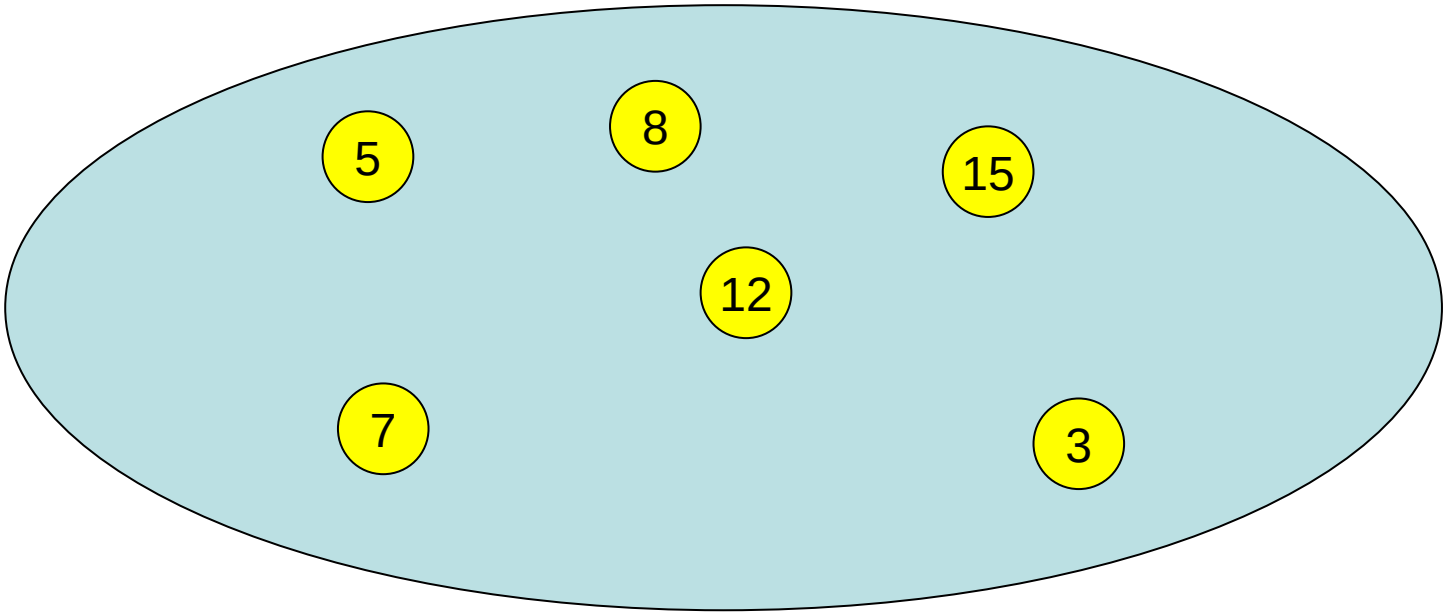
Effiziente Algorithmen und Datenstrukturen I

Kapitel 2: Priority Queues

Christian Scheideler

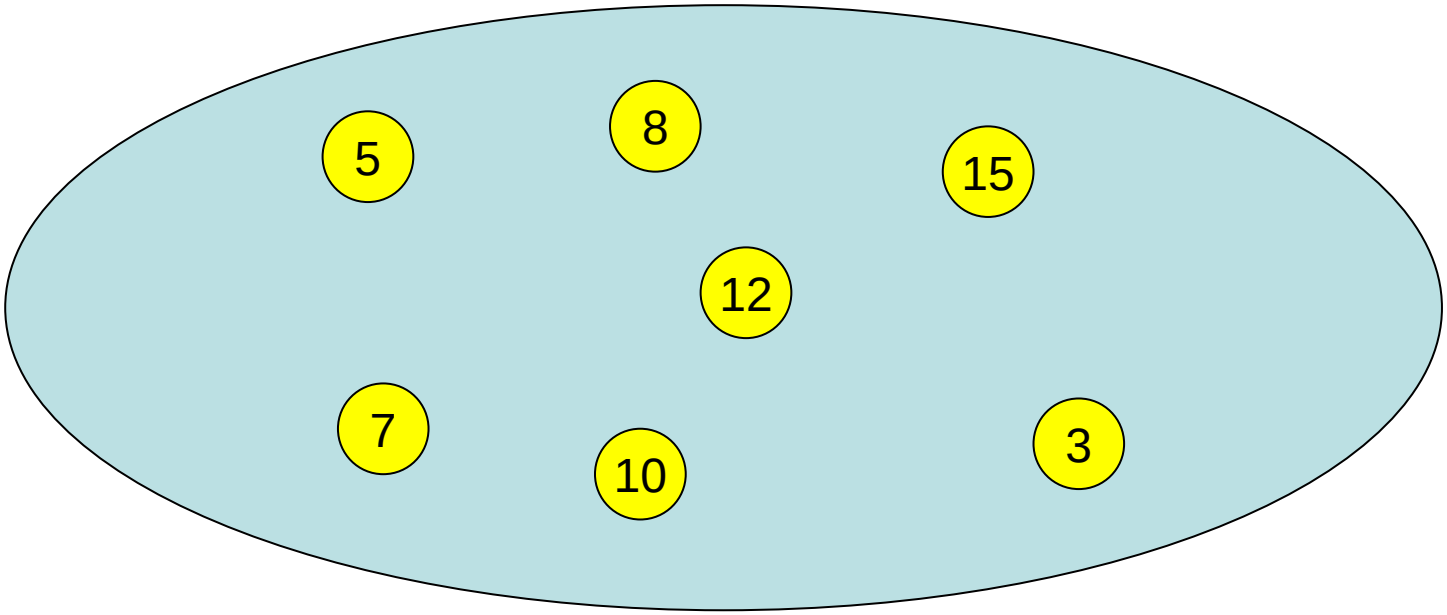
WS 2008

Priority Queue



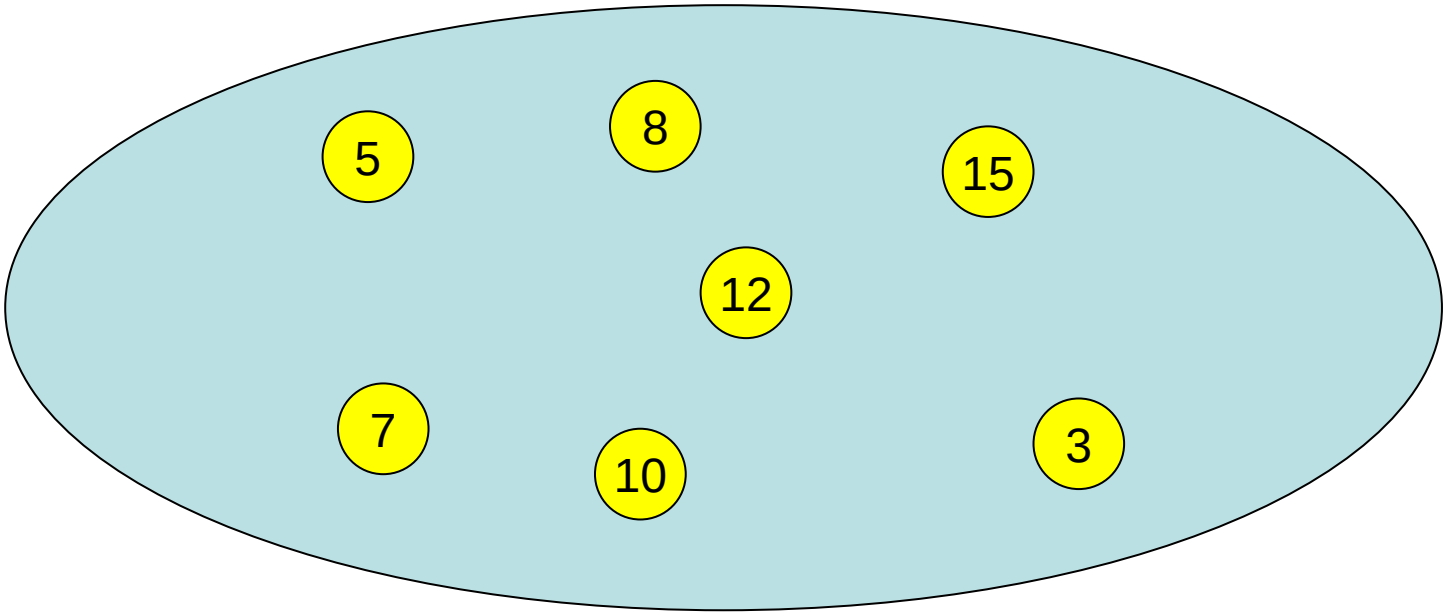
Priority Queue

insert(10)



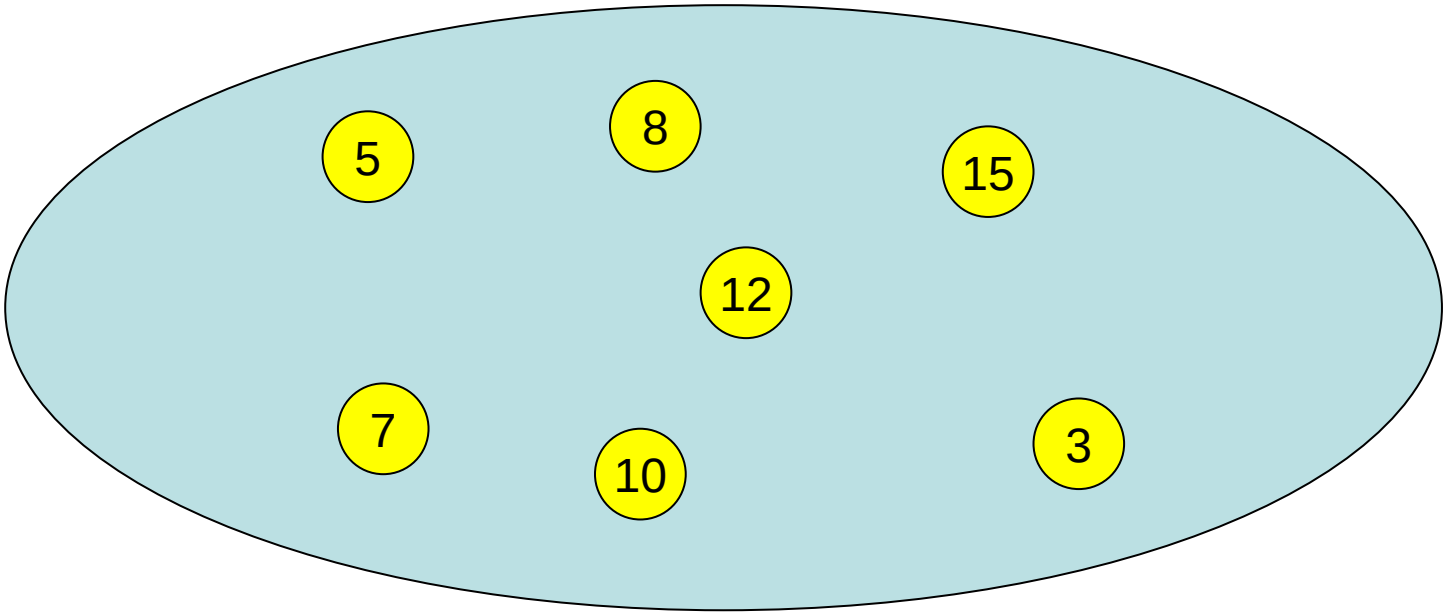
Priority Queue

min() ergibt 3 (minimales Element)



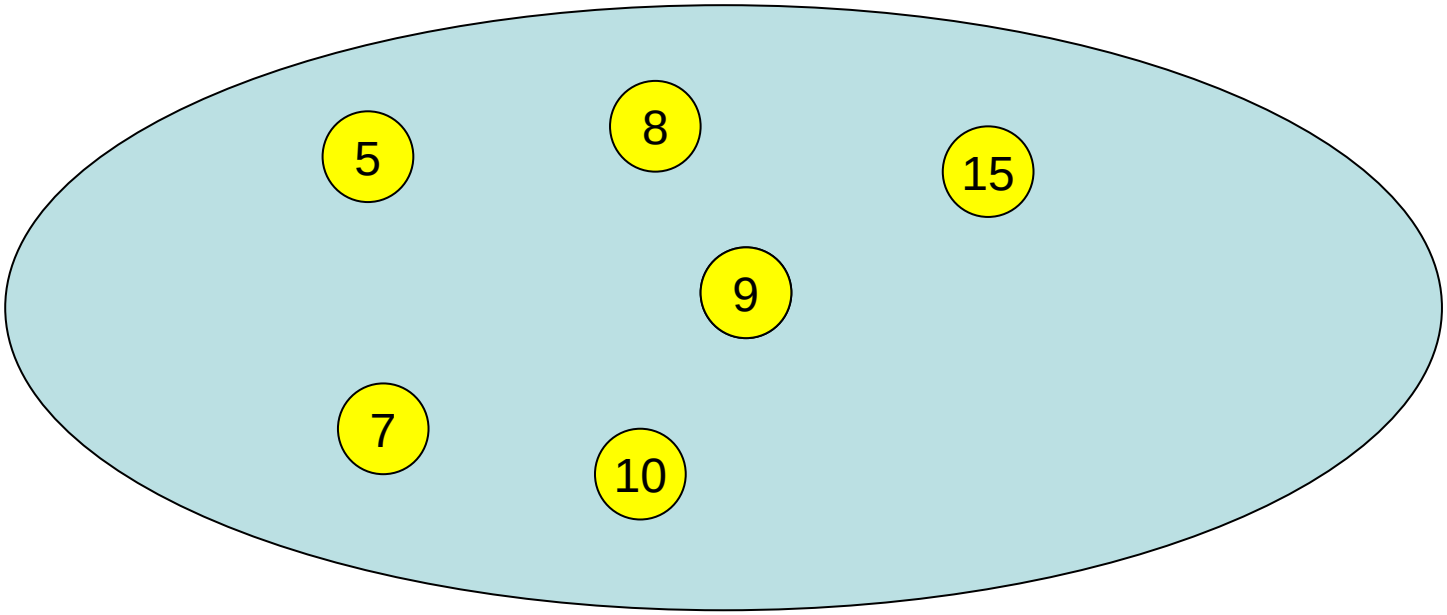
Priority Queue

deleteMin()



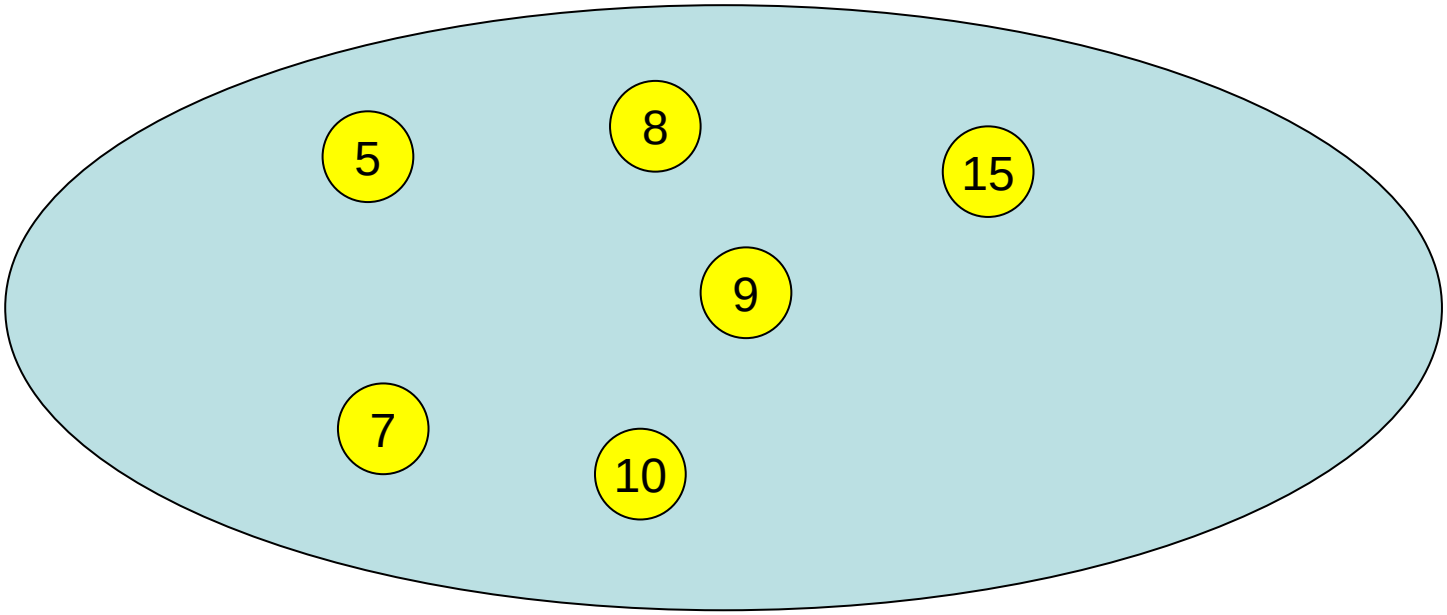
Priority Queue

decreaseKey(12,3)



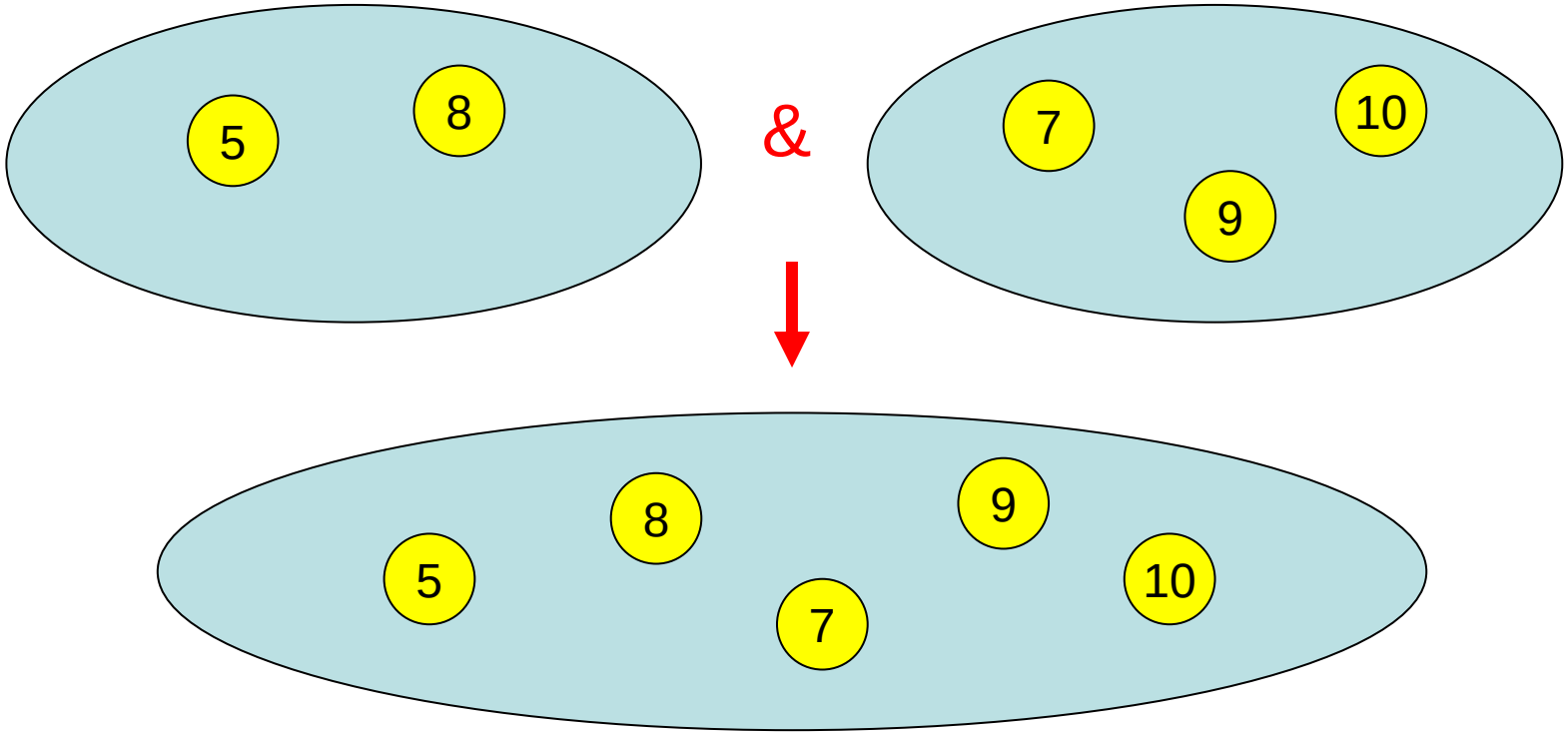
Priority Queue

delete(8)



Priority Queue

merge(Q,Q')



Priority Queue

M: Menge von Elementen

Jedes Element e identifiziert über key(e).

Operationen:

- M.build($\{e_1,\ldots,e_n\}$): M:=$\{e_1,\ldots,e_n\}$
- M.insert(e: Element): M:=M $\cup$ {e}
- M.min: gib e$\in$M mit minimalem key(e) aus
- M.deleteMin: wie M.min, aber zusätzlich M:=M\{e}, für e mit minimalem key(e)

Erweiterte Priority Queue

Zusätzliche Operationen:

- M.delete(e: Element): M:=M\{e}
- M.decreaseKey(e:Element, Δ): key(e):=key(e)-Δ
- M.merge(M´): M:=M $\cup$ M´

Priority Queue

- Priority Queue mittels unsortierter Liste:
  - build($\{e_1,\ldots,e_n\}$): Zeit O(n)
  - insert(e): O(1)
  - min, deleteMin: O(n)
- Priority Queue mittels sortiertem Feld:
  - build($\{e_1,\ldots,e_n\}$): Zeit O(n log n) (sortieren)
  - insert(e): O(n) (verschiebe Elemente in Feld)
  - min, deleteMin: O(1)

Bessere Struktur als Liste oder Feld notwendig!

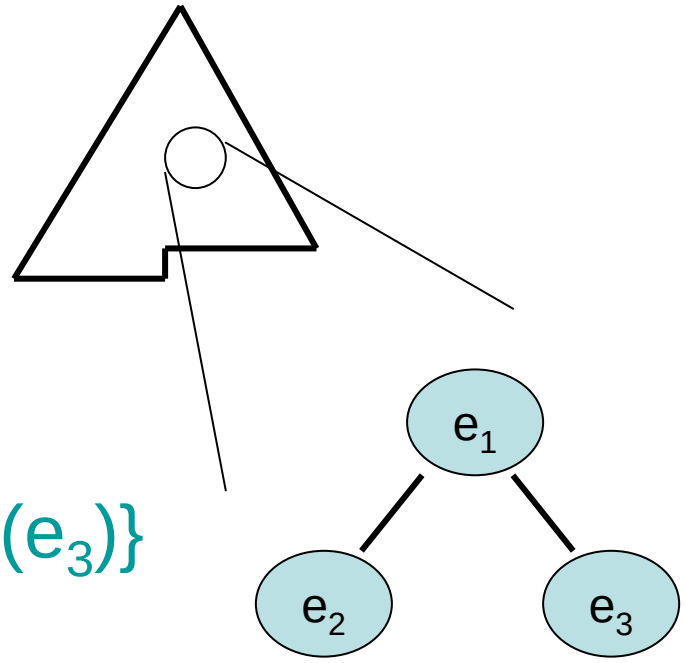
Binärer Heap

Idee: verwende binären Baum statt Liste

Bewahre zwei Invarianten:

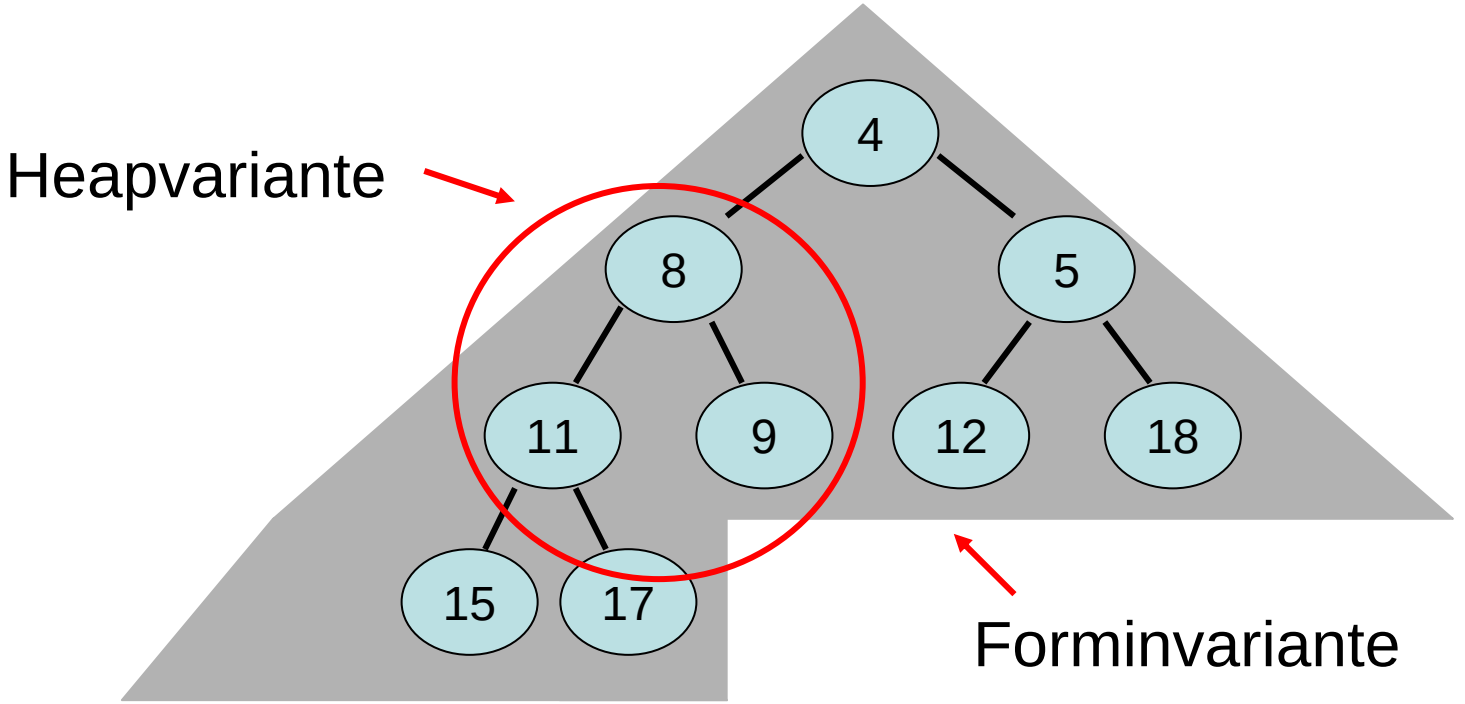
- Form-Invariante:vollst. Binärbaum bis auf unterste Ebene
- Heap-Invariante:

$key(e_1) <= min\{key(e_2), key(e_3)\}$



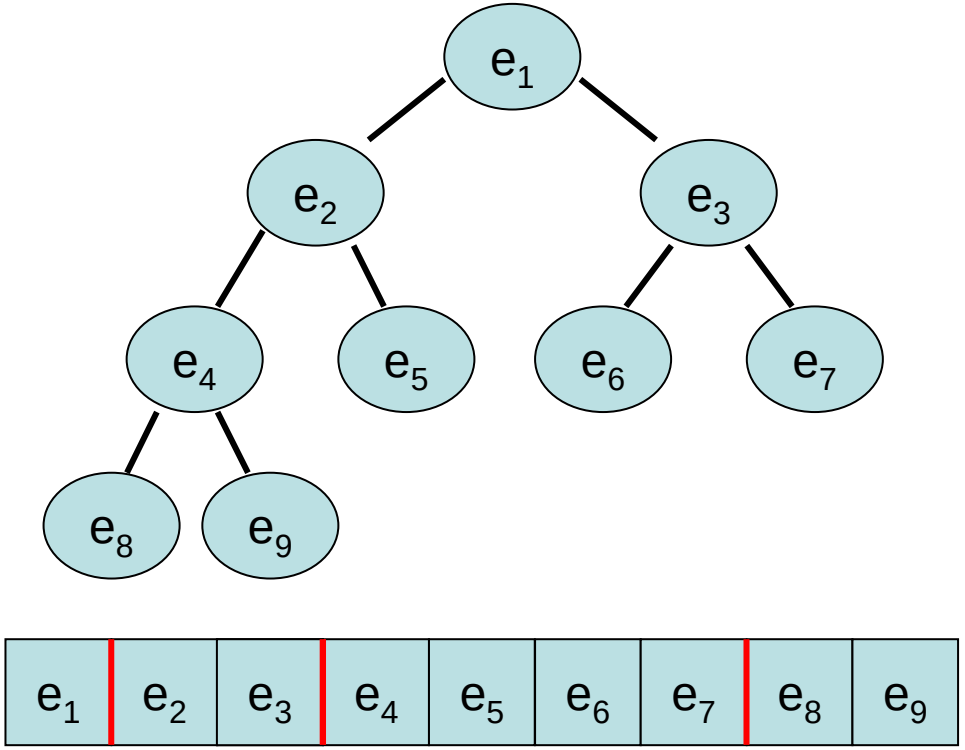
Binärer Heap

Beispiel:



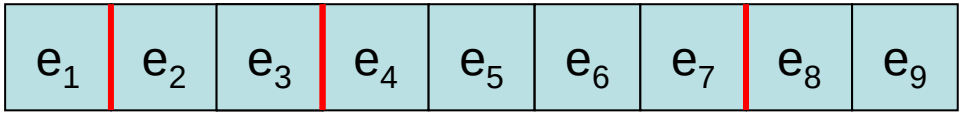
Binärer Heap

Realisierung eines Binärbaums als Feld:



Binärer Heap

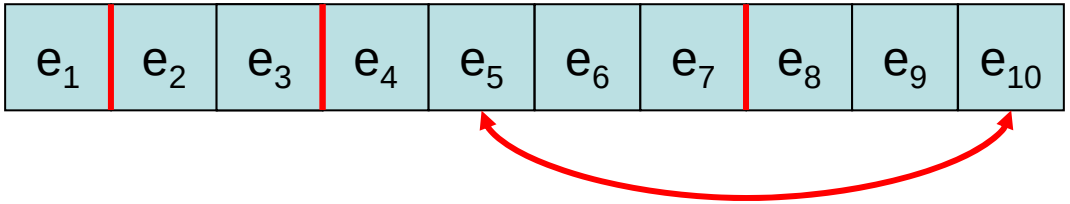
Realisierung eines Binärbaums als Feld:



- H: Array [1..n] of Element
- Kinder von e in H[i]: in H[2i], H[2i+1]
- Form-Invariante: H[1],…,H[n] besetzt
- Heap-Invariante:
  key(H[i])<=min{key(H[2i]),key(H[2i+1])}

Binärer Heap

Realisierung eines Binärbaums als Feld:



insert(e):

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

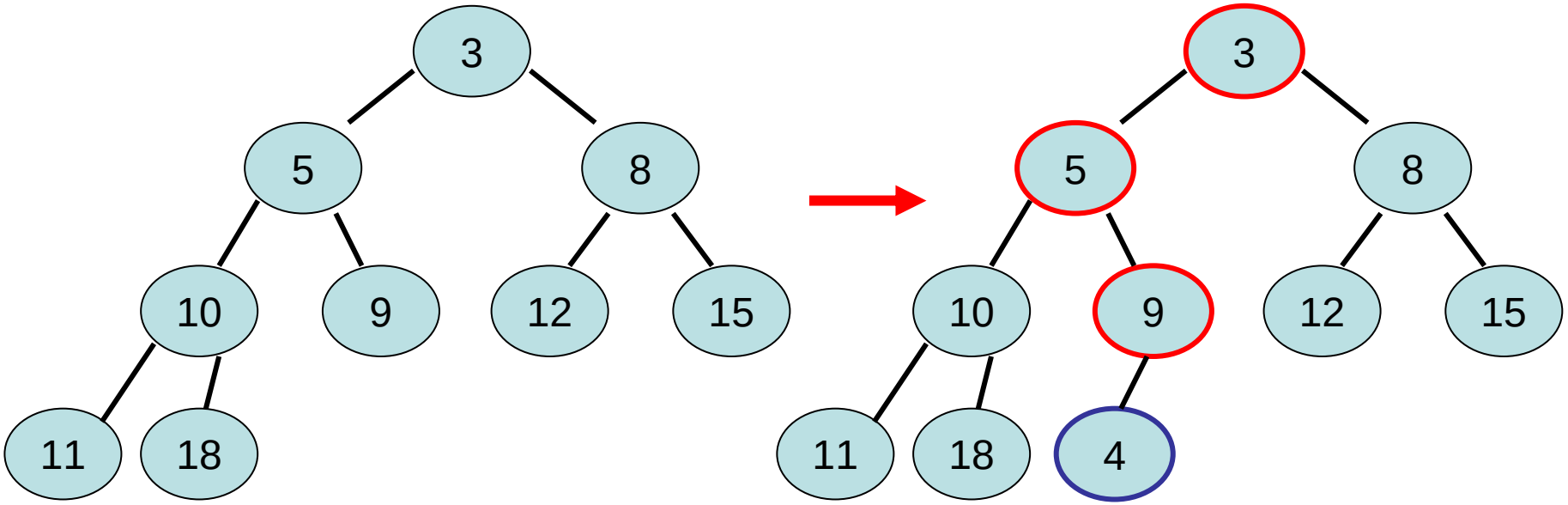
Insert Operation

Procedure insert(e: Element)
  n:=n+1; H[n]:=e
  siftUp(n)

Procedure siftUp(i: Integer)
  while i>1 and key(H[$\lfloor$i/2$\rfloor$])>key(H[i]) do
    H[i] $\leftrightarrow$ H[$\lfloor$i/2$\rfloor$]
    i:=$\lfloor$i/2$\rfloor$

Laufzeit: O(log n)

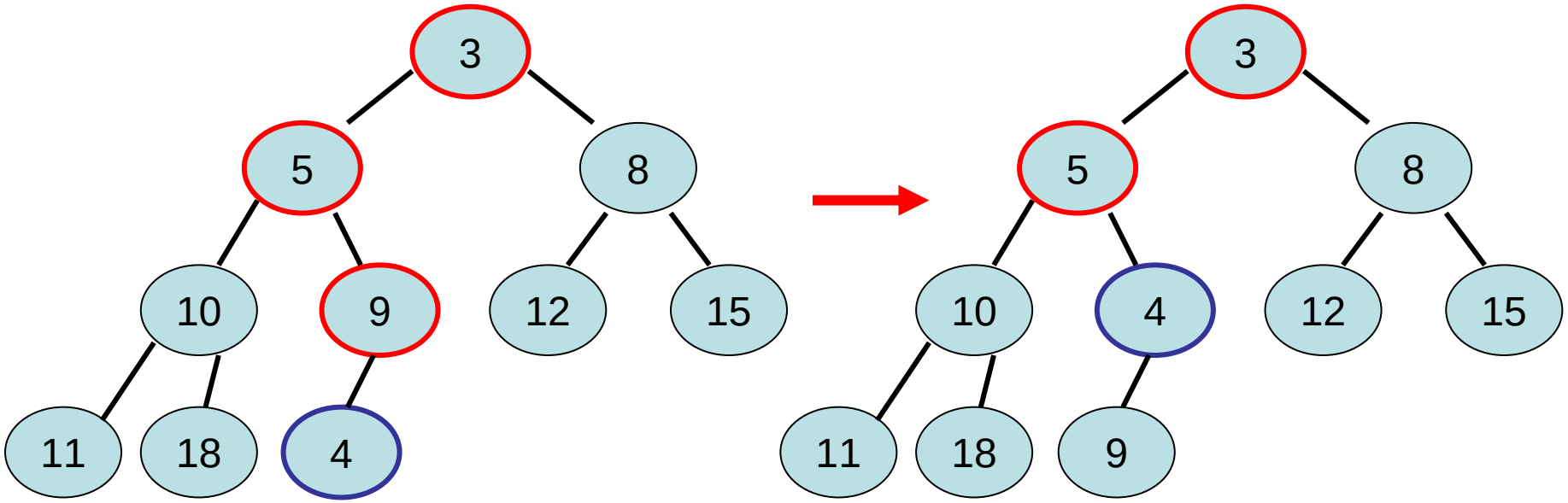
Insert Operation - Korrektheit



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

: Knoten, die Invariante eventuell verletzen

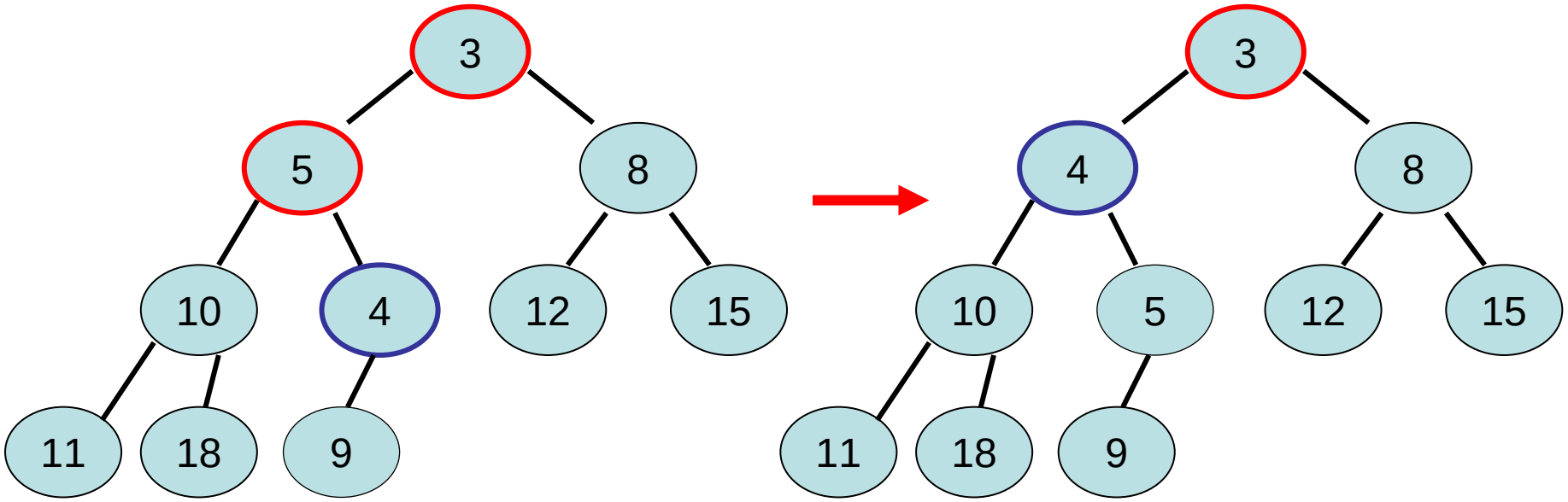
Insert Operation - Korrektheit



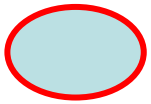
Invariante: H[k] ist minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

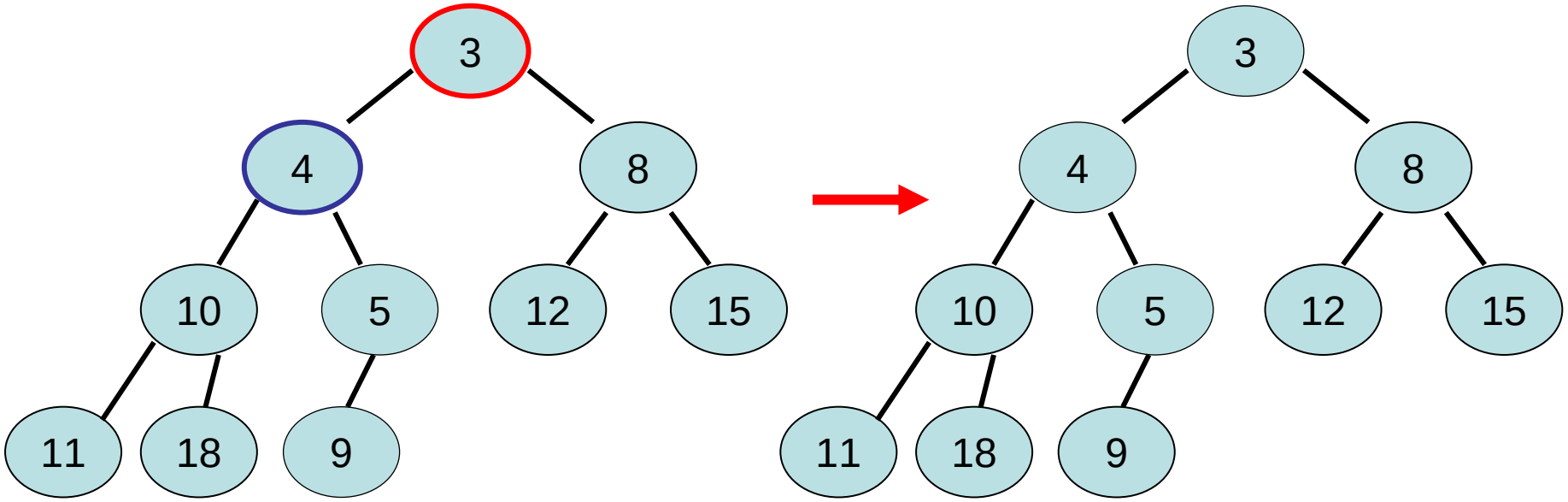
Insert Operation - Korrektheit



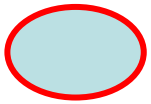
Invariante: H[k] ist minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

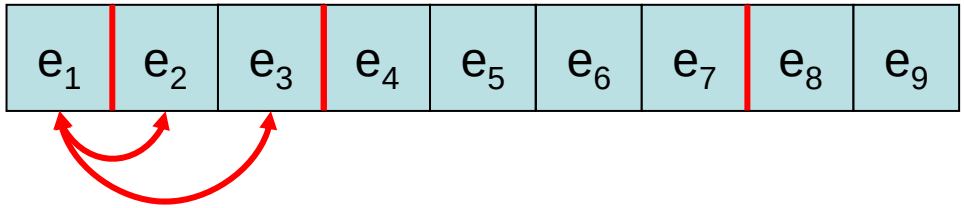
Insert Operation - Korrektheit



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

 : Knoten, die Invariante eventuell verletzen

Binärer Heap



deleteMin:

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

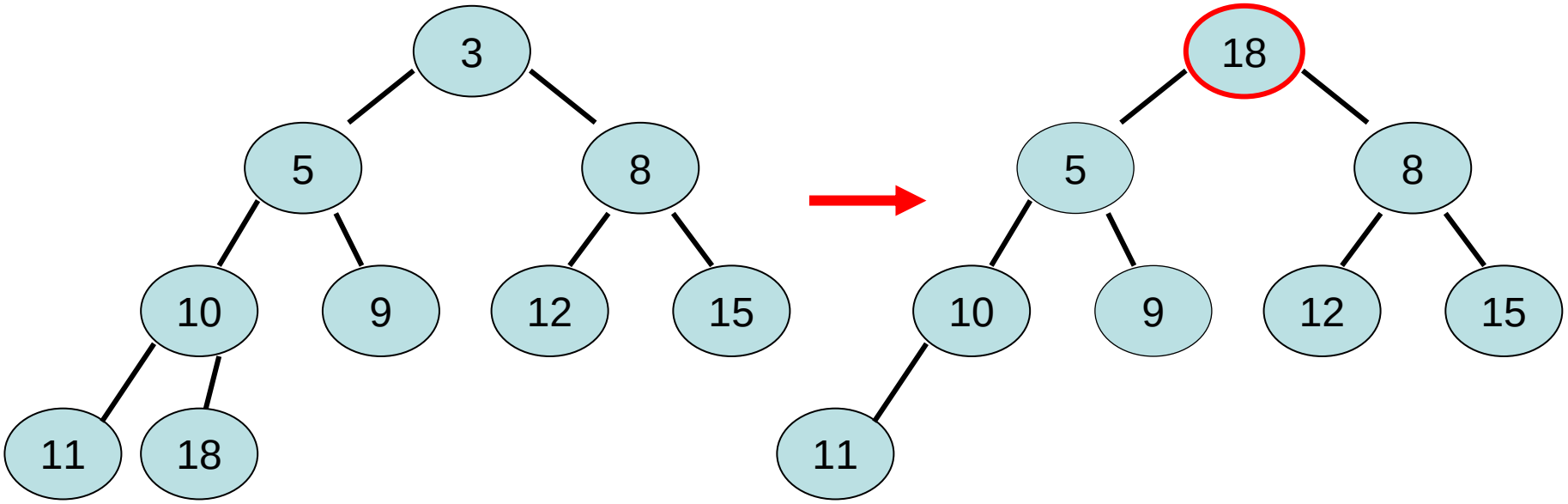
Binärer Heap

Laufzeit: $O(\log n)$

```
Function deleteMin(): Element
    e:=H[1]; H[1]:=H[n]; n:=n-1
    siftDown(1)
    return e

Procedure siftDown(i: Integer)
    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 return  // Heap-Inv gilt
        H[i] ↔ H[m]; i:=m
```

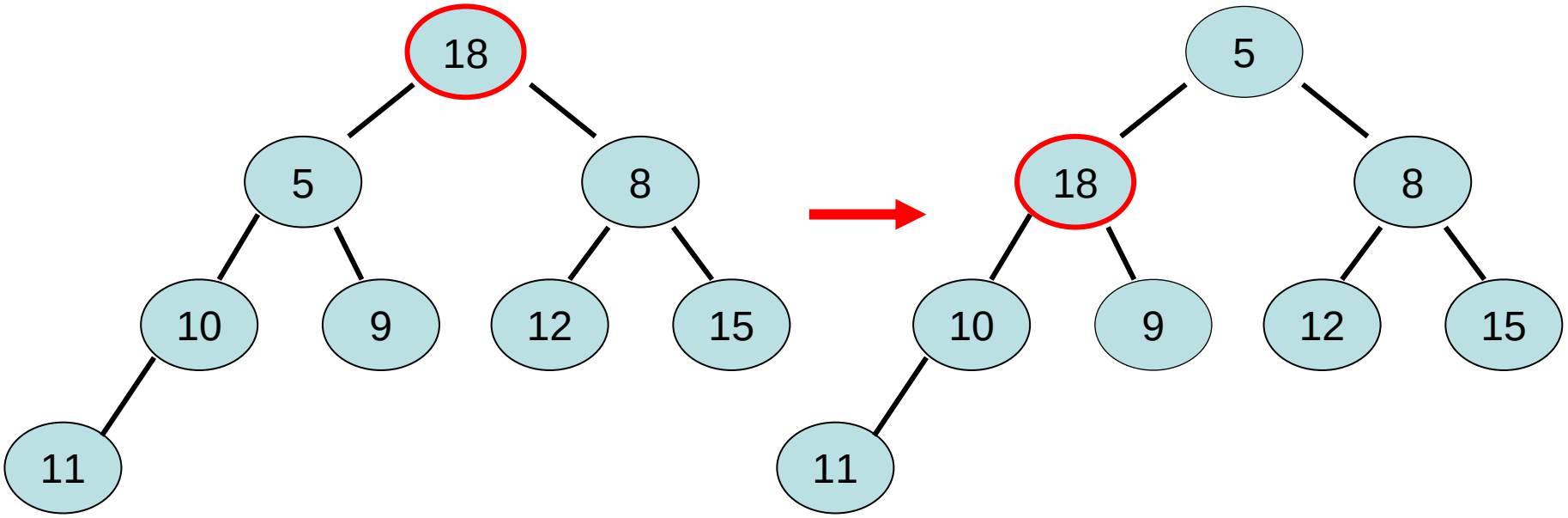
deleteMin Operation - Korrektheit



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

: Knoten, die Invariante eventuell verletzen

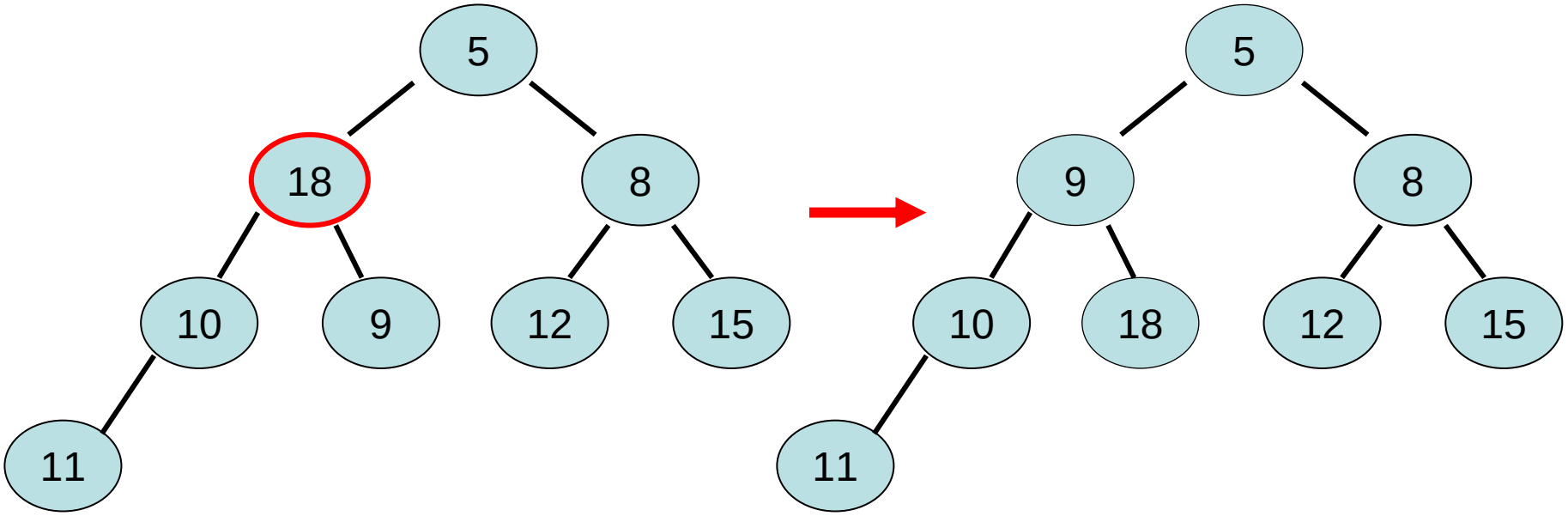
deleteMin Operation - Korrektheit



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

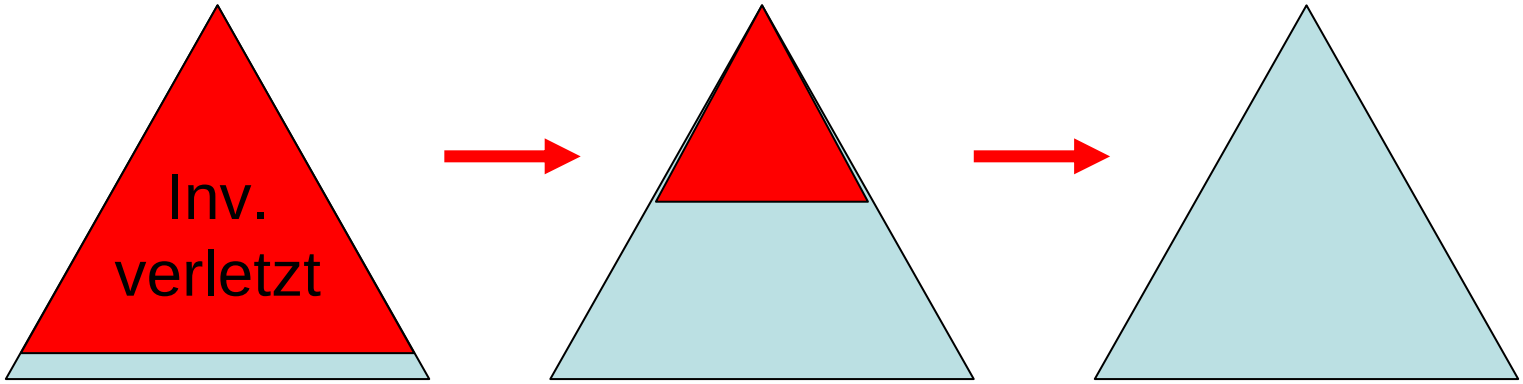
Binärer Heap

Build($\{e_1,\ldots,e_n\}$):

- Naive Implementierung: über n insert(e)-Operationen. Laufzeit $O(n \log n)$
- Bessere Implementierung:
  Setze $H[i]:=e_i$ für alle i. Rufe siftDown(i) für $i=\lfloor n/2 \rfloor$ runter bis 1 auf.
  Aufwand (mit $k=\lceil \log n \rceil$):
  $O(\sum_{1<=l<k} 2^l (k-l)) = O(2^k \sum_{j>=1} j/2^j) = O(n)$

Binärer Heap

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



Invariante: $\forall\ j>i$: $H[j]$ min für Teilbaum von $H[j]$

Binärer Heap

Laufzeiten:

- Build($\{e_1,\ldots,e_n\}$): O(n)
- Insert(e): O(log n)
- Min: O(1)
- deleteMin: O(log n)

Erweiterte Priority Queue

Zusätzliche Operationen:

- M.delete(e: Element): $M:=M\backslash\{e\}$
- M.decreaseKey(e:Element, Δ): key(e):=key(e)-Δ
- M.merge(M´): $M:=M \cup M´$

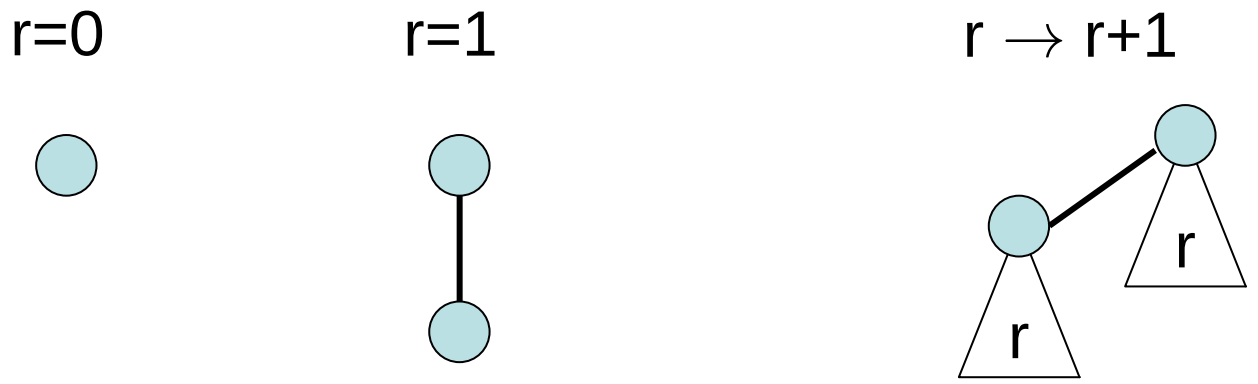
Delete und decreaseKey in Zeit $O(\log n)$ in Heap (wenn Position von e bekannt), aber merge ist teuer ($\Theta(n)$ Zeit)!

Binomial-Heap

Binomial-Heap basiert auf Binomial-Bäumen

Binomial-Baum muss erfüllen:

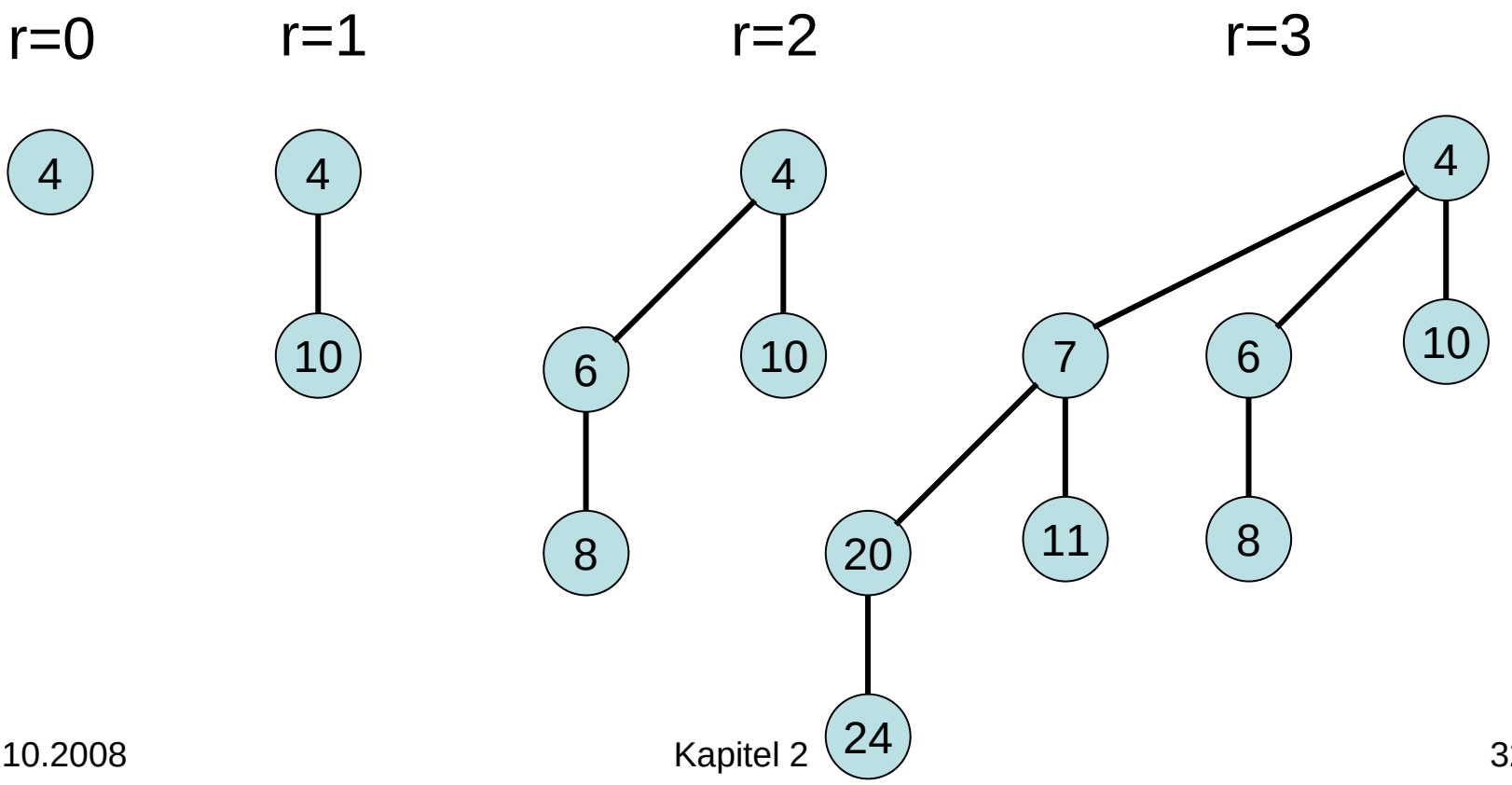
- Form-Invariante (r: Rang):



- Heap-Invariante (key(Vater)<=key(Kinder))

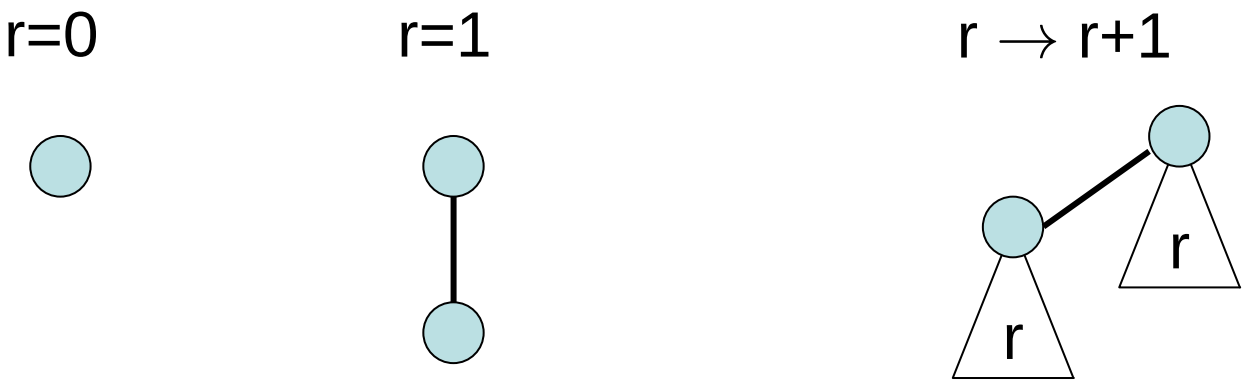
Binomial-Heap

Beispiel für korrekte Binomial-Bäume:



Binomial-Heap

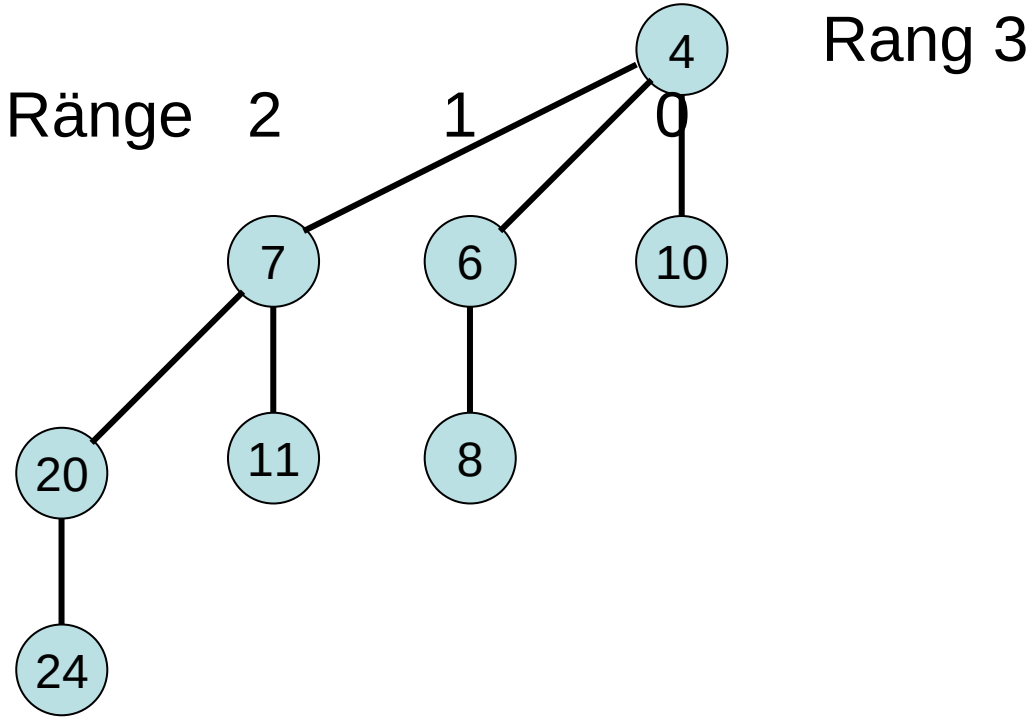
Eigenschaften von Binomial-Bäumen:



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

Binomial-Heap

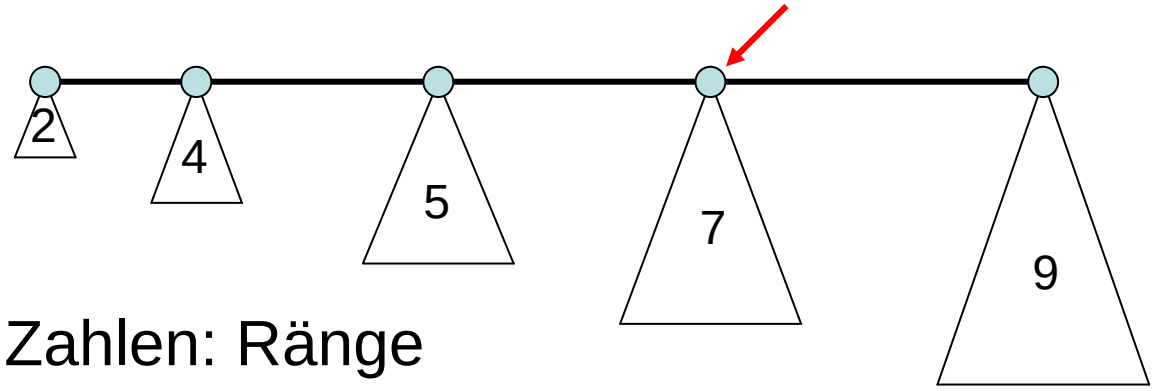
Beispiel für Zerfall in Binomial-Bäume mit Rang 0 bis r-1



Binomial-Heap

Binomial-Heap:

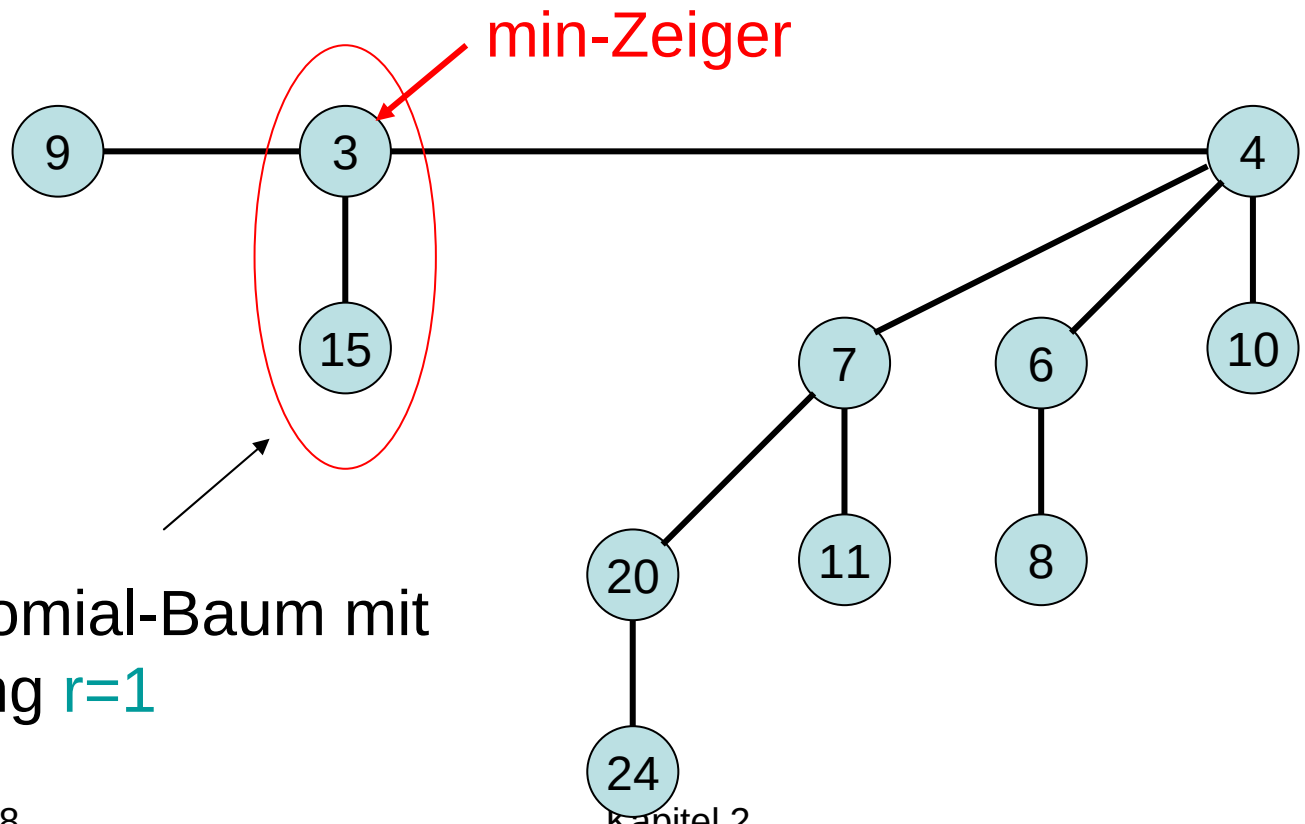
- verkettete Liste von Binomial-Bäumen
- Pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem key



Zahlen: Ränge

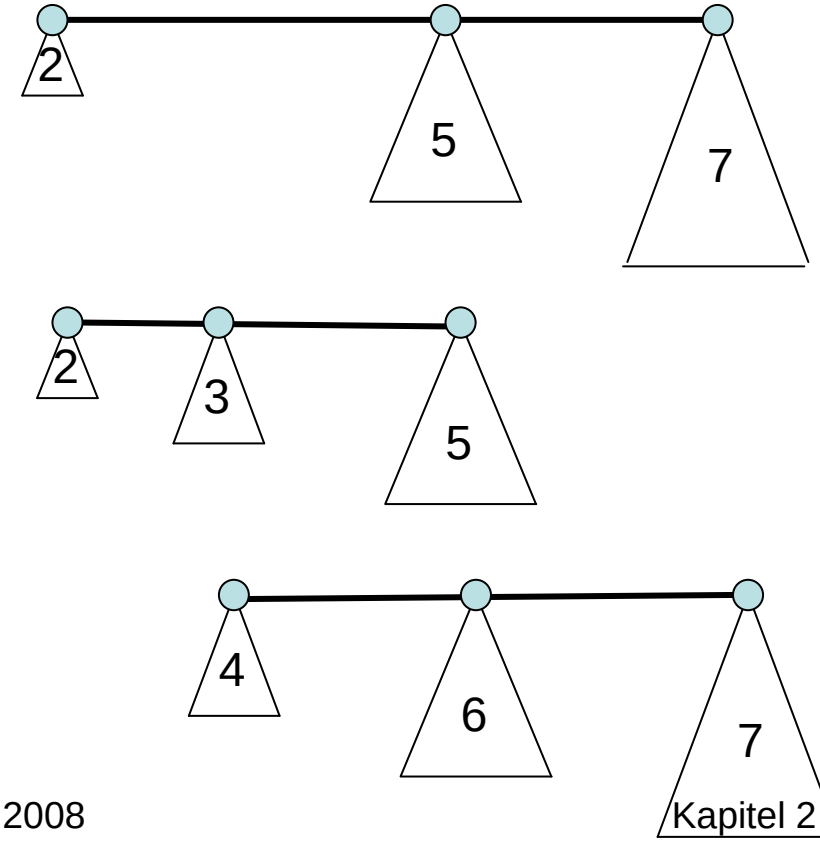
Binomial-Heap

Beispiel eines korrekten Binomial-Heaps:



Binomial-Heap

Merge von Binomial-Heaps H_1 und H_2:



H_1

H_2

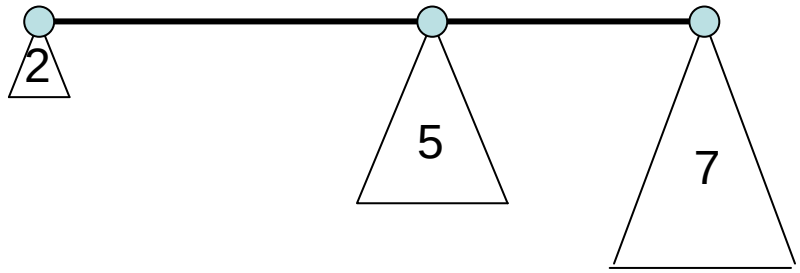
wie Binäraddition

10100100

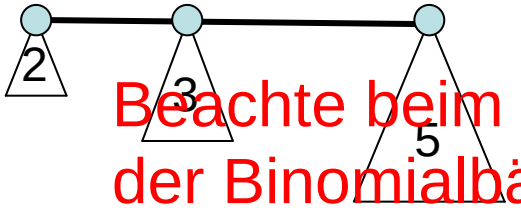
\+ 101100

11010000

Beispiel einer Merge-Operation

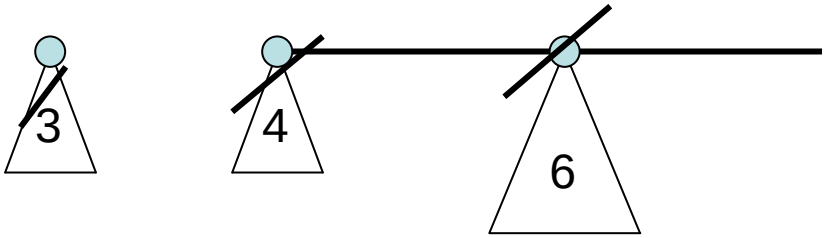


H_1



H_2

Beachte beim Mergen der Binomialbäume die Heap-Eigenschaft!



Ergebnis-Heap

Zahlen geben die Ränge an

Binomial-Heap

Aufwand für Merge-Operation: $O(\log n)$

B_i: Binomial-Baum mit Rang i

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

Binomial-Heap

- decreaseKey(e,Δ): siftUp-Operation in Binomial-Baum von e und aktualisierte min-Zeiger. Zeit O(log n)

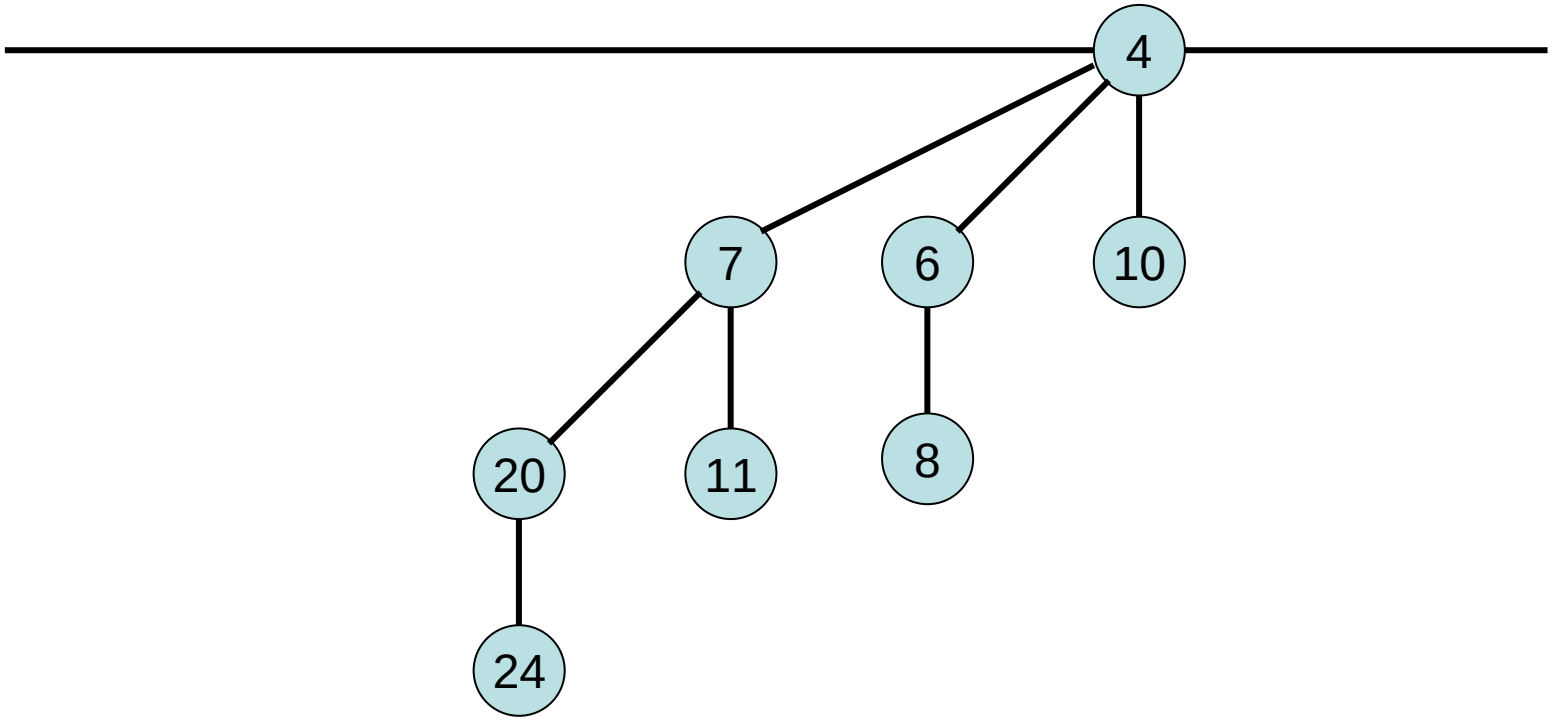
- delete(e): (min-Zeiger zeigt nicht auf e) setze key(e):= $-\infty$ und wende siftUp-Operation auf e an, bis e in der Wurzel; dann weiter wie bei deleteMin. Zeit O(log n)

Fibonacci-Heap

- Baut auch 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 Listen
- Lazy delete: erzeugt unvollständige Binomial-Bäume

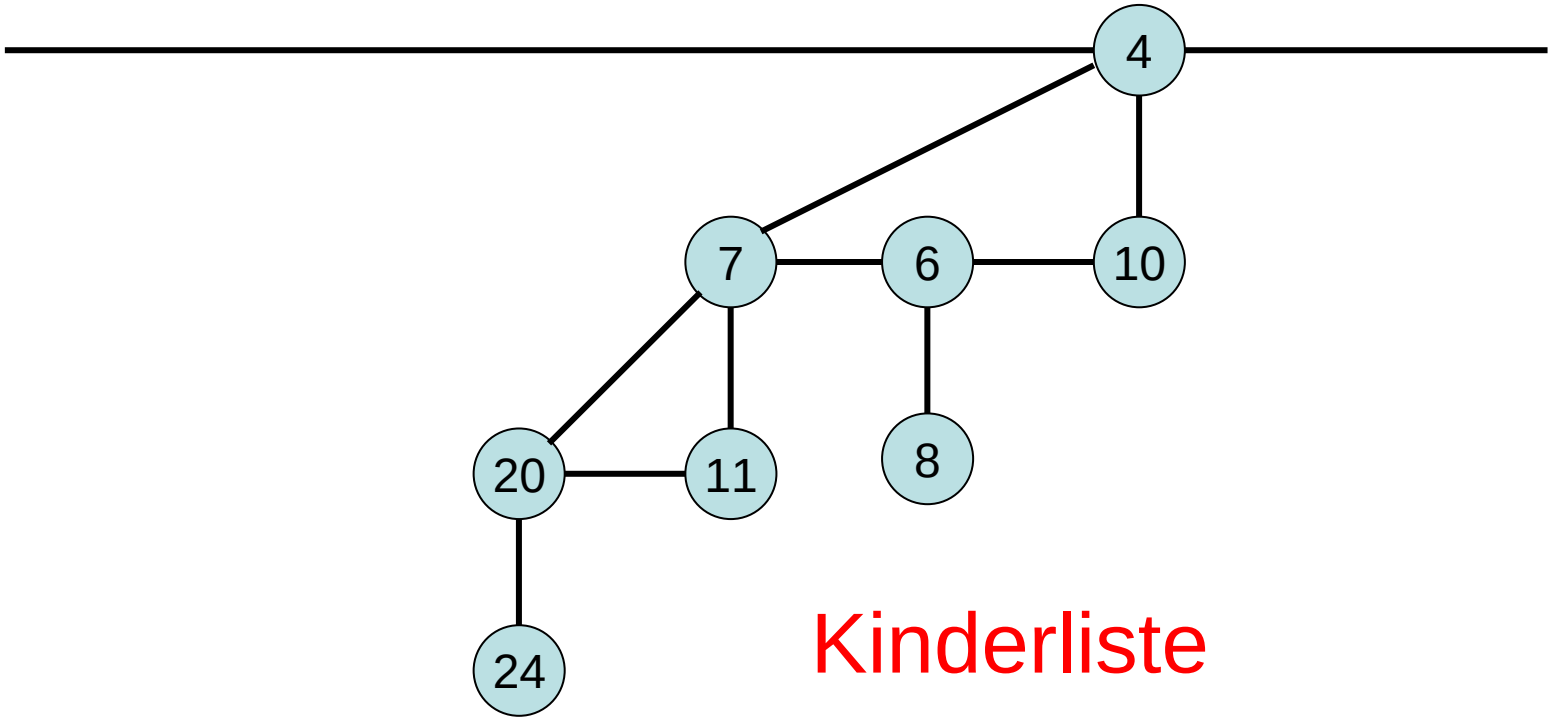
Fibonacci-Heap

Baum in Binomial-Heap:



Fibonacci-Heap

Baum in Fibonacci-Heap:



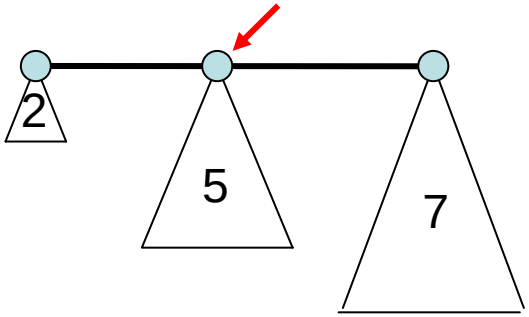
Fibonacci-Heap

Jeder Knoten merkt sich:

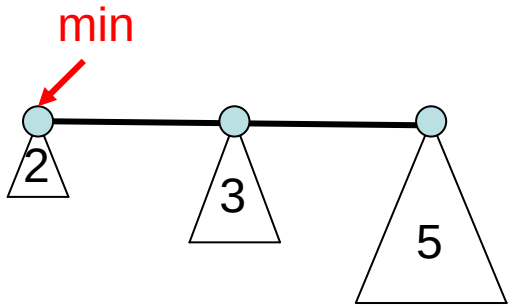
- im Knoten gespeichertes Element
- Vater
- Liste der Kinder (mit Start- und Endpunkt)
- Rang (Anzahl Kinder)

Fibonacci-Heap

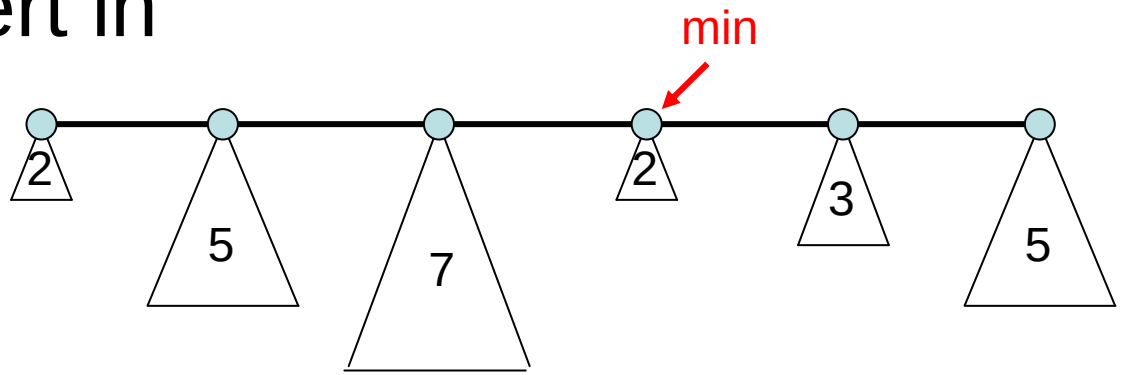
Lazy merge von



&

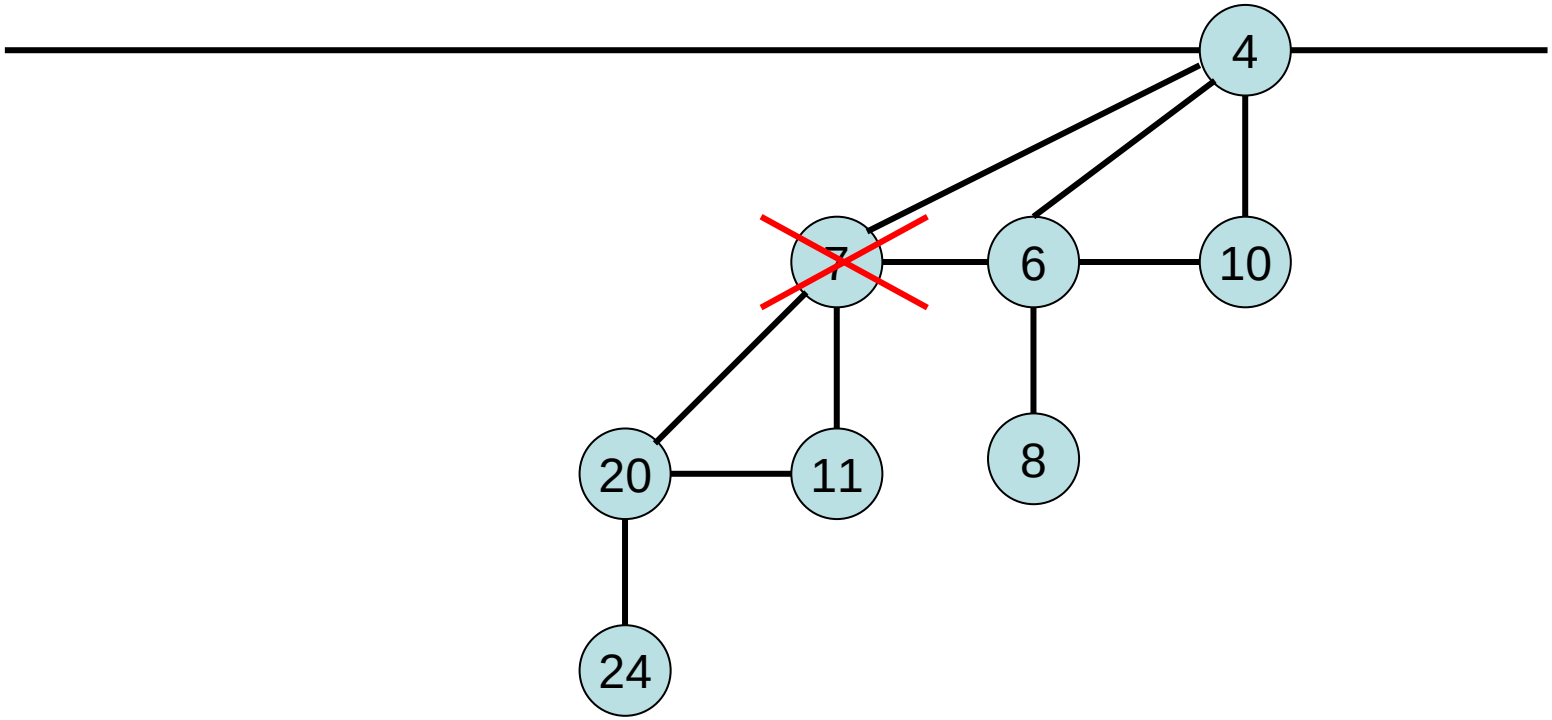


resultiert in



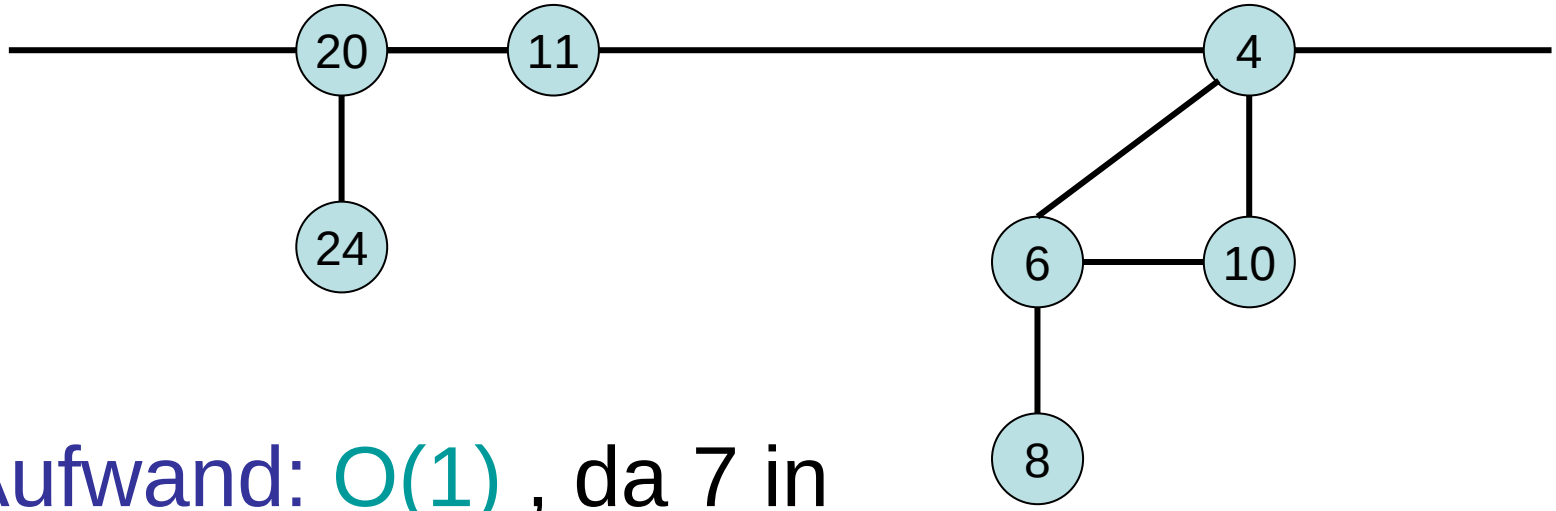
Fibonacci-Heap

Lazy delete:



Fibonacci-Heap

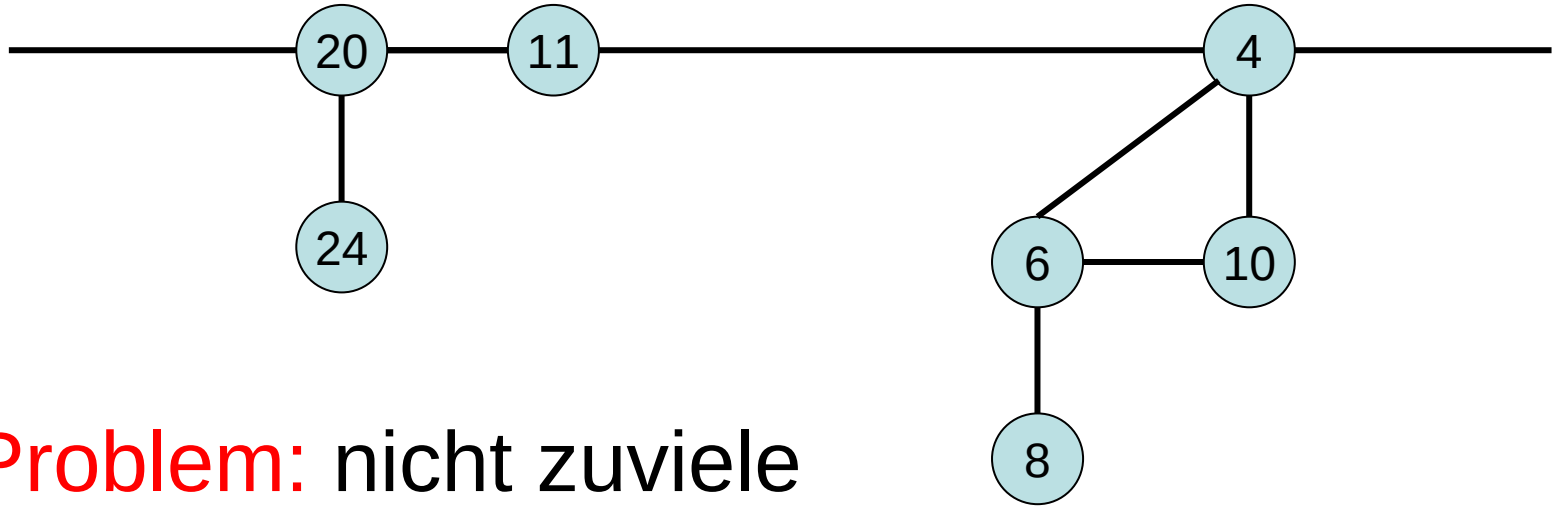
Lazy delete:



Aufwand: O(1) , da 7 in O(1) Zeit entfernbar und Kinderliste von 7 in O(1) Zeit in Wurzelliste integrierbar

Fibonacci-Heap

Lazy delete:



Problem: nicht zuviele Knoten raus aus Baum

$\rightarrow$ wird mit binären Markern geprüft

Fibonacci-Heap

Operationen:

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

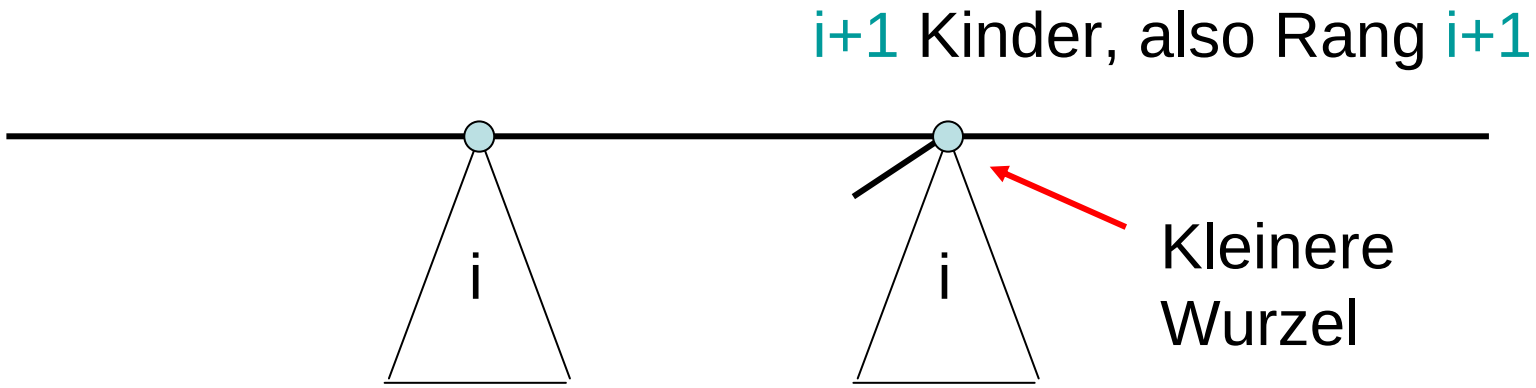
Fibonacci-Heap

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

Algorithmus deleteMin()
  entferne x aus Wurzelliste
  konkateniere Kinderliste von x mit Wurzelliste
  while >=2 Bäume mit gleichem Rang i do
    merge Bäume zu Baum mit Rang i+1
    (wie bei zwei Binomial-Bäumen)
  aktualisiere den min-Pointer

Fibonacci-Heap

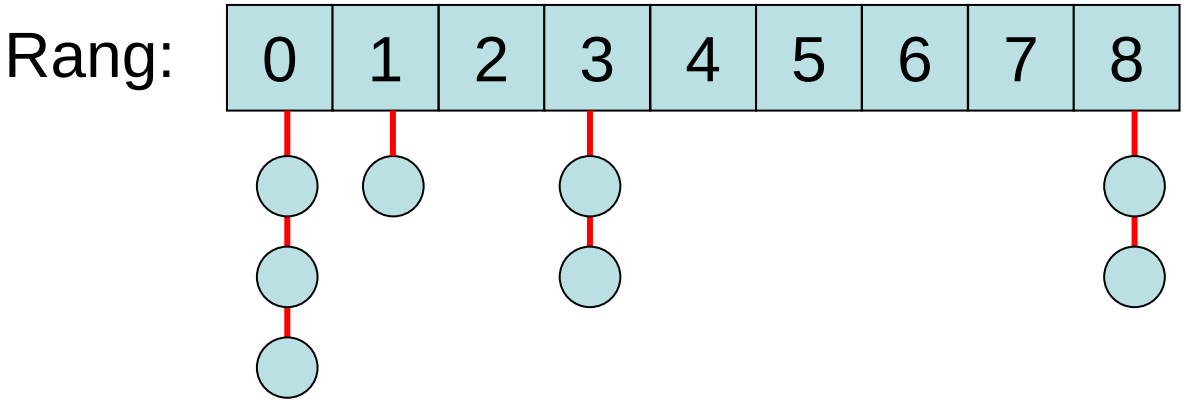
Verschmelzung zweier Bäume mit Rang i (d.h. Wurzel hat i Kinder):



Fibonacci-Heap

Effiziente Findung von Wurzeln mit gleichem Rang:

- Scanne vor while-Schleife alle Wurzeln und speichere diese nach Rängen in Feld:



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

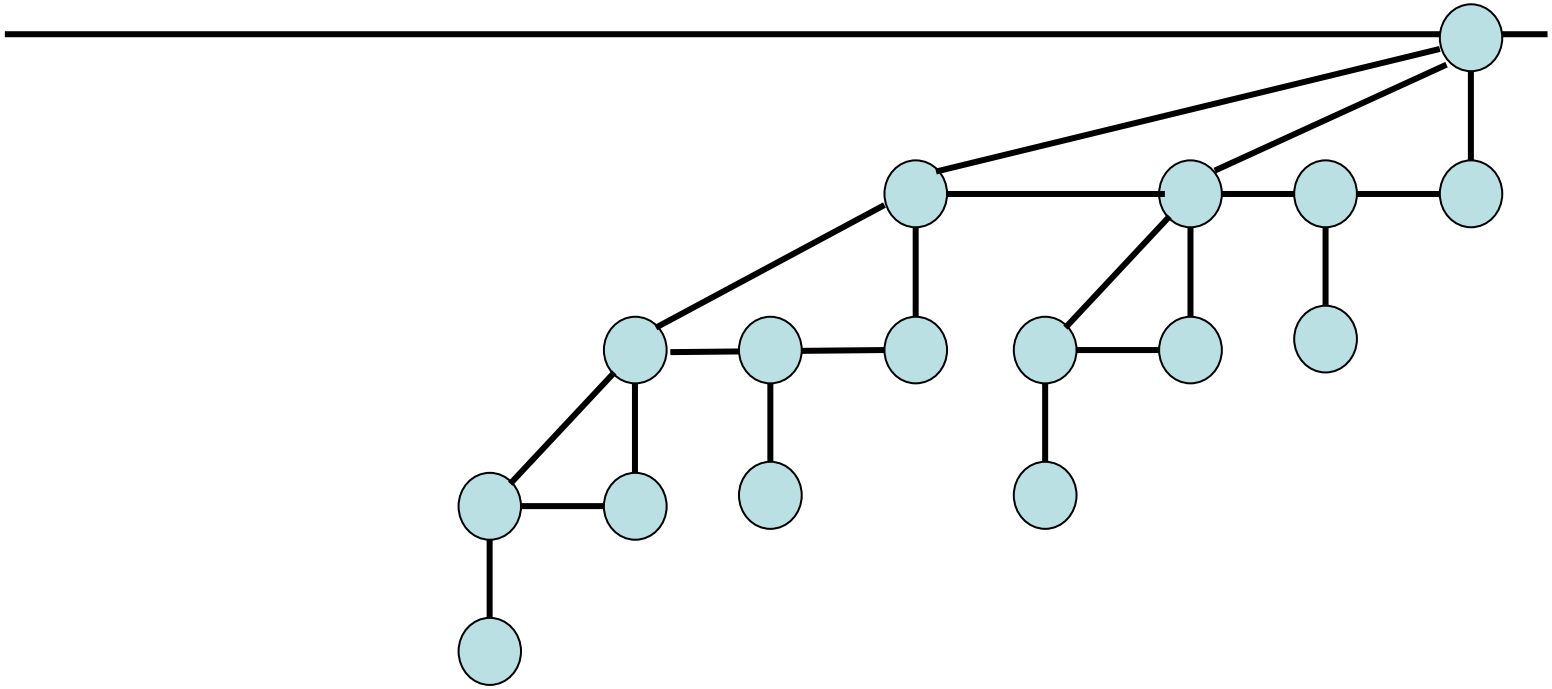
Fibonacci-Heap

Sei P(x) Vater von Knoten x. Wenn x neu eingefügt wird, ist Mark(x)=0. (Mark(x) speichert, wieviele Kinder schon weg.)

```
Algorithmus delete(x):
    if x ist min-Wurzel then deleteMin()
    sonst
      lösche x, füge Kinder von x in Wurzelliste ein
      if P(x)=NULL then return      // x ist Wurzel
      while true do
        x:=P(x)
        if P(x)=NULL then return     // x ist Wurzel
        if Mark(x)=0 then Mark(x):=1; return
        else       // Mark(x)=1, also schon Kind weg
            hänge x samt Unterbaum in Wurzelliste
            Mark(x):=0  // Wurzeln benötigen kein Mark
```

Fibonacci-Heap

Beispiel für delete(x): (● : Mark=1)



Fibonacci-Heap

```
Algorithmus decreaseKey(x,Δ):
    füge x samt Unterbaum in Wurzelliste ein
    key(x):=key(x)-Δ
    aktualisiere min-Pointer
   if P(x)=NULL then return      // war x Wurzel?
    while true do
      x:=P(x)
      if P(x)=NULL then return     // x ist Wurzel
      if Mark(x)=0 then Mark(x):=1; return
      else       // Mark(x)=1
          hänge x samt Unterbaum in Wurzelliste
          Mark(x):=0
```

Fibonacci-Heap

Zeitaufwand:

- deleteMin():
  O(max. Rang + #Baumverschmelzungen)
- delete(x), decreaseKey(x,Δ):
  O(1 + #kaskadierende Schnitte)
  d.h. #umgehängter markierter Knoten

Wir werden sehen: Zeitaufwand kann O(n) in beiden Fällen sein, aber im Durchschnitt viel günstiger.

Amortisierte Analyse

Betrachte Folge von n Operationen auf Anfangs leerem Fibonacci-Heap.

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

Amortisierte Analyse

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

$$s_0 \xrightarrow{Op_1} s_1 \xrightarrow{Op_2} s_2 \xrightarrow{Op_3} \ldots \xrightarrow{Op_n} s_n$$

- 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, heißt Familie amortisierter Zeitschranken falls für jede Sequenz F von Operationen gilt

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

für eine Konstante c unabhängig von F

Amortisierte Analyse

Theorem 2.1: Sei S der Zustandsraum einer Datenstruktur, sei s_0 der Anfangszustand und sei $\phi: S \rightarrow \mathbb{R}_{>=0}$ eine nichtnegative Funktion. Für eine Operation X und einen Zustand s mit $s \xrightarrow{X} s'$ definiere

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

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

Amortisierte Analyse

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

Beweis:

$$\sum_{i=1}^{n} A_{Op_i}(s_{i-1}) = \sum_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1}) + T_{Op_i}(s_{i-1})]$$

$$= T(F) + \sum_{i=1}^{n} [\phi(s_i) - \phi(s_{i-1})]$$

$$= T(F) + \phi(s_n) - \phi(s_0)$$

$$\Rightarrow T(F) = \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0) - \phi(s_n)$$

$$<= \sum_{i=1}^{n} A_{Op_i}(s_{i-1}) + \phi(s_0)$$ konstant

Amortisierte Analyse

$\phi: S \rightarrow IR_{>=0}$ wird auch Potential genannt.

Für Fibonacci-Heaps verwenden wir das Potential

bal(s):= #Bäume + 2·#markierte Knoten in Zustand s

Fibonacci-Heap

Lemma 2.2: Sei x ein Knoten im Fibonacci-Heap mit Rang(x)=k. Seien die Kinder von x sortiert in der Reihenfolge ihres Anfügens an x. Dann ist der Rang des i-ten Kindes >=i-2.

Beweis:

- Beim Einfügen des i-ten Kindes ist 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, d.h. sein Rang ist >=i-2.

Fibonacci-Heap

Theorem 2.3: Sei x ein Knoten im Fibonacci-Heap mit Rang(x)=k. Dann enthält der Baum mit Wurzel x mindestens F_{k+2} Elemente, wobei F_k die k-te Fibonacci-Zahl ist.

Definition der Fibonacci-Zahlen:

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

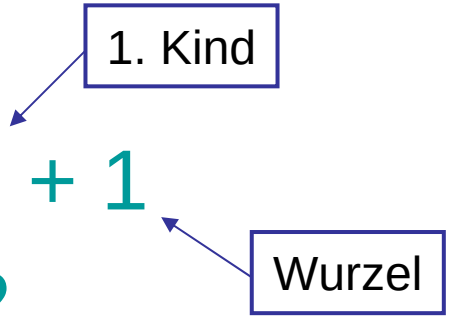
Daraus folgt, dass $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.

Fibonacci-Heap

Beweis von Theorem 2.3:

- Sei f_k die minimale Anzahl von Elementen in einem Baum mit Rang k.
- Aus Lemma 2.2 folgt:

$$f_k >= f_{k-2}+f_{k-3}+\ldots+f_0 + 1 + 1$$



- Weiterhin ist $f_0=1$ und $f_1=2$
- Also folgt nach den Fibonacci-Zahlen:

$$f_k >= F_{k+2}$$

Fibonacci-Heap

- Es ist bekannt, dass $F_{k+2} > \Phi^{k+2}$ ist für

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

- D.h. ein Baum mit Rang k im Fibonacci-Heap hat mindestens $1{,}61^{k+2}$ Knoten.
- Ein Fibonacci-Heap aus n Elementen hat also Bäume vom Rang höchstens $O(\log n)$ (wie bei Binomial-Heap)

Fibonacci-Heap

- t_i: Zeit für Operation i
- bal_i: Wert von $bal(s)$ nach Operation i
  ($bal(s)$ = #Bäume + 2·#markierte Knoten)
- a_i: amortisierter Aufwand für Operation i
  $a_i = t_i + \Delta bal_i$ mit $\Delta bal_i = bal_i - bal_{i-1}$

Amortisierte Kosten der Operationen:

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

Fibonacci-Heap

Theorem 2.4: Die amortisierten Kosten von deleteMin() sind O(log n).

Beweis:

- Einfügen der Kinder von x in Wurzelliste:
  Δbal = Rang(x) - 1
- Jeder Merge-Schritt verkleinert #Bäume um 1:
  Δbal = - #Merge-Schritte
- Wegen Theorem 2.3 (Rang max. O(log n)) gilt:
  #Merge-Schritte = #Bäume – O(log n)
- Insgesamt: Δbal_i = Rang(x) - #Bäume + O(log n)
- Laufzeit (in geeigneten Zeiteinheiten):
  t_i = #Bäume + O(log n)
- Amortisierte Laufzeit:
  $a_i = t_i + \Delta bal_i$ = O(log n)

Fibonacci-Heap

Theorem 2.5: Die amortisierten Kosten von delete(x) sind O(log n).

Beweis: (x ist kein min-Element – sonst wie Theorem 2.4)

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

Fibonacci-Heap

Theorem 2.5: Die amortisierten Kosten von delete(x) sind $O(\log n)$.

Beweis (Fortsetzung):

- Insgesamt:
  Δbal_i = Rang(x) - #kaskadierende Schnitte + $O(1)$
  = $O(\log n)$ - #kaskadierende Schnitte
  wegen Theorem 2.3
- Laufzeit (in geeigneten Zeiteinheiten):
  $t_i = O(1)$ + #kaskadierende Schnitte
- Amortisierte Laufzeit:
  $a_i = t_i + \Delta bal_i = O(\log n)$

Fibonacci-Heap

Theorem 2.6: Die amortisierten Kosten von decreaseKey(x,Δ) sind O(1).

Beweis:

- Jeder kask. Schnitt erhöht die Anzahl Bäume um 1: Δbal = #kaskadierende Schnitte
- Jeder kask. Schnitt entfernt eine Markierung (bis auf x): $\Delta bal <= -2 \cdot$(#kaskadierende Schnitte-1)
- Der letzte Schnitt erzeugt evtl. eine Markierung: $\Delta bal \in \{0,2\}$
- Insgesamt: Δbal_i = - #kask. Schnitte + O(1)
- Laufzeit: t_i = #kask. Schnitte + O(1)
- Amortisierte Laufzeit: $a_i = t_i + \Delta bal_i = O(1)$

Laufzeitvergleich

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

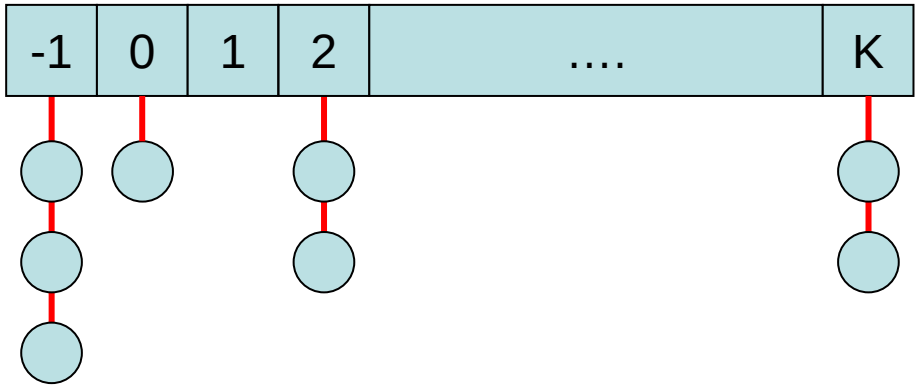
Radix-Heap

Annahmen:

- alle Elemente haben eine maximale Differenz von C zueinander
- Insert(e) fügt nur Elemente e ein mit $key(e) >= k_{min}$ (k_{min}: min. Schlüssel)

Radix-Heap

Array B mit Listen B[-1] bis B[K], $K=1+\lfloor \log C \rfloor$.



Regel: Element e in $B[\min(msd(k_{min},key(e)),K)]$

$msd(k_{min},key(e))$: höchstes Bit, in dem sich Binärdarstellungen von k_{min} und key(e) unterscheiden (-1: kein Unterschied)

Radix-Heap

Beispiel für $msd(k_{min},k)$:

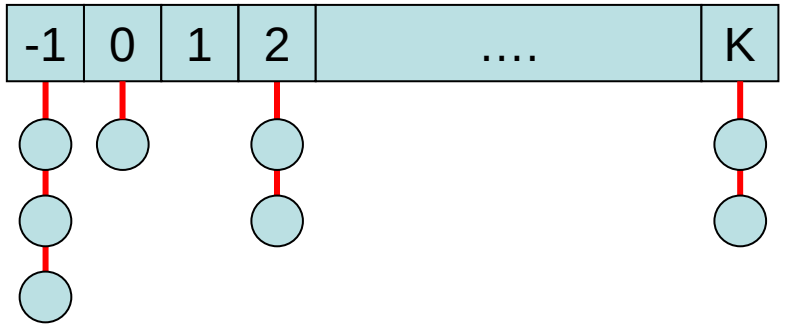
- sei $k_{min}=17$ oder binär 10001
- $k=17$: $msd(k_{min},k)=-1$
- $k=18$: binär 10010, also $msd(k_{min},k)=1$
- $k=21$: binär 10101, also $msd(k_{min},k)=2$
- $k=52$: binär 110100, also $msd(k_{min},k)=5$

Berechnung von msd für $a \neq b$:

$$msd(a,b)=\lfloor \log(a \oplus b) \rfloor$$

Zeit $O(1)$ (mit entspr. Maschinenbefehlen)

Radix-Heap



min():

- gib k_{min} in B[-1] zurück

Laufzeit: O(1)

Radix-Heap

insert(e): (key(e)>=k_{min})

- i:=min{msd(k_{min},key(e)), K}
- speichere e in B[i]

Laufzeit: O(1)

delete(e): (key(e)>k_{min})

- lösche e aus seiner Liste B[j] raus

Laufzeit: O(1)

decreaseKey(x,Δ): (key(e) - Δ > k_{min})

- rufe delete(e) und insert(e) mit key(e):=key(e) - Δ auf

Laufzeit: O(1)

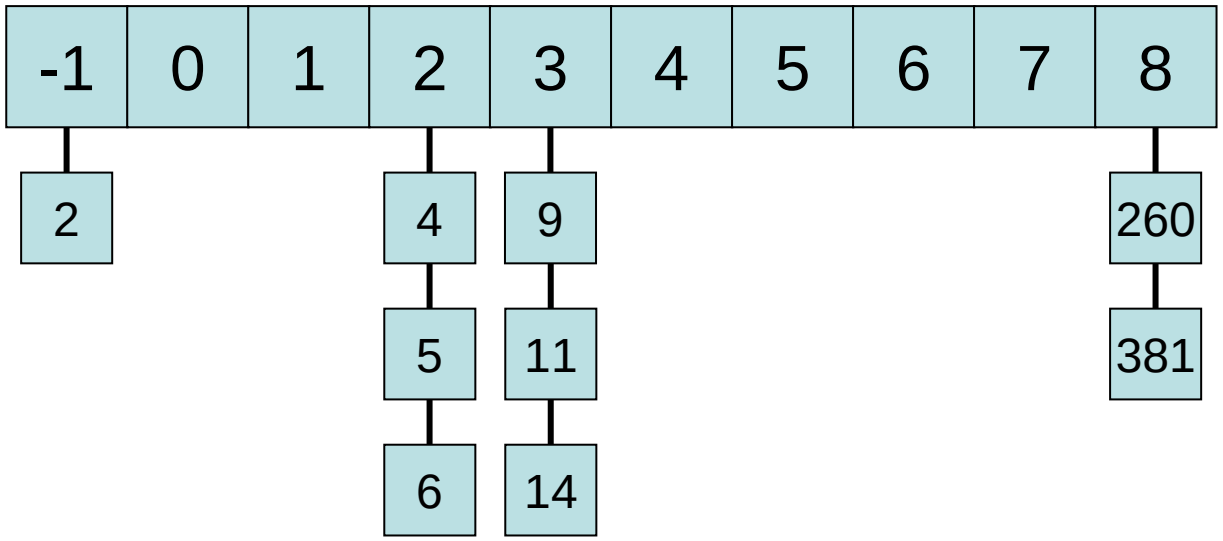
Radix-Heap

deleteMin():

- Falls B[-1] besetzt, entnehme ein e aus B[-1] (sonst kein Element mehr in Heap und fertig)
- finde minimales i, so dass $B[i] \neq \emptyset$ (falls kein solches i oder i=-1, dann fertig)
- bestimme k_{min} in B[i] (in O(1) Zeit möglich, falls z.B. k_{min}-Werte in extra Feld min[-1..K] gehalten)
- verteile Knoten in B[i] über B[-1],…,B[i-1] gemäß neuem k_{min}

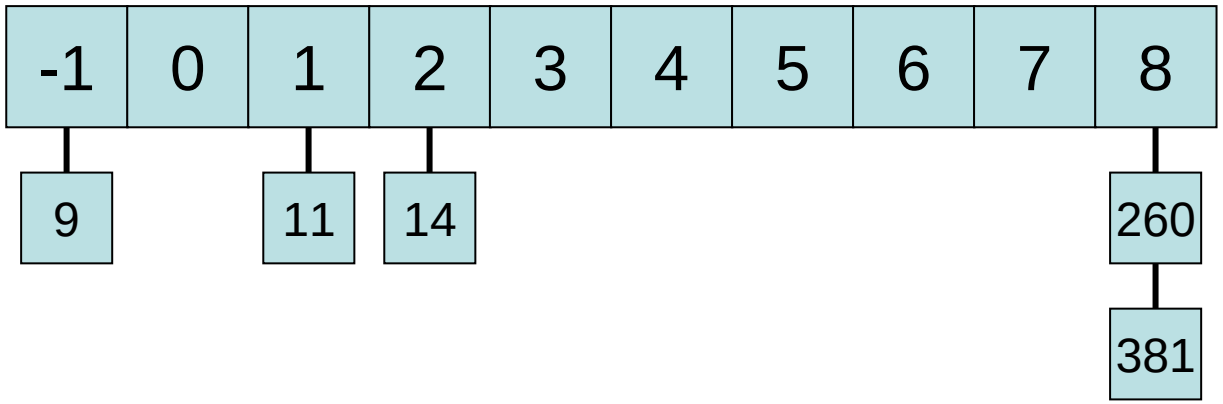
Entscheidend: für alle e in B[j], j>i, gilt nach wie vor $msd(k_{min},key(e))=j$, d.h. sie müssen nicht verschoben werden.

Radix-Heap



Wir betrachten Sequenz von deleteMin Operationen

Radix-Heap



Wir betrachten Sequenz von deleteMin Operationen

Radix-Heap

Lemma 2.7: Sei B[i] die minimale nichtleere Liste, i>0. Sei x_{min} der kleinste Schlüssel in B[i]. Dann ist $msd(x_{min},x)<i$ für alle Schlüssel x in B[i].

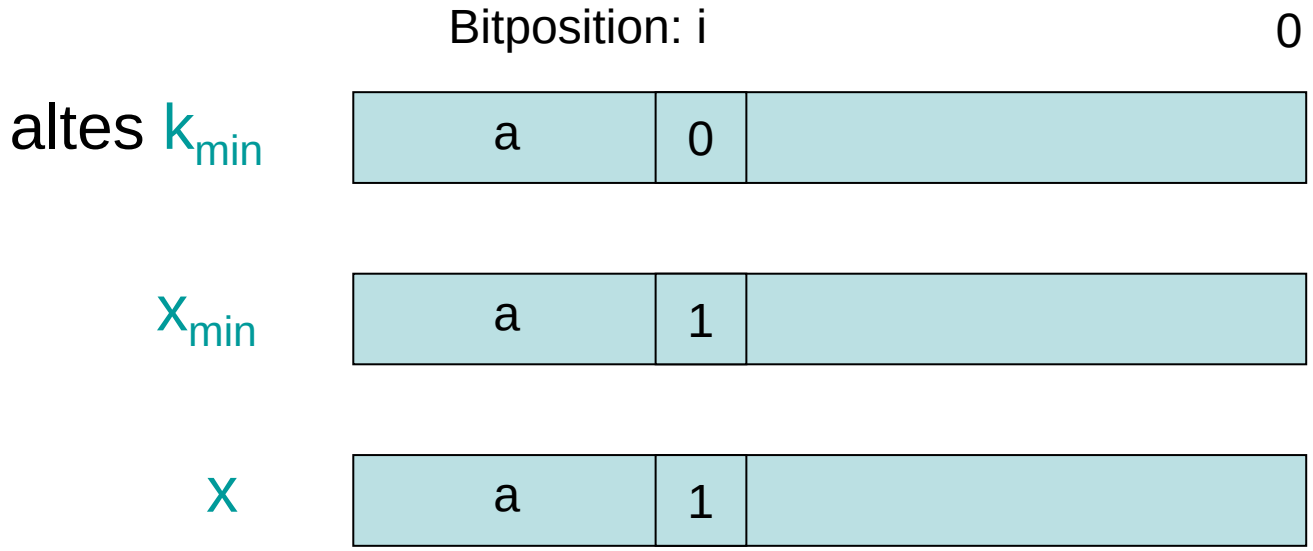
Alle Elemente in B[i] nach links

Beweis:

- $x=x_{min}$: klar
- $x \neq x_{min}$: wir unterscheiden zwei Fälle:
  1) i<K, 2) i=K

Radix-Heap

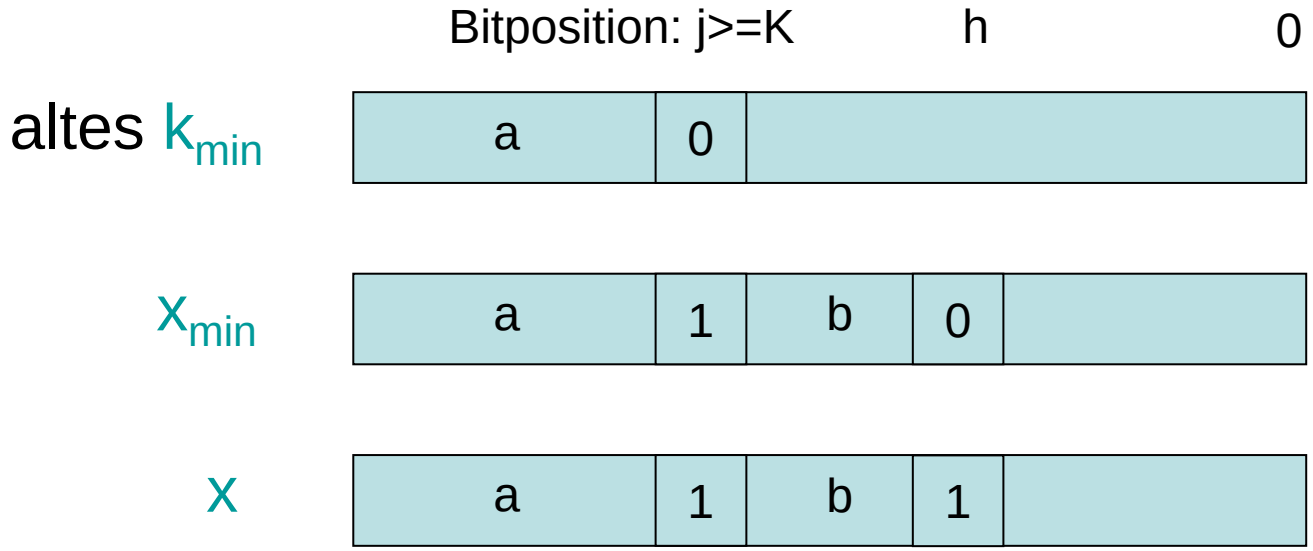
Fall i<K:



Also muss $msd(x_{min},x)<i$ sein.

Radix-Heap

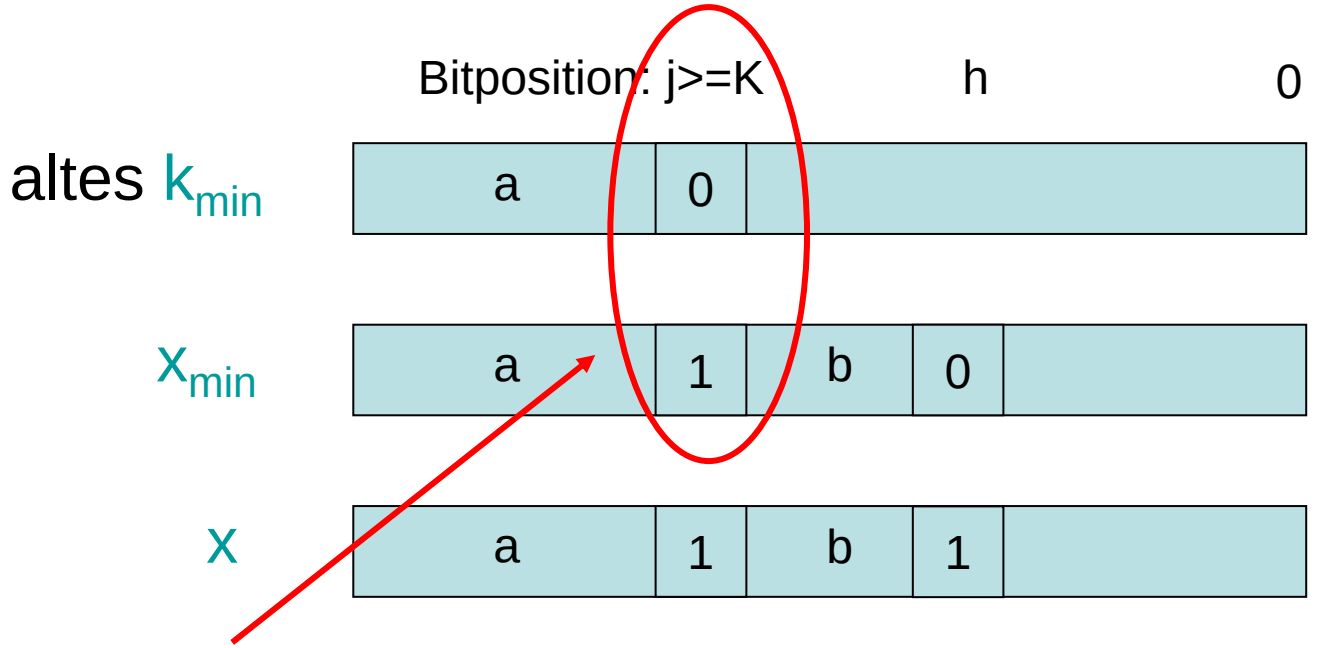
Fall i=K:



- Für alle x: $k_{min} < x_{min} \leq x \leq k_{min}+C$
- $j = msd(k_{min},x_{min})$, $h = msd(x_{min},x)$

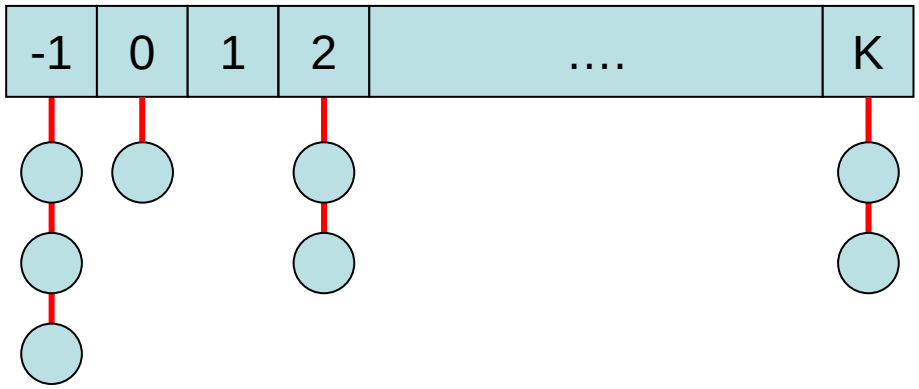
Radix-Heap

Fall i=K:



- Wegen $j \geq K$ hat Binärdarstellung von jedem übrigbleibenden x dasselbe Präfix $a1b$ bis runter zu Bitstelle $j' \leq K$, d.h. $h<K$

Radix-Heap



Konsequenz:

- jeder Knoten nur maximal K-mal verschoben in Radix Heap
- insert(): amortisierte Laufzeit O(log C).

Laufzeitvergleich

Laufzeit	Fibonacci-Heap	Radix-Heap
insert	O(1)	O(log C) amor.
min	O(1)	O(1)
deleteMin	O(log n) amor.	O(1) amor.
delete	O(log n) amor.	O(1)
merge	O(1)	???
decreaseKey	O(1)	O(1)

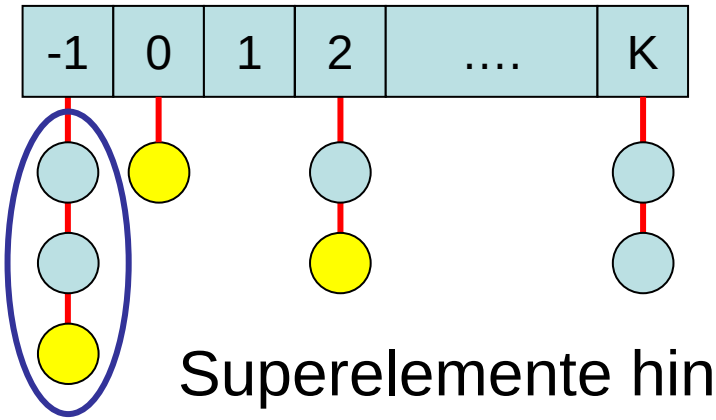
Erweiterter Radix-Heap

Annahmen:

- alle Elemente haben eine maximale Differenz von C zueinander
- Insert(e) fügt nur Elemente e ein mit key(e)>=k_{min} (k_{min}: min. Schlüssel)

Mit geringfügigen Ergänzungen machbar.

Erweiterter Radix-Heap



mind. ein normales Element

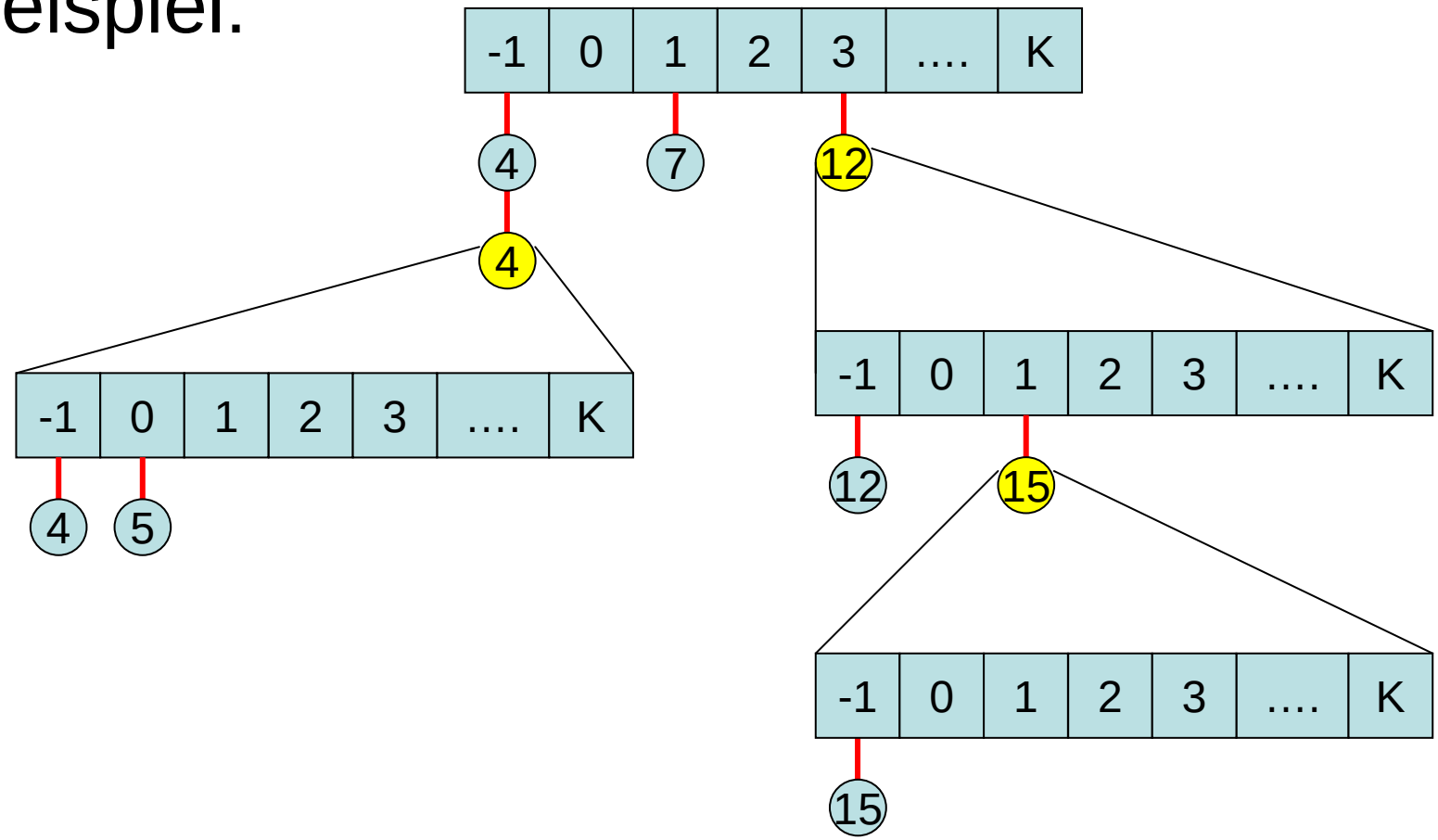
Superelemente hinten

● : “Superelement” e enthält Heap mit k_{min}=key(e), wobei k_{min} der kleinste Wert im Radix-Heap von e ist und $B_e[-1]$ mind. ein normales Element hat.

Superelemente können Superelemente enthalten.

Erweiterter Radix-Heap

Beispiel:



Erweiterter Radix-Heap

Merge von Radix-Heaps B und B':

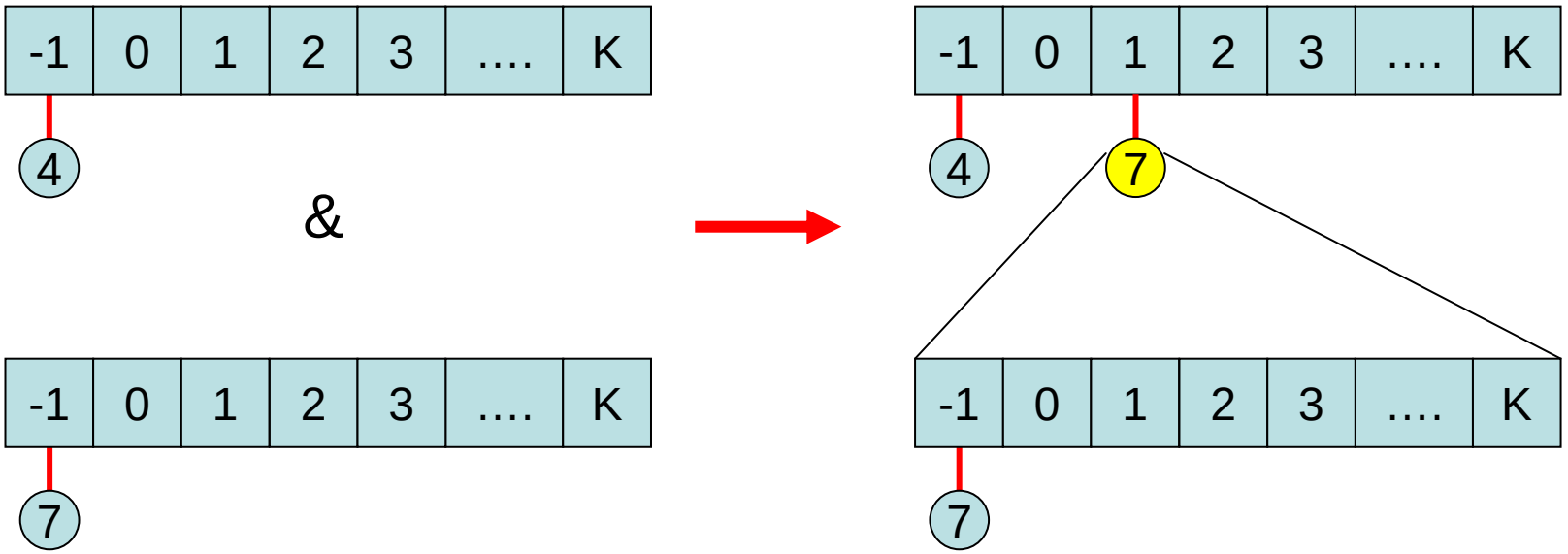
Annahme: $k_{min}(B) <= k_{min}(B')$
(Fall $k_{min}(B) > k_{min}(B')$ ist analog)

- hänge B' in ein Superelement e ein mit $key(e) = k_{min}(B')$
- führe insert(e) auf B aus

Laufzeit: O(1)

Erweiterter Radix-Heap

Beispiel für merge-Operation:



Erweiterter Radix-Heap

insert(e):

- key(e)>=k_{min}: wie bei Radix-Heap
- Sonst wie merge alten Heap mit neuem Heap mit Element e

Laufzeit: O(1)

min(): wie bei Radix-Heap

Laufzeit: O(1)

Erweiterter Radix-Heap

deleteMin():

- Entferne einfaches Element e aus B[-1]
  (B: Radix-Heap auf höchster Ebene)
- Falls B[-1] keine Elemente mehr hat, dann aktualisiere B wie im normalen Radix-Heap (d.h. löse kleinstes nichtleeres Bucket B[i] auf)
- Falls B[-1] keine einfachen Elemente hat, dann nimm erstes Superelement e' aus B[-1] und merge die Listen in e' mit B
  (damit wieder einfaches Element in B[-1])

Laufzeit: O(log C) + Aktualisierungsaufwand

Erweiterter Radix-Heap

delete(e):

Fall 1: key(e)>k_{min} für Heap von e:

- wie delete(e) im normalen Radix-Heap

Fall 2: key(e)=k_{min} für Heap von e:

- wie deleteMin() oben, aber auf Heap von e
- falls e in Superelement e' und e' danach leer, dann entferne e' aus übergeordnetem Heap B' (da einfaches Element in B'[-1], kein Problem!)

Laufzeit: O(log C) + Aktualisierungsaufwand

Erweiterter Radix-Heap

decreaseKey(e,Δ):

- führe delete(e) in Heap von e aus
- setze key(e):=key(e)-Δ
- führe insert(e) auf oberstem Radix-Heap aus

Laufzeit: O(log C) + Aktualisierungsaufwand

Laufzeitvergleich

Laufzeit	Radix-Heap	erw. Radix-Heap
insert	O(log C) amor.	O(log C) amor.
min	O(1)	O(1)
deleteMin	O(1) amor.	O(1) amor.
delete	O(1)	O(1) amor.
merge	???	O(log C) amor.
decreaseKey	O(1)	O(log C) amor.

Nächstes Kapitel

Thema: Suchstrukturen