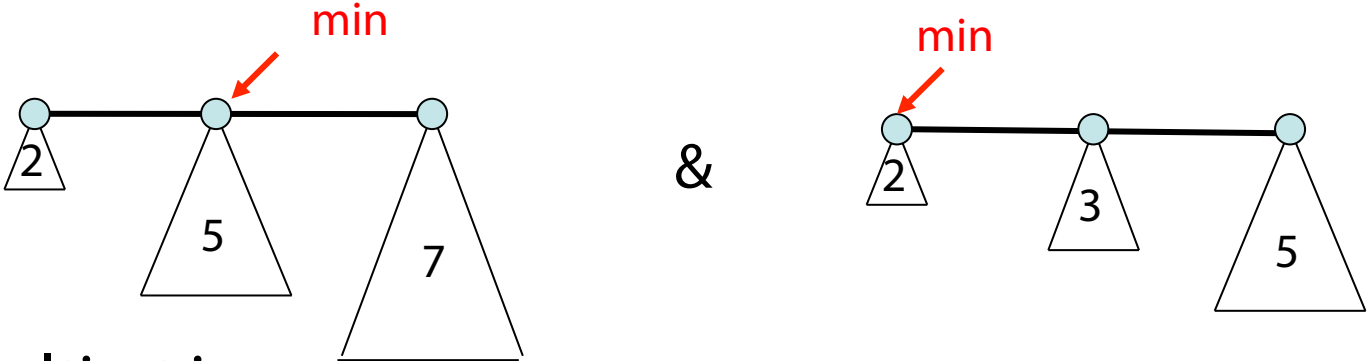
---

Jeder Knoten merkt sich:

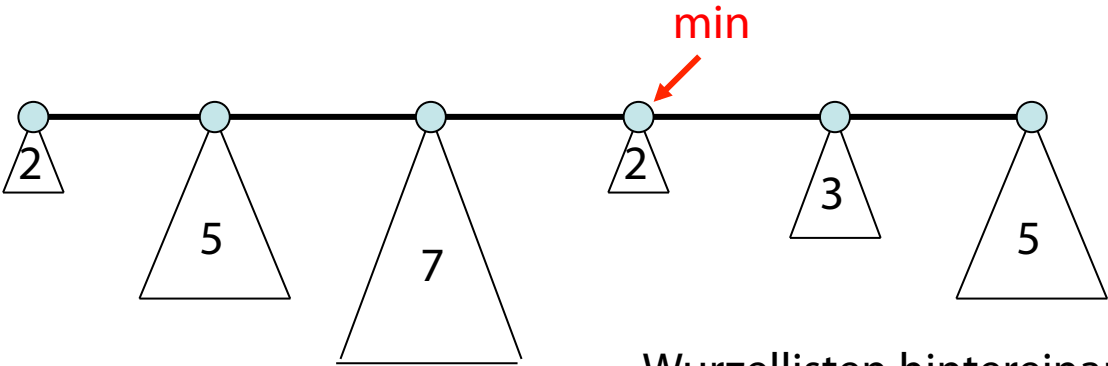
- im Knoten gespeichertes Element
- Vater
- Liste der Kinder (mit Start- und Endpunkt)
- **Rang**

# Fibonacci-Heap: Lazy-Merge

Lazy merge von



resultiert in



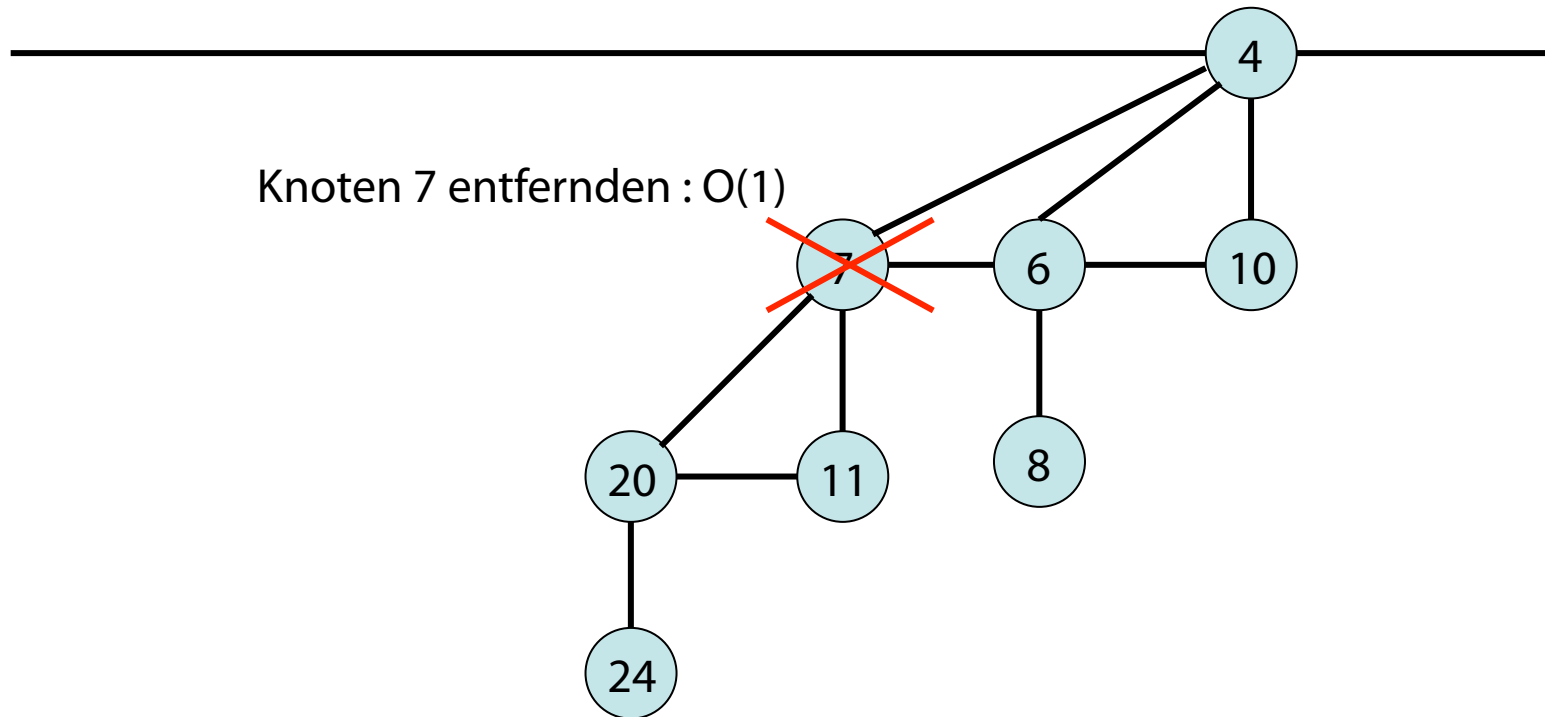
Wurzellisten hintereinanderhängen:  $O(1)$

# Fibonacci-Heap: Lazy-Delete

Lazy delete:

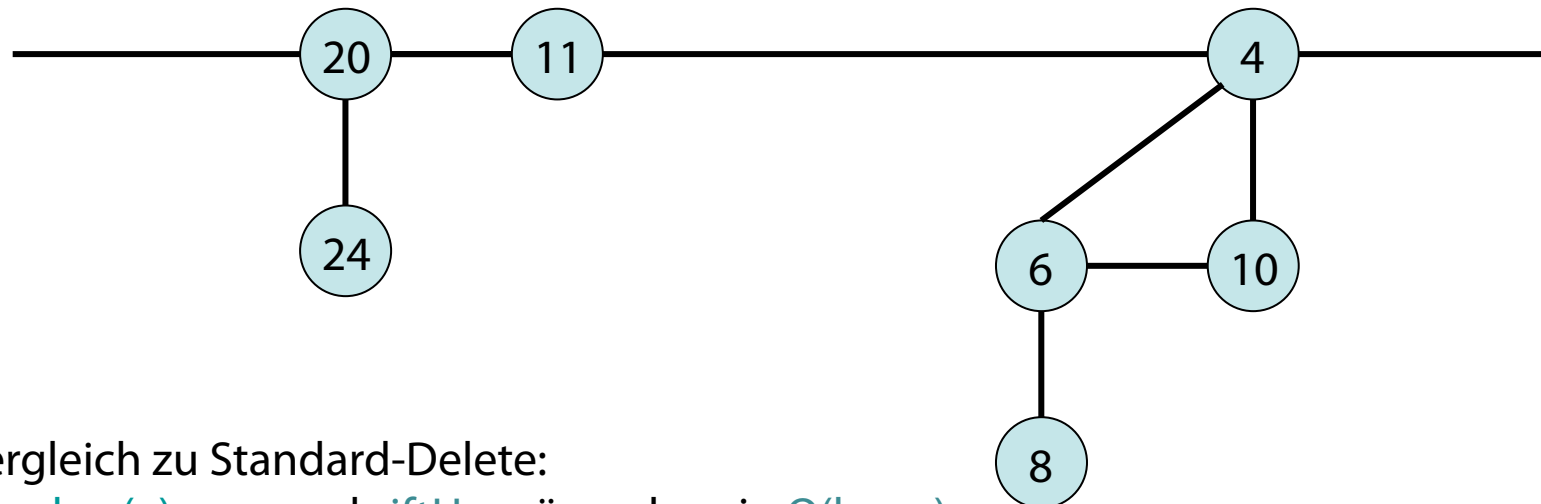
Kinderliste in Wurzelliste integrieren:  $O(1)$

Knoten 7 entfernen:  $O(1)$



# Fibonacci-Heap: Lazy-Delete

Lazy delete: **Annahme:** Knoten 7 ist im direkten Zugriff  
**Aufwand:**  $O(1)$ , da gegebener Knoten 7 in  $O(1)$  Zeit entfernbar und Kinderliste von 7 in  $O(1)$  Zeit in Wurzelliste integrierbar

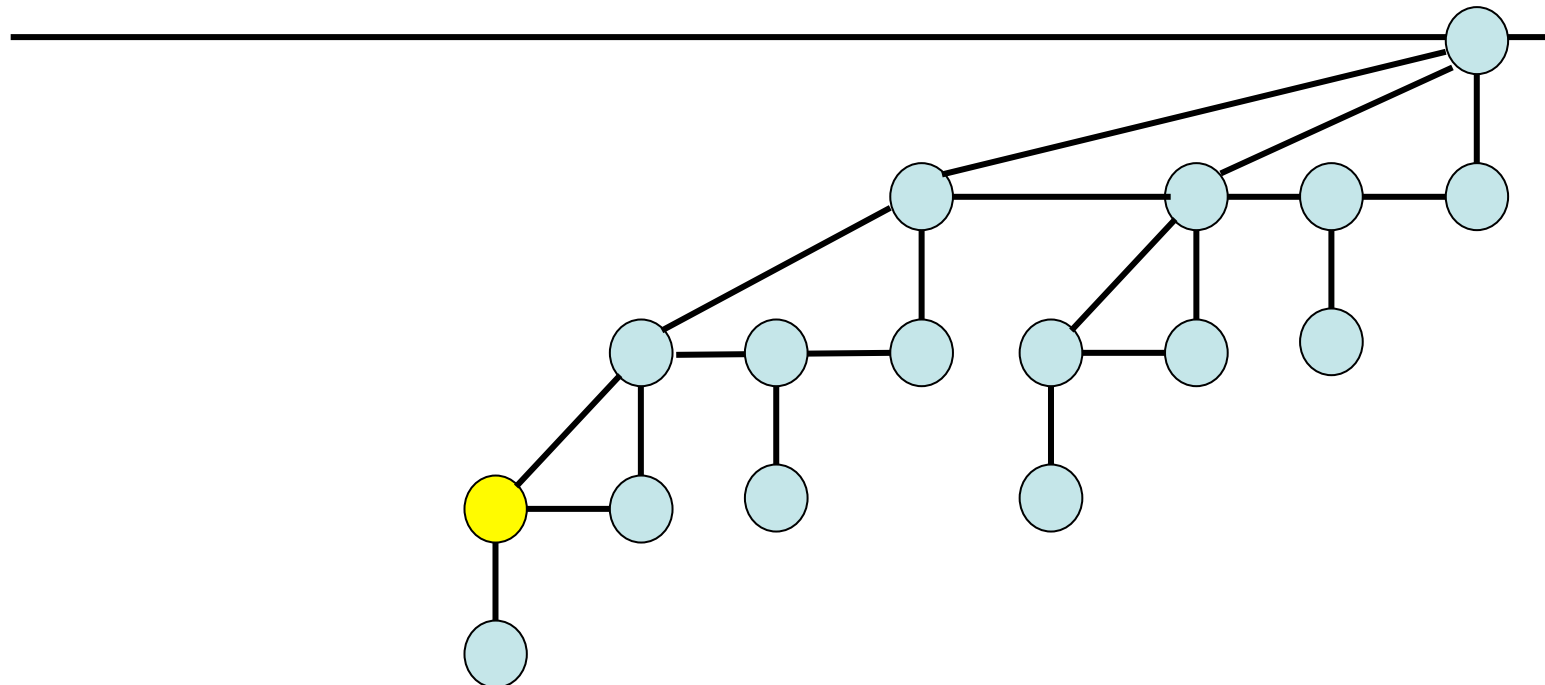


- Vergleich zu Standard-Delete:
  - $key(e) := -\infty$  und `siftUp` wäre schon in  $O(\log n)$
- Also kein `siftUp`
- Problem:
  - Bäume gleichen Ranges treten in der Wurzelliste auf
  - Binomial-Heap-Eigenschaft kann verletzt sein

# Fibonacci-Heap

---

Beispiel für delete(x)





# Fibonacci-Heap: Übersicht

---

## Operationen:

- **merge**: Verbinde Wurzellisten, aktualisiere min-Pointer: Zeit  $O(1)$
- **insert(x, pq)**: Füge  $B_0$  (mit  $x$ ) in Wurzelliste ein, aktualisiere min-Pointer. Zeit  $O(1)$
- **min(pq)**: Gib Element, auf das der min-Pointer zeigt, zurück. Zeit  $O(1)$
- **deleteMin(pq), delete(x, pq), decreaseKey(x, pq,  $\Delta$ )**: noch zu bestimmen...

# Fibonacci-Heap

---

`deleteMin(pq)`: Diese Operation hat Aufräumfunktion.  
Der min-Pointer zeige auf  $x$ .

## **procedure** deleteMin()

*entferne  $x$  aus Wurzelliste*

*konkatiniere Kinderliste von  $x$  mit Wurzelliste*

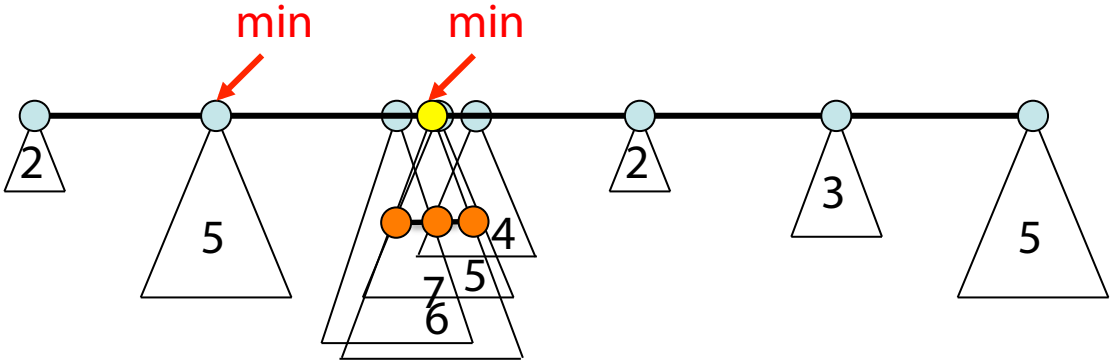
**while**  $\geq 2$  Bäume mit gleichem Rang **i do**

*merge Bäume zu Baum mit Rang  $i+1$  // (wie bei zwei Binomial-Bäumen)*

*aktualisiere den min-Pointer*

- Durch die Integration der Kinderliste in die Wurzelliste können dort Bäume gleichen Ranges auftreten, die Struktur wird jedoch danach konsolidiert
- Die schon durch `delete` auftretenden Heaps gleichen Ranges werden gleich mit behandelt!

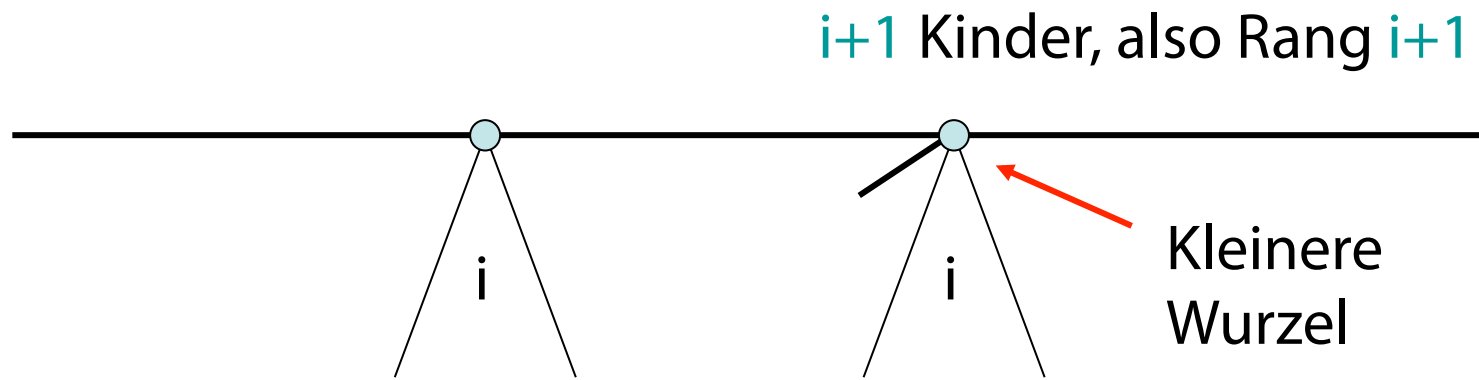
# Beispiel: deleteMin



vorher

# Fibonacci-Heap

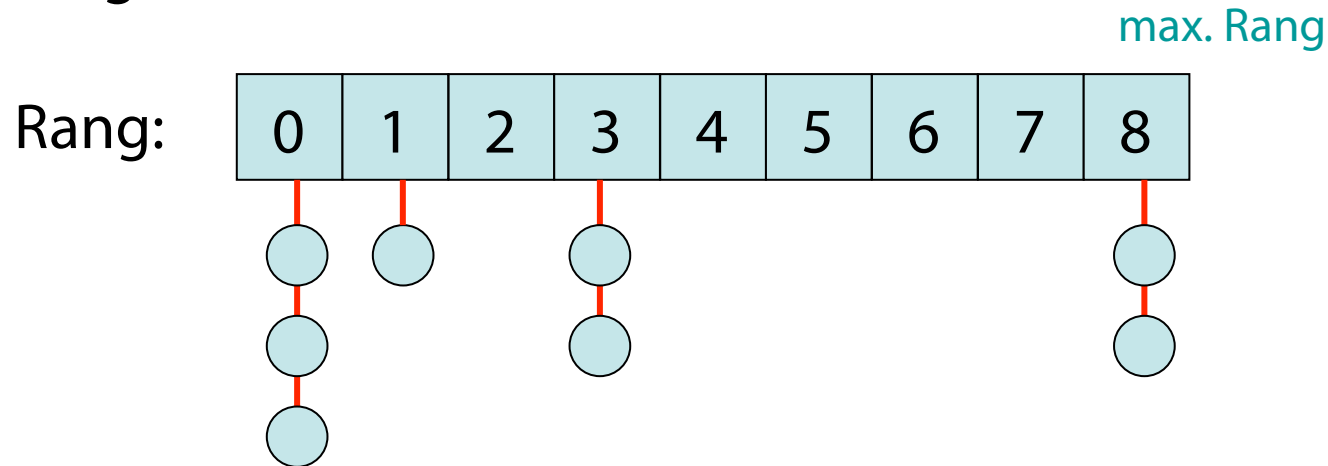
Verschmelzung zweier Bäume mit Rang  $i$   
(d.h. Wurzel hat  $i$  Kinder):



# Fibonacci-Heap

Effiziente Findung von Wurzeln mit gleichem Rang:

- Scanne vor while-Schleife alle Wurzeln und speichere diese nach Rängen in Feld (Eimerkette!):



- Merge dann wie bei Binomialbäumen von Rang 0 an bis maximaler Rang erreicht (wie Binäraddition)

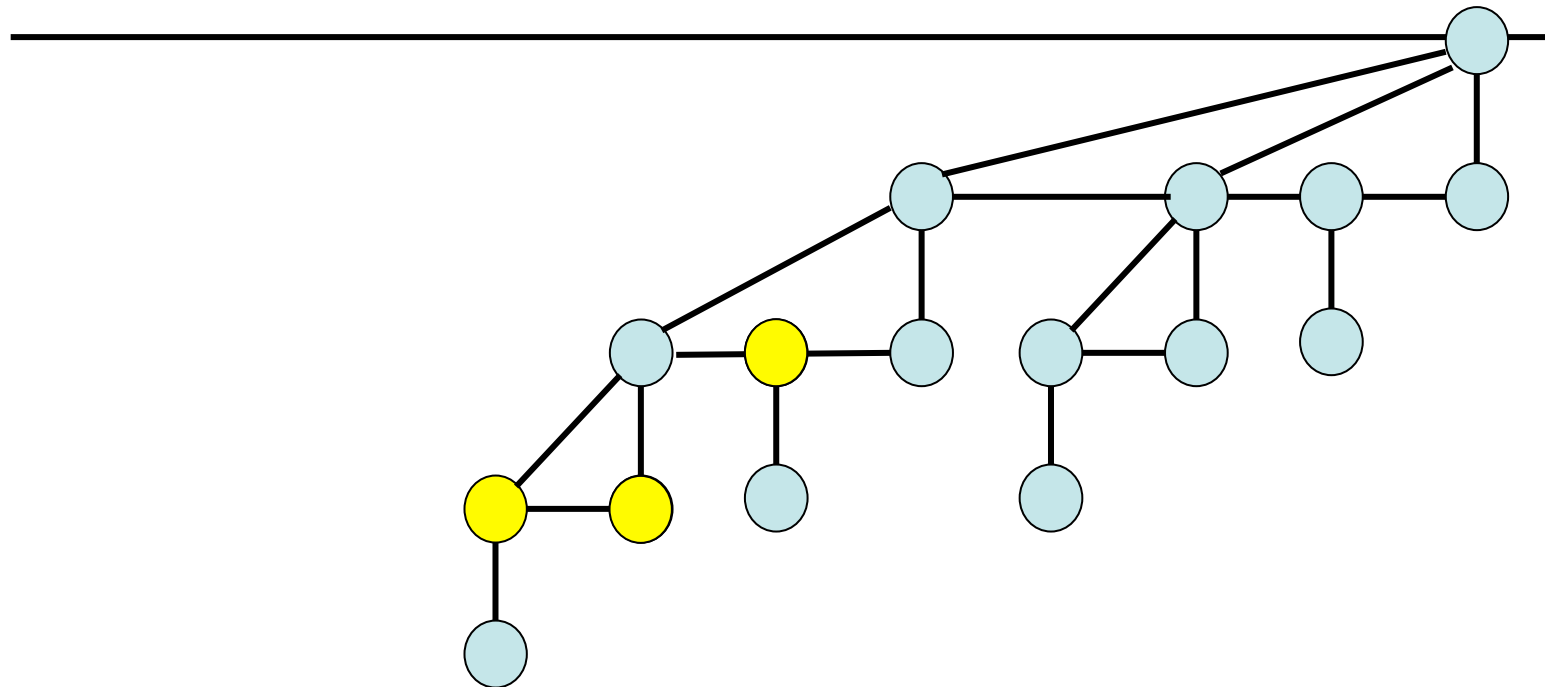
# Fibonacci-Heap

Sei  $\text{parent}(x)$  Vater von Knoten  $x$ . Wenn  $x$  neu eingefügt wird, ist  $\text{Mark}(x)=0$ . ( $\text{Mark}(x)$  speichert, ob Kinder schon weg.)

```
procedure delete(x):  
  if  $x$  ist min-Wurzel then  
    deleteMin()  
  else  
    lösche  $x$ , füge Kinder von  $x$  in Wurzelliste ein  
  if not parentExists(x) then  
    exit //  $x$  ist Wurzel  
  while true do  
     $x:=\text{parent}(x)$   
    if not parentExists(x) then  
      exit //  $x$  ist Wurzel  
    if  $\text{Mark}(x)=0$  then  
       $\text{Mark}(x):=1$   
      exit  
    else //  $\text{Mark}(x)=1$ , also schon Kind weg  
      hänge  $x$  samt Unterbaum in Wurzelliste  
       $\text{Mark}(x):=0$  // Wurzeln benötigen kein Mark
```

# Fibonacci-Heap

Beispiel für delete(x): (● : Mark=1)



# Fibonacci-Heap

---

```
procedure decreaseKey(x,Δ):  
    füge x samt Unterbaum in Wurzelliste ein  
    key(x):=key(x)-Δ  
    aktualisiere min-Pointer  
if not parentExists(x) then  
    exit // war x Wurzel?  
while true do  
    x:=parent(x)  
    if not parentExists(x) then  
        exit // x ist Wurzel  
    if Mark(x)=0 then  
        Mark(x):=1  
        exit  
    else // Mark(x)=1  
        hänge x samt Unterbaum in Wurzelliste  
        Mark(x):=0
```



# Fibonacci-Heap mit markierten Fehlern

---

## Zeitaufwand:

- `deleteMin()`:  
 $O(\text{max. Rang} + \#\text{Baumverschmelzungen})$
- `delete(x)`, `decreaseKey(x, Δ)`:  
 $O(1 + \#\text{kaskadierende Schnitte})$   
d.h.  $\#\text{umgehänger markierter Knoten}$

## Wir werden sehen:

Zeitaufwand kann in beiden Fällen  $O(n)$  sein,  
aber richtig verteilt viel günstiger.

# Strukturfehler: Verschiebung der Arbeit

---

- Statt bei jedem **merge** und **decreaseKey** einen Aufwand von  $O(\log n)$  zu leisten, ...
- ... wird die Arbeit bei einem **deleteMin** mit übernommen, ...
- ... mit der Idee, dass man die entsprechenden Strukturen dort sowieso anfassen muss
- Das Umverteilen kann sich über eine längere Sequenz von Operationen durchaus amortisieren
- Vgl. **build** für binäre Heaps



# Amortisierte Analyse

---

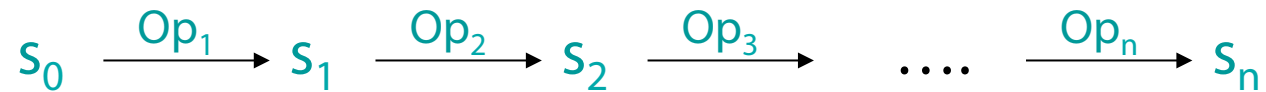
Betrachte Folge von  $n$  Operationen auf anfangs leerem Fibonacci-Heap.

- Summierung der worst-case Kosten viel zu hoch!
- Average-case Analyse auch nicht sehr aussagekräftig
- **Besser: amortisierte Analyse**, d.h. durchschnittliche Kosten aller Operationen der Sequenz im worst-case Fall (teuerste **Folge**)

# Amortisierte Analyse

---

- $S$ : Zustandsraum einer Datenstruktur
- $F$ : beliebige Folge von Operationen  
 $Op_1, Op_2, Op_3, \dots, Op_n$
- $s_0$ : Anfangszustand der Datenstruktur



- Zeitaufwand  $T(F) = \sum_{i=1}^n T_{Op_i}(s_{i-1})$

# Amortisierte Analyse

---

- Zeitaufwand  $T(F) = \sum_{i=1}^n T_{Op_i}(s_{i-1})$
- Eine Familie von Funktionen  $A_X(s)$ , eine pro Operation  $X \in \{Op_1, Op_2, Op_3, \dots, Op_n\}$ , heißt **Familie amortisierter Zeitschranken** falls für jede Sequenz  $F$  von Operationen gilt

$$T(F) \leq A(F) := c + \sum_{i=1}^n A_{Op_i}(s_{i-1})$$

für eine Konstante  $c$  unabhängig von  $F$

# Amortisierte Analyse: Potentialmethode

---

**Behauptung:** Sei  $S$  der Zustandsraum einer Datenstruktur, sei  $s_0$  der Anfangszustand und sei  $\phi: S \rightarrow \mathbb{R}_{\geq 0}$  eine nichtnegative Funktion.

$\phi: S \rightarrow \mathbb{R}_{\geq 0}$  wird auch **Potential** genannt.

Für eine Operation  $X$  und einen Zustand  $s$  mit  $s \xrightarrow{X} s'$  definiere  $A_X(s)$  über die **Potentialdifferenz**:

$$A_X(s) := \phi(s') - \phi(s) + T_X(s) := \Delta\phi(s) + T_X(s)$$

Dann sind die Funktionen  $A_X(s)$  eine **Familie amortisierter Zeitschranken.**

# Amortisierte Analyse: Potentialmethode

---

Zu zeigen:  $T(F) \leq c + \sum_{i=1}^n A_{Op_i}(s_{i-1})$

Beweis:

$$\begin{aligned}\sum_{i=1}^n A_{Op_i}(s_{i-1}) &= \sum_{i=1}^n [\phi(s_i) - \phi(s_{i-1}) + T_{Op_i}(s_{i-1})] \\ &= T(F) + \sum_{i=1}^n [\phi(s_i) - \phi(s_{i-1})] \\ &= T(F) + \phi(s_n) - \phi(s_0)\end{aligned}$$

$$\begin{aligned}\models T(F) &= \sum_{i=1}^n A_{Op_i}(s_{i-1}) + \phi(s_0) - \phi(s_n) \\ &\leq \sum_{i=1}^n A_{Op_i}(s_{i-1}) + \phi(s_0) \text{ konstant}\end{aligned}$$

# Amortisierte Analyse: Potentialmethode

---

Für Fibonacci-Heaps verwenden wir für das Potential den Begriff Balance (bal)

$\text{bal}(s) := \# \text{Bäume} + 2 \cdot \# \text{markierte Knoten}$   
im Zustand  $s$



# Fibonacci-Heap: Eigenschaften

---

**Invariante 1:** Sei  $x$  ein Knoten im Fibonacci-Heap mit  $\text{Rang}(x)=k$ . Seien die Kinder von  $x$  sortiert in der Reihenfolge ihres Anfügens an  $x$ . Dann ist der Rang des  $i$ -ten Kindes  $\geq i-2$ .

**Beweis der Gültigkeit:**

- Beim Einfügen des  $i$ -ten Kindes ist  $\text{Rang}(x)=i-1$ .
- Das  $i$ -te Kind hat zu dieser Zeit auch Rang  $i-1$ .
- Danach verliert das  $i$ -te Kind höchstens eines seiner Kinder<sup>1</sup>, d.h. sein Rang ist  $\geq i-2$ .

<sup>1</sup> Bei einem schon markierten Vater eines gelöschten Knotens wird schon konsolidiert

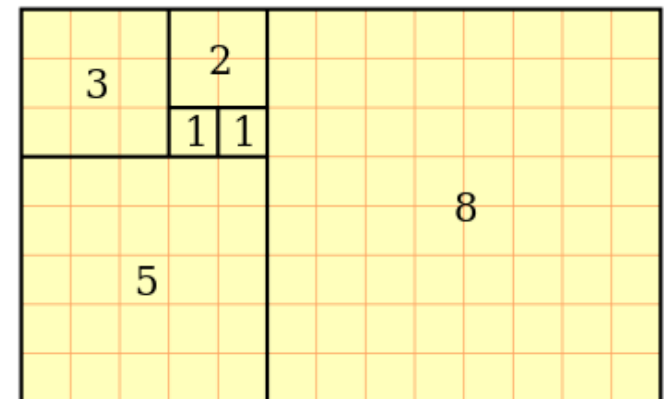
# Fibonacci-Heap: Eigenschaften

**Invariante 2:** Sei  $x$  ein Knoten im Fibonacci-Heap mit  $\text{Rang}(x)=k$ . Dann enthält der Baum mit Wurzel  $x$  mindestens  $F_{k+2}$  Elemente, wobei  $F_{k+2}$  die  $(k+2)$ -te Fibonacci-Zahl ist.

**Einschub: Definition der Fibonacci-Zahlen:**

- $F_0 = 0$  und  $F_1 = 1$
- $F_i = F_{i-1} + F_{i-2}$  für alle  $k > 1$

Daraus folgt, dass  $F_{i+2} = 1 + \sum_{j=0}^i F_j$   
(siehe Tafel)



[Wikipedia]

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, usw.



# Fibonacci-Heap

---

## Beweis der Gültigkeit von Invariante 2:

- Sei  $f_k$  die minimale Anzahl von Elementen in einem Baum mit Rang  $k$ .

- Aus Invariante 1 folgt:

$$f_k \geq f_{k-2} + f_{k-3} + \dots + f_0 + 1 + 1$$

- Weiterhin ist  $f_0=1$  und  $f_1=2$

- Also folgt nach den Fibonacci-Zahlen:

$$f_k \geq F_k$$

1. Kind

Wurzel

# Fibonacci-Heap

---

- Man hat herausgefunden, dass  $F_k > \Phi^k$  ist für

$$\Phi = (1 + \sqrt{5}) / 2 \approx 1,618034$$

- D.h. ein Baum mit Rang  $k$  im Fibonacci-Heap hat mindestens  $1,61^k$  Knoten.
- Ein **Fibonacci-Heap** aus  $n$  Elementen hat also **Bäume vom Rang maximal  $O(\log n)$**  (wie bei Binomial-Heap)

Warum?  
(s.Tafel)

<http://www.was-darwin-nicht-wusste.de/wunder/mathematische-ueberraschungen.html>

# aus: [Wikipedia]

## Verwandtschaft mit dem Goldenen Schnitt [\[Bearbeiten\]](#)

Wie von [Johannes Kepler](#) festgestellt wurde, nähert sich der [Quotient](#) zweier aufeinander folgender Fibonacci-Zahlen dem [Goldenen Schnitt](#)  $\Phi$  an. Dies folgt unmittelbar aus der [Näherungsformel](#) für große  $n$ :

$$\lim_{n \rightarrow \infty} \frac{f_{n+1}}{f_n} = \lim_{n \rightarrow \infty} \frac{\Phi^{n+1}}{\Phi^n} = \Phi \approx 1,618 \dots$$

Diese Quotienten zweier aufeinander folgender Fibonacci-Zahlen haben eine bemerkenswerte [Kettenbruchdarstellung](#)

$$\frac{1}{1} = 1 \quad \frac{2}{1} = 1 + \frac{1}{1} \quad \frac{3}{2} = 1 + \frac{1}{1 + \frac{1}{1}} \quad \frac{5}{3} = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}} \quad \frac{8}{5} = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}$$

Da diese Quotienten im Grenzwert gegen den goldenen Schnitt konvergieren, lässt sich dieser als der unendliche Kettenbruch

$$\Phi = 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \dots}}}}$$

darstellen.

Die Zahl  $\Phi$  ist [irrational](#). Das bedeutet, dass sie sich nicht durch ein Verhältnis zweier ganzer Zahlen darstellen lässt, ein Umstand, der wesentlich zu ihrer Bedeutung in Kunst und Natur beiträgt. Am besten lässt sich  $\Phi$  durch Quotienten zweier aufeinander folgender Fibonacci-Zahlen darstellen. Dies gilt auch für verallgemeinerte Fibonaccifolgen, bei denen  $f_0$  und  $f_1$  beliebige natürliche Zahlen annehmen.

# Fibonacci-Heap: insert, merge, min

- $t_i$ : Zeit für Operation  $i$  ausgeführt im Zustand  $s$
- $a_i$ : amortisierter Aufwand für Operation  $i$   
 $a_i = t_i + \Delta bal_i$  mit  $\Delta bal_i = bal(s') - bal(s)$ , falls  $i: s \rightarrow s'$  im aktuellen Zustand  $s$  ausgeführt wird

## Amortisierte Kosten der Operationen:

$$bal(s) = \#B\ddot{a}ume(s) + 2 \cdot \#markierte\ Knoten(s)$$

- insert:  $t=O(1)$  und  $\Delta bal_{insert} = +1$ , also  $a=O(1)$
- merge:  $t=O(1)$  und  $\Delta bal_{merge} = 0$ , also  $a=O(1)$
- min:  $t=O(1)$  und  $\Delta bal_{min} = 0$ , also  $a=O(1)$

# Fibonacci-Heap: deleteMin

## Behauptung:

Die amortisierten Kosten von deleteMin() sind  $O(\log n)$ .

## Beweis:

- Einfügen der Kinder von  $x$  in Wurzelliste ( $\#Kinder(x) = Rang(x)$ ):  
 $\Delta bal_1 = Rang(x) - 1$  (-1 weil  $x$  entfernt wird)
- Jeder Merge-Schritt verkleinert #Bäume um 1:  
 $\Delta bal_2 = -(\#Merge-Schritte)$
- Wegen Invariante 2 (Rang der Bäume max.  $O(\log n)$ ) gilt:  $\#Merge-Schritte = \#Bäume - O(\log n)$
- Insgesamt:  $\Delta bal_{deleteMin} = Rang(x) - \#Bäume + O(\log n)$
- Laufzeit (in geeigneten Zeiteinheiten):  
 $t_{deleteMin} = \#Bäume + O(\log n)$
- Amortisierte Laufzeit:  
 $a_{deleteMin} = t_{deleteMin} + \Delta bal_{deleteMin} = O(\log n)$

Warum?

# Fibonacci-Heap: delete

**Behauptung:** Die amortisierten Kosten von `delete(x)` sind  $O(\log n)$ .

**Beweis:** (x ist kein min-Element – sonst wie oben)

- Einfügen der Kinder von `x` in Wurzelliste:  
 $\Delta \text{bal}_1 \leq \text{Rang}(x)$
- Jeder kaskadierende Schnitt (Entfernung eines markierten Knotens) erhöht die Anzahl Bäume um 1:  
 $\Delta \text{bal}_2 = \# \text{kaskadierende Schnitte}$
- Jeder kaskadierende Schnitt entfernt eine Markierung:  
 $\Delta \text{bal}_3 = -2 \cdot \# \text{kaskadierende Schnitte}$
- Der letzte Schnitt erzeugt evtl. eine Markierung:  
 $\Delta \text{bal}_4 \in \{0, 2\}$



# Fibonacci-Heap: delete (Forts.)

Behauptung: Die amortisierten Kosten von  $\text{delete}(x)$  sind  $O(\log n)$ .

Beweis (Fortsetzung):

- Insgesamt:

$$\begin{aligned}\Delta \text{bal}_{\text{delete}} &= \text{Rang}(x) - \#\text{kaskadierende Schnitte} + O(1) \\ &= O(\log n) - \#\text{kaskadierende Schnitte}\end{aligned}$$

wegen Invariante 2

- Laufzeit (in geeigneten Zeiteinheiten):

$$t_{\text{delete}} = O(1) + \#\text{kaskadierende Schnitte}$$

- Amortisierte Laufzeit:

$$a_{\text{delete}} = t_{\text{delete}} + \Delta \text{bal}_{\text{delete}} = O(\log n)$$

# Fibonacci-Heap: decreaseKey

**Behauptung:** Die amortisierten Kosten von decreaseKey(x,Δ) sind  $O(1)$ .

**Beweis:**

- Jeder kask. Schnitt erhöht die Anzahl Bäume um 1:  
 $\Delta bal_1 = \#kaskadierende\ Schnitte$
- Jeder kask. Schnitt entfernt eine Markierung (bis auf x):  
 $\Delta bal_2 \leq -2 \cdot (\#kaskadierende\ Schnitte - 1)$
- Der letzte Schnitt erzeugt evtl. eine Markierung:  
 $\Delta bal_3 \in \{0, 2\}$
- Insgesamt:  $\Delta bal_{decreaseKey} = - \#kask. Schnitte + O(1)$
- Laufzeit:  $t_{decreaseKey} = \#kask. Schnitte + O(1)$
- Amortisierte Laufzeit:  
 $a_{decreaseKey} = t_{decreaseKey} + \Delta bal_{decreaseKey} = O(1)$

# Zusammenfassung: Laufzeitvergleich



Laufzeit	Binärer Heap	Binomial-Heap	Fibonacci-Heap
insert	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$ amor.
delete	$O(\log n)$	$O(\log n)$	$O(\log n)$ amor.
decreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$ amor.
merge	$O(n)$	$O(\log n)$	$O(1)$

# Zusammenfassung: Laufzeitvergleich



Laufzeit	Binärer Heap	Binomial-Heap	Fibonacci-Heap
insert	$O(\log n)$	$O(\log n)$	$O(1)$
min	$O(1)$	$O(1)$	$O(1)$
deleteMin	$O(\log n)$	$O(\log n)$	$O(\log n)$ amor.
delete	$O(\log n)$	$O(\log n)$	$O(\log n)$ amor.
decreaseKey	$O(\log n)$	$O(\log n)$	$O(1)$ amor.
merge	$O(n)$	$O(\log n)$	$O(1)$

Weitere Entwicklung unter Ausnutzung von Dateneigenschaften: Radix-Heap

Ahuja, Ravindra K.; Mehlhorn, Kurt; Orlin, James B.; Tarjan, Robert E.,  
 Faster algorithms for the shortest path problem, Journal of the  
 Association for Computing Machinery 37 (2): 213–223, 1990