Algorithmen und Datenstrukturen

Prioritätswarteschlangen mit binären Heaps

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (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

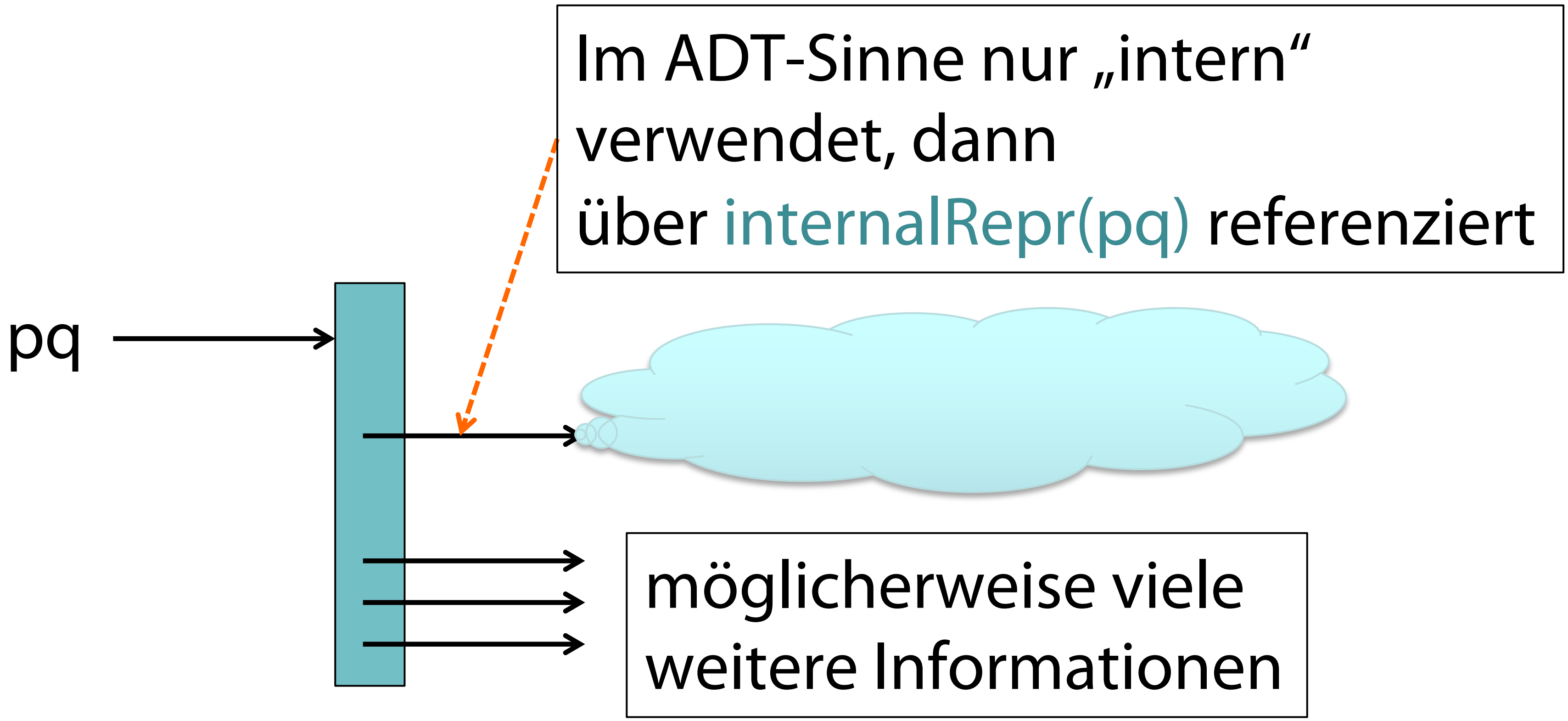
Geg.: $\{e_1,\ldots,e_n\}$ eine Menge von Elementen

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

Operationen:

- function build($\{e_1,\ldots,e_n\}$, key) liefert neue Warteschlange, in der die Elemente e_i nach key(e_i) priorisiert verwaltet werden
  - Funktion key wird von der erzeugten Warteschlange verwaltet
- 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, und pq wird verändert, wenn etwas gelöscht wird
- function mt?(pq) prüft, ob Warteschlange pq leer

Prioritätswarteschlangen als ADTs



Erweiterte Prioritätswarteschlangen

Zusätzliche Operationen:

- procedure delete(e, pq) löscht e aus pq, falls vorhanden, verändert ggf. pq
- procedure decreaseKey(e, pq, Δ): key(e) := key(e)-Δ, 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 im Feld)
  - min: $O(1)$
  - deleteMin: $O(n)$ (verschiebe Elemente im Feld)

Bessere Struktur als Liste oder Feld möglich?

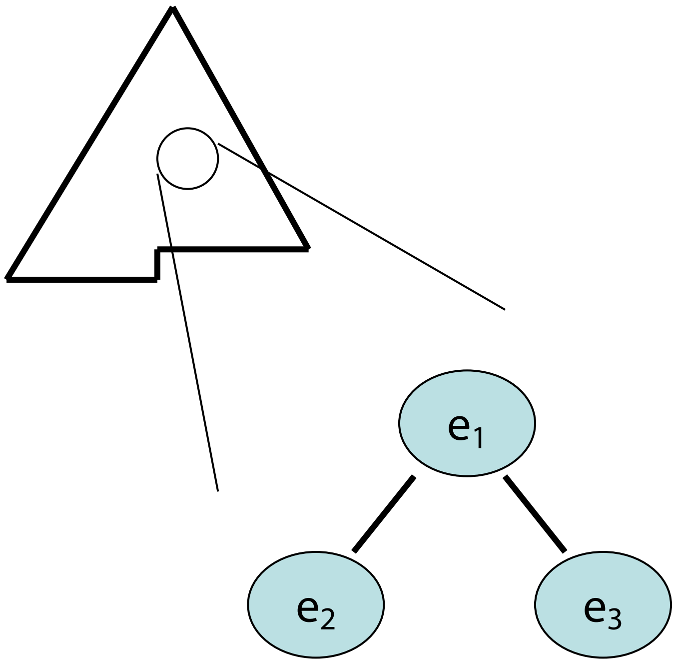
Binärer Heap (Wiederholung)

Idee: Verwende binären Baum statt Liste

Bewahre zwei Invarianten:

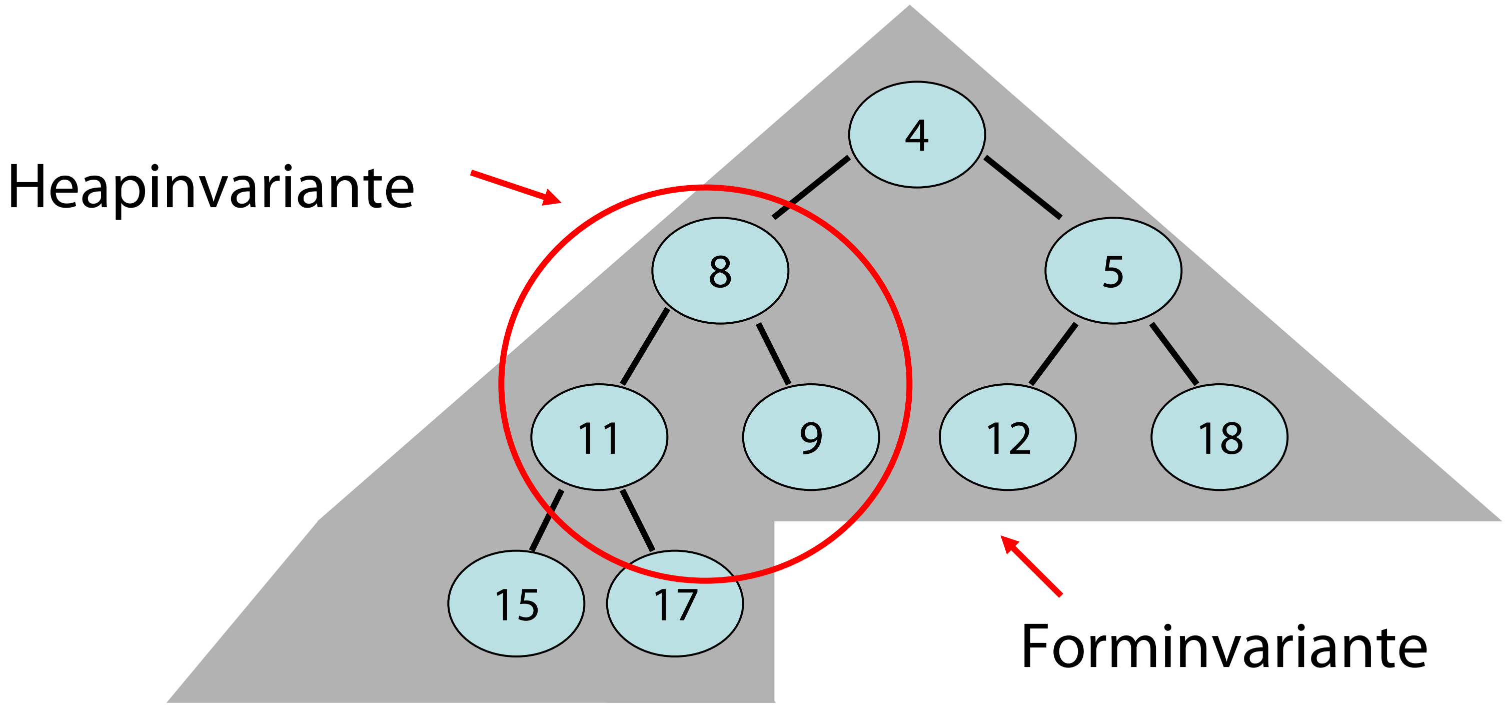
- Form-Invariante:
  vollst. Binärbaum bis auf unterste Ebene
- (Min)Heap-Invariante:

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



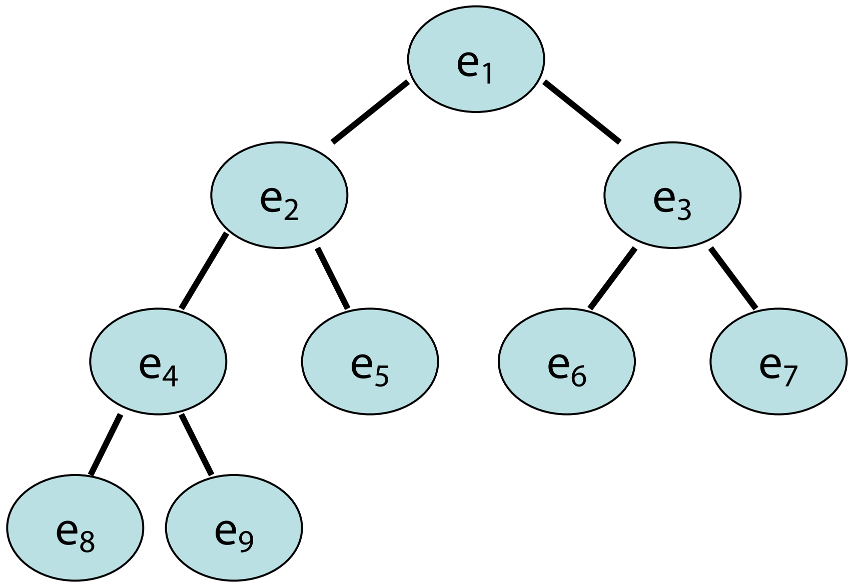
Binärer Heap (Wiederholung)

Beispiel:



Binärer Heap (Wiederholung)

Realisierung eines Binärbaums als Feld:



e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	

Binärer Heap (Wiederholung)

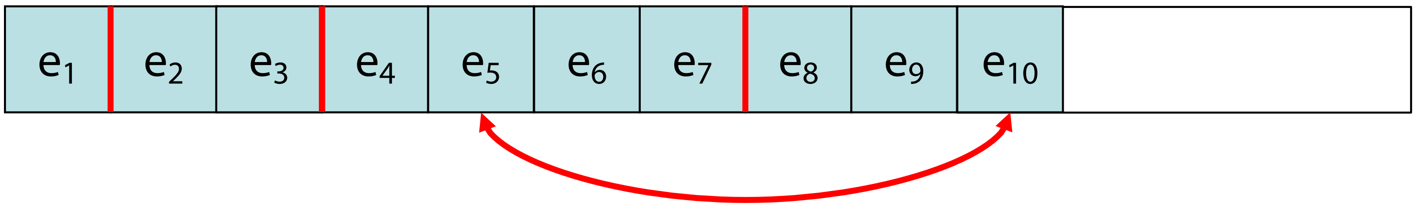
Realisierung eines Binärbaums als Feld:

e_1	e_2	e_3	e_4	e_5	e_6	e_7	e_8	e_9	

- H: Array [1..n]
- Kinder von e in H[i]: in H[2i], H[2i+1]
- Form-Invariante: H[1],…,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 = internalRepr(pq))

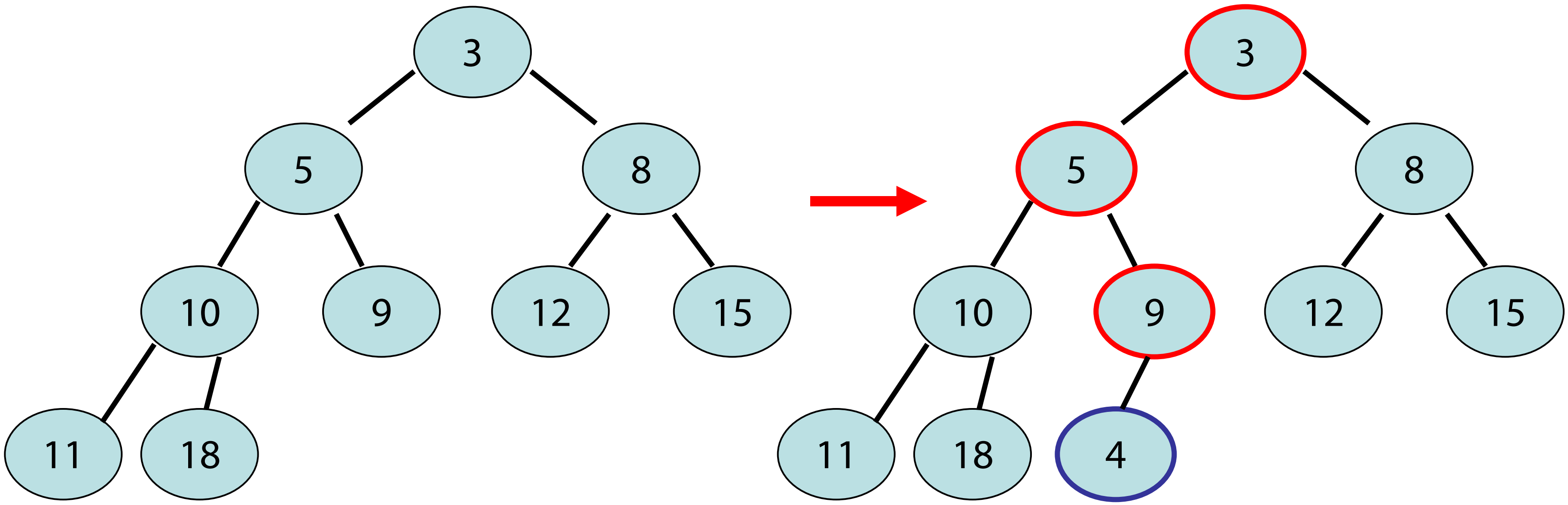
- Form-Invariante: n:=n+1; H[n]:=e
- Heap-Invariante: vertausche e mit Vater bis key(H[$\lfloor k/2 \rfloor$]) $\leq$ key(e) für e in H[k] oder e in H[1]

Insert Operation

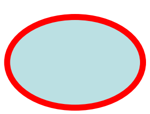
```
Procedure insert(e, pq)
    H:=internalRepr(pq); n:=n+1; H[n]:=e
    siftUp(n, H)

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

Laufzeit: O(log n)

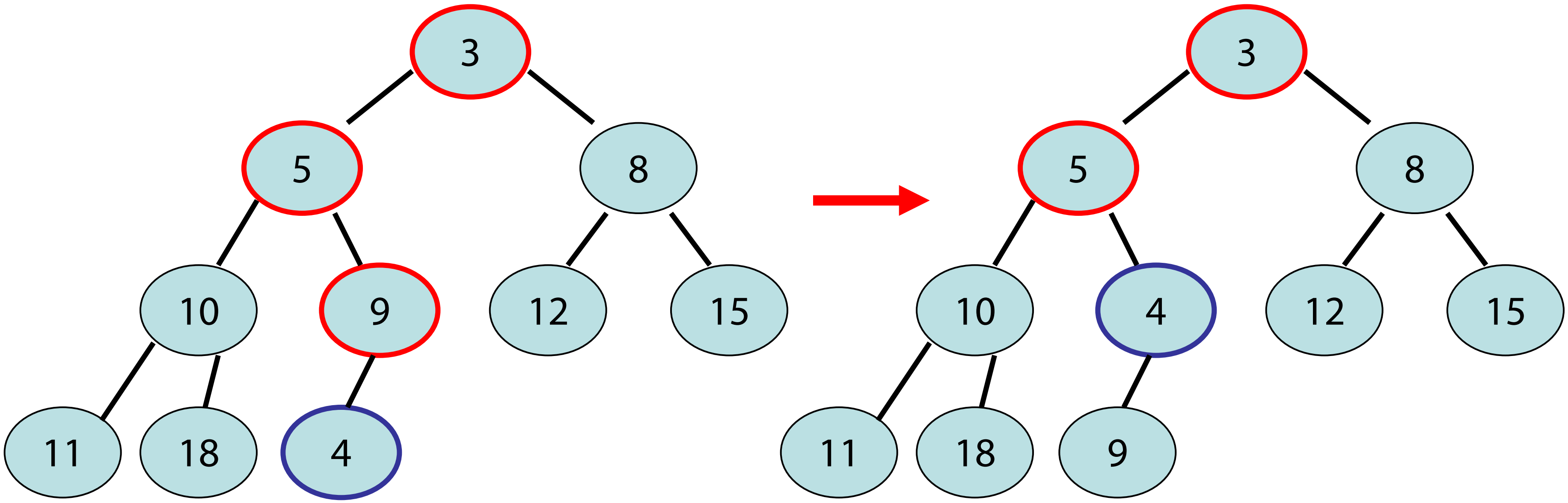
Insert - Binärer Heap



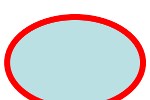
Invariante: H[k] ist minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

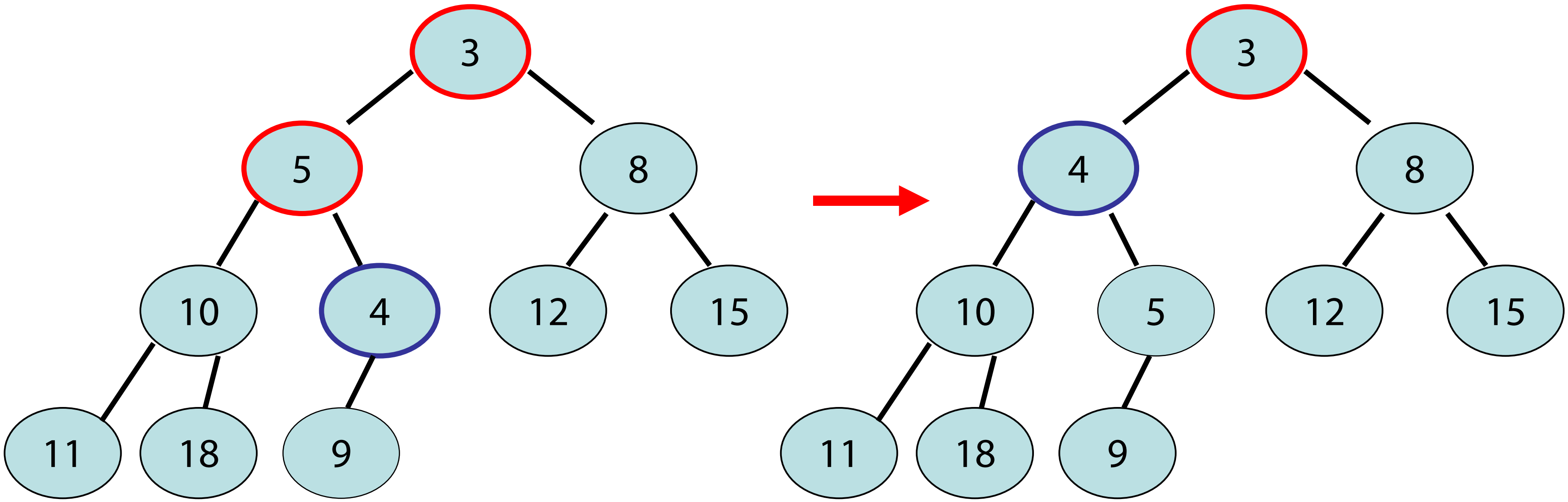
Insert Operation - Korrektheit



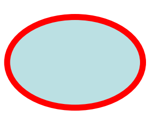
Invariante: H[k] ist minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

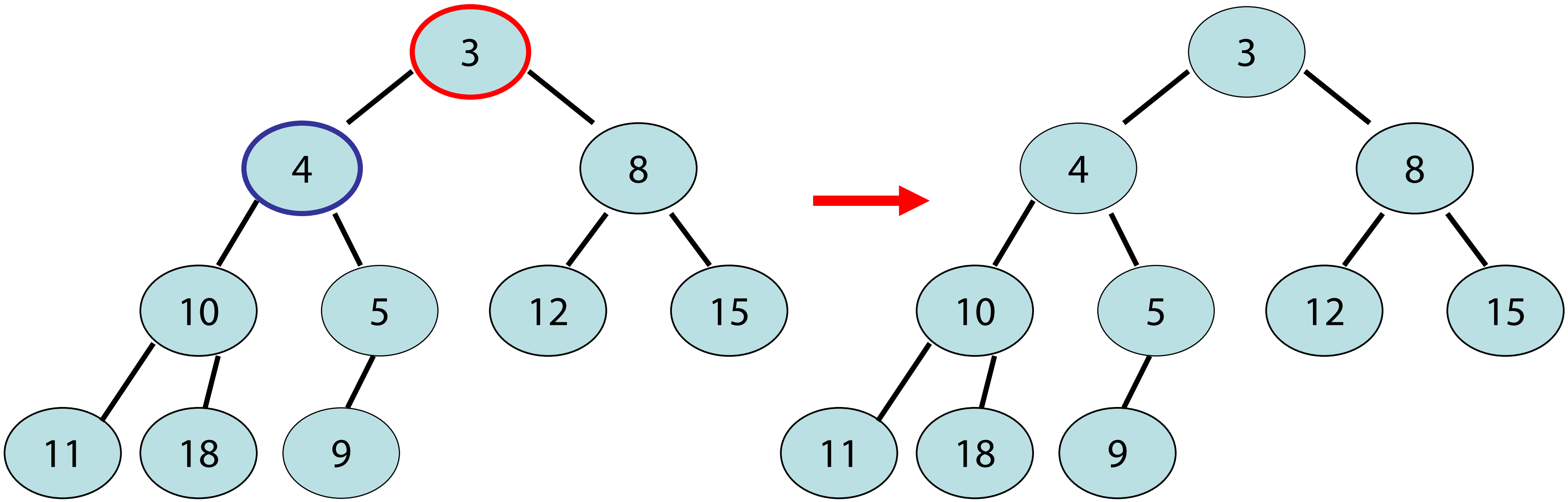
Insert Operation - Korrektheit



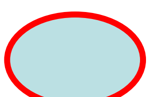
Invariante: H[k] ist minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

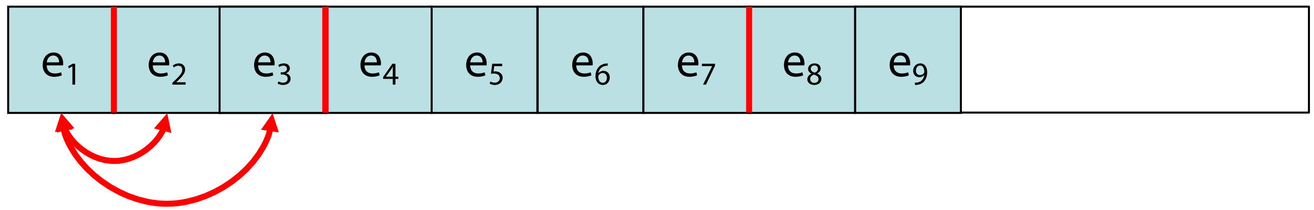
Insert Operation - Korrektheit



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


: Knoten, die Invariante eventuell verletzen

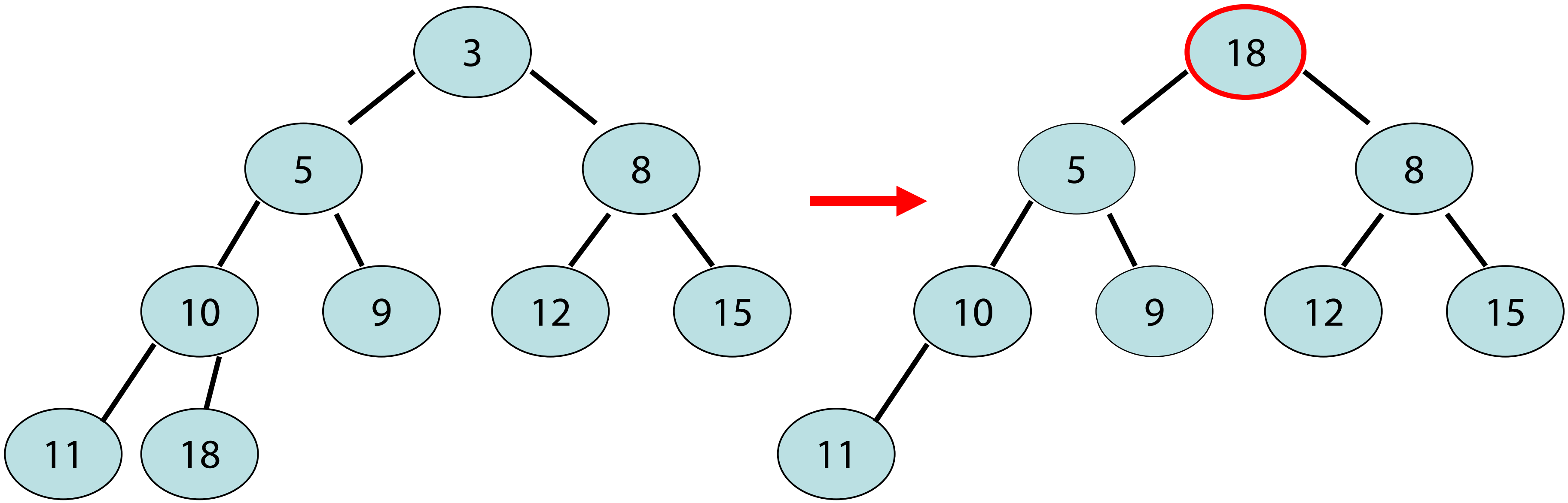
DeleteMin: Binärer Heap



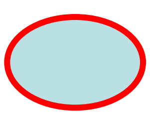
deleteMin(pq):

- 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] ≤ min({ H[2k], H[2k+1] }) für Position k von e
  oder e in Blatt

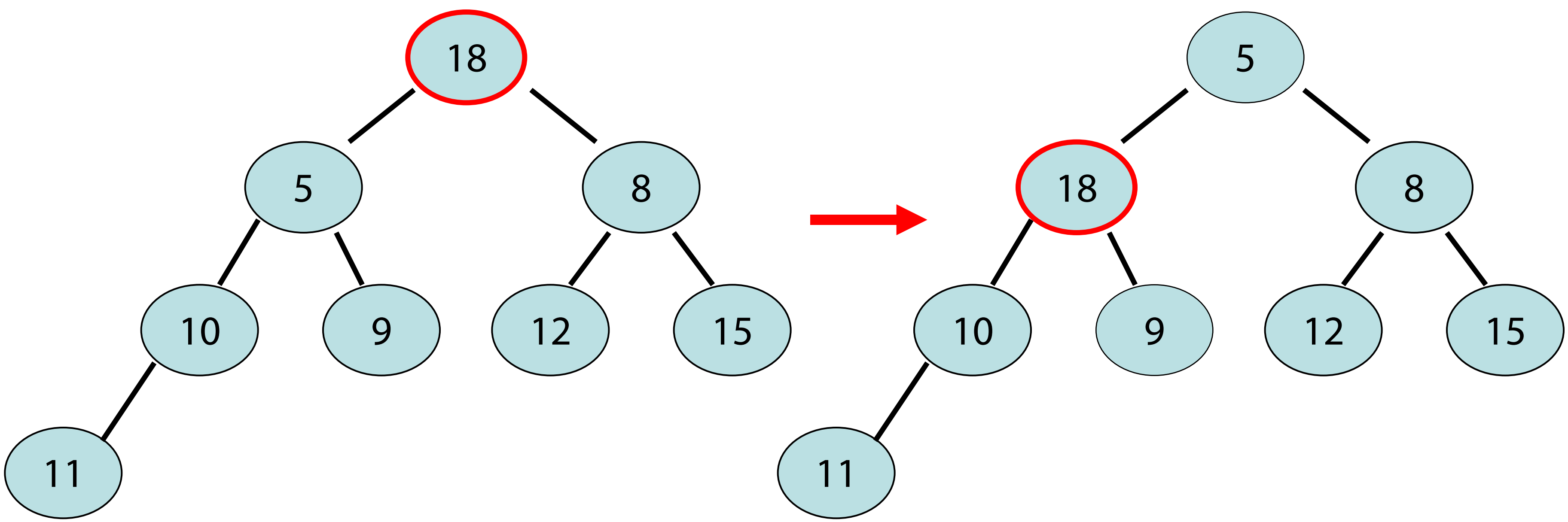
DeleteMin Operation - Korrektheit



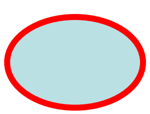
Invariante: H[k] ist minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

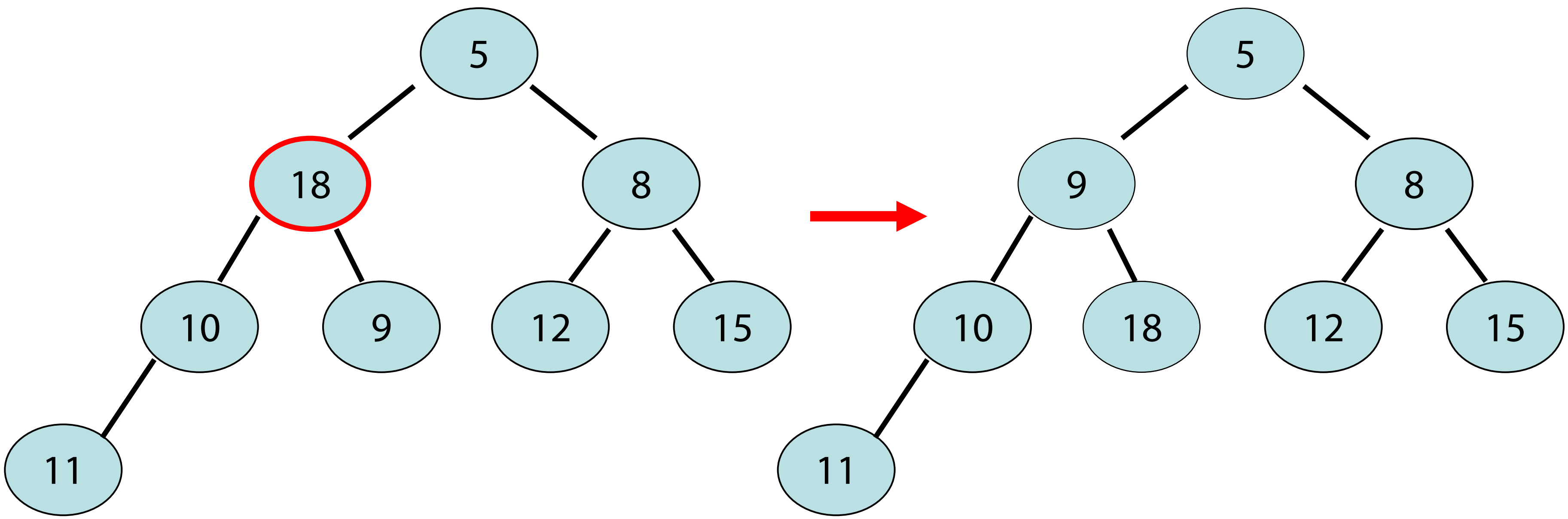
DeleteMin Operation - Korrektheit



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

```
function deleteMin(pq):
    H:= internalRepr(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 not(isLeaf(i)) do
        if not(exist(rightChild(i))) then m:=leftChild(i)
        else
            if key(H[leftChild(i)]) < key(H[rightChild(i)])
               then m:= leftChild(i)
               else m:=rightChild(i)
        if key(H[i])≤key(H[m])
            then exit    // Heap-Inv gilt
        temp := H[i]; H[i] := H[m]; H[m] := temp;
        i:=m
```

Prioritätswarteschlange mit binärem Heap

Operator	Laufzeit
insert	O(log n)
min	O(1)
deleteMin	O(log n)

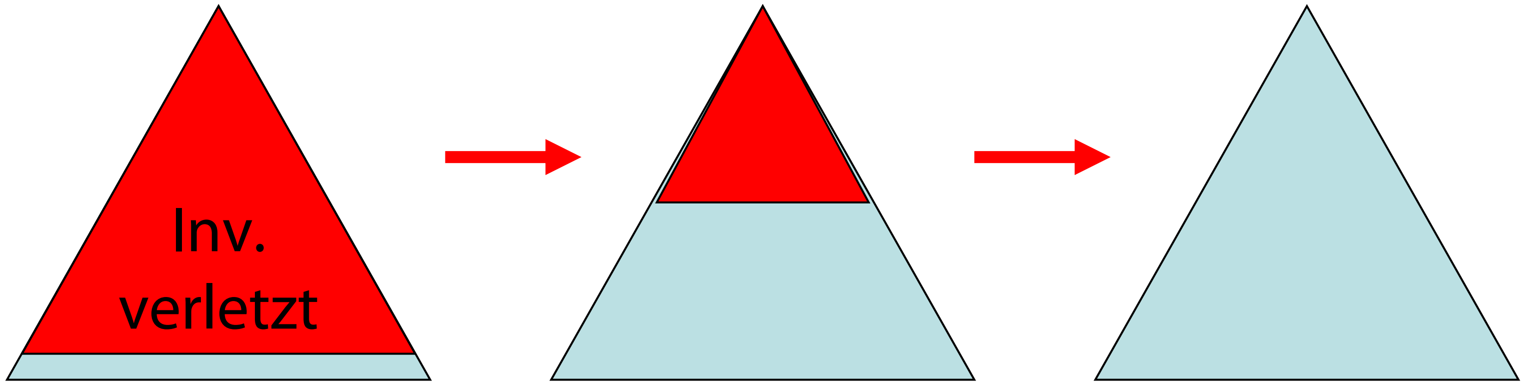
Binärer Heap

build($\{e_1,\ldots,e_n\}$, key):

- 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) auf für i=parent(n) 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 siftDown(i, H) für i=parents(n) 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 $\lfloor \log n \rfloor$...
- ... und maximal $\lceil n/2^{h+1} \rceil$ viele Knoten (Teilbäume) mit Höhe $h \in \{0,\ldots,\lfloor \log n \rfloor\}$
- 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 $x = ½$ setzen in

$$\sum_{k=0}^{\infty} k x^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

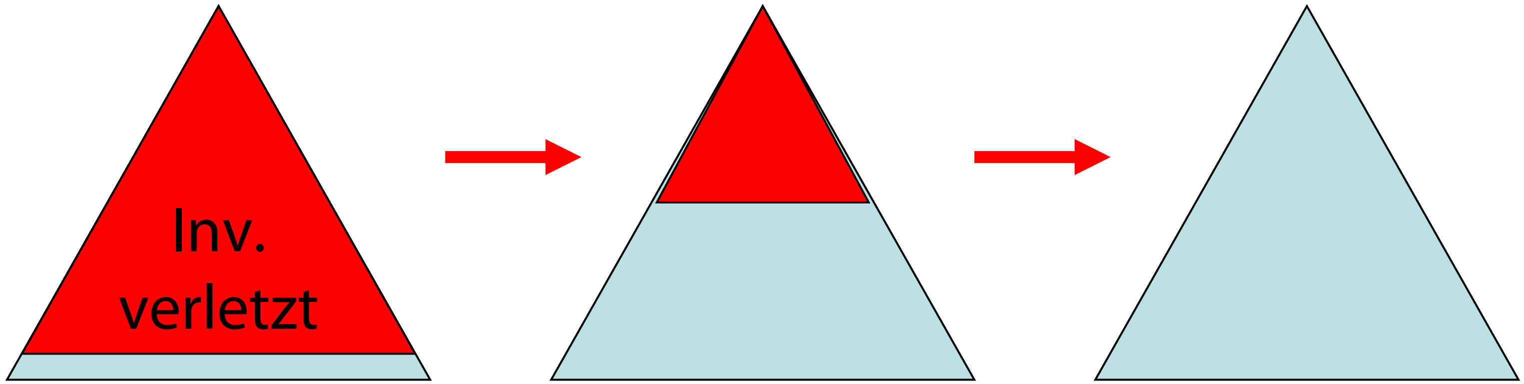
Ergibt sich aus der Ableitung der geometrischen Reihe mit $a_0 = 1$:

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

Robert W. Floyd: Algorithm 245: Treesort 3.
In: Communications of the ACM. 7, Nr. 12, S. 701, **1964**

Binärer Heap: Operation build

Setze $H[i]{:=}e_i$ für alle i. Rufe siftDown(i, H) für i=parent(n) 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)

Prioritätswarteschlangen als ADTs

- Unsortierte Liste?

Operator	Laufzeit
insert	O(1)
min	O(n)
deleteMin	O(n)
build	O(n)

- Sortiertes Array?

Operator	Laufzeit
insert	O(n)
min	O(1)
deleteMin	O(n)
build	O(n log n)

- Binärer Heap:

Operator	Laufzeit
insert	O(log n)
min	O(1)
deleteMin	O(log n)
build	O(n)