Grundlagen der Algorithmen und Datenstrukturen

Kapitel 6

Christian Scheideler + Helmut Seidl

SS 2009

Priority Queue

M: Menge von Elementen

Jedes Element e identifiziert über key(e).

Operationen:

- M.build($\{e_0,\ldots,e_{n-1}\}$): $M = \{e_0,\ldots,e_{n-1}\}$;
- M.insert(Element e): $M = M \cup \{e\}$;
- M.min(): gib $e \in M$ mit minimalem key(e) aus
- M.deleteMin(): wie M.min(), aber zusätzlich $M = M \setminus \{e\}$; für e mit minimalem key(e)

Erweiterte Priority Queue

Zusätzliche Operationen:

- M.remove(Element e): M = M\{e};
- M.decreaseKey(Element e, int Δ): key(e) = key(e)-Δ ;
  (bzw. M.decreaseKey(Element e), um M nach update von key(e) zu aktualisieren)
- M.merge(M´): M =M $\cup$ M´;

Priority Queue

- Priority Queue mittels unsortierter Liste:
  - build($\{e_0,\ldots,e_{n-1}\}$): Zeit O(n)
  - insert(e): O(1)
  - min, deleteMin: O(n)
- Priority Queue mittels sortierter Liste:
  - build($\{e_0,\ldots,e_{n-1}\}$): Zeit O(n log n)
  - insert(e): O(n)
  - min, deleteMin: O(1)

Bessere Struktur als Liste notwendig!

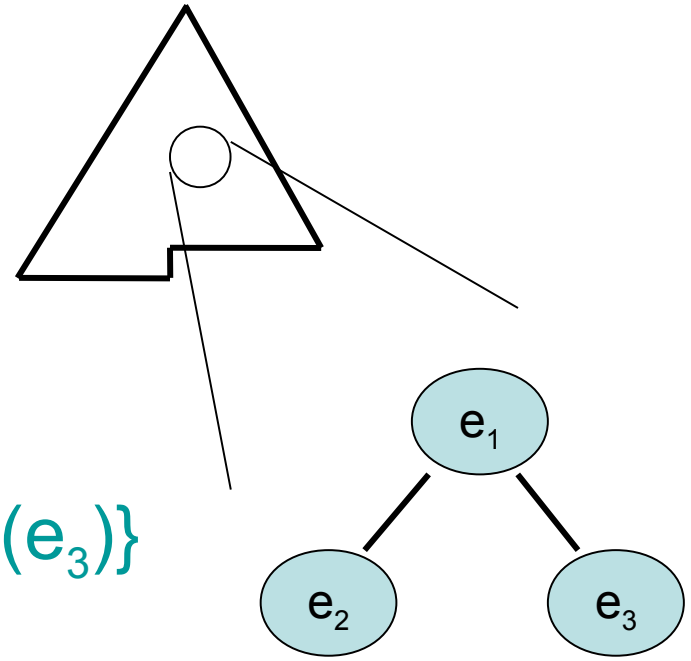
Binärer Heap

Idee: verwende binären Baum statt Liste

Bewahre zwei Invarianten:

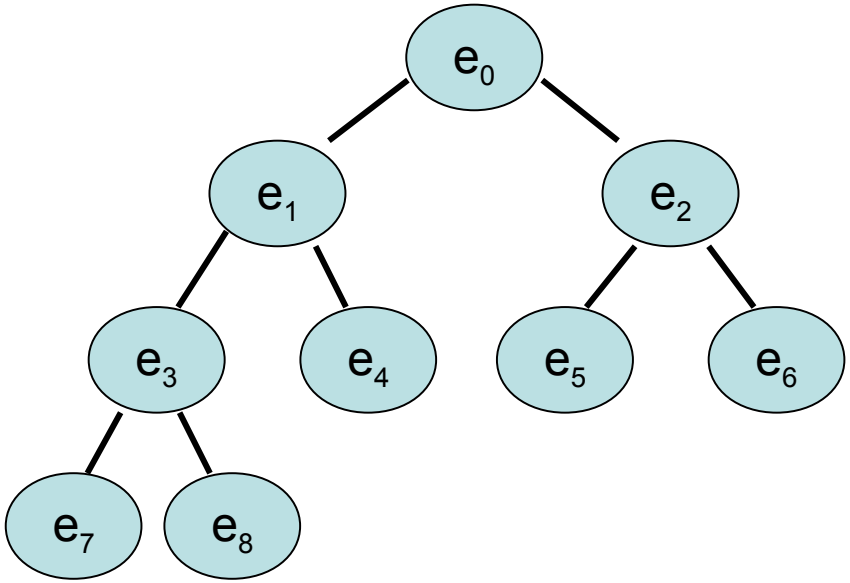
- Form-Invariante: vollst. Binärbaum bis auf unterste Ebene
- Heap-Invariante:

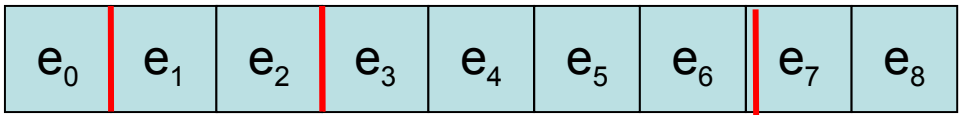
$$key(e_1) \leq \min\{key(e_2), key(e_3)\}$$



Binärer Heap

Realisierung eines Binärbaums als Feld:





Binärer Heap

Realisierung eines Binärbaums als Feld:

- Element [] H;
- Kinder von e in H[i]: in H[2i+1], H[2i+2]
- Form-Invariante: H[0],…,H[n-1] besetzt
- Heap-Invariante:
  $key(H[i]) \leq \min\{key(H[2i+1]), key(H[2i+2])\}$

Binärer Heap

Realisierung eines Binärbaums als Feld:

insert(e):

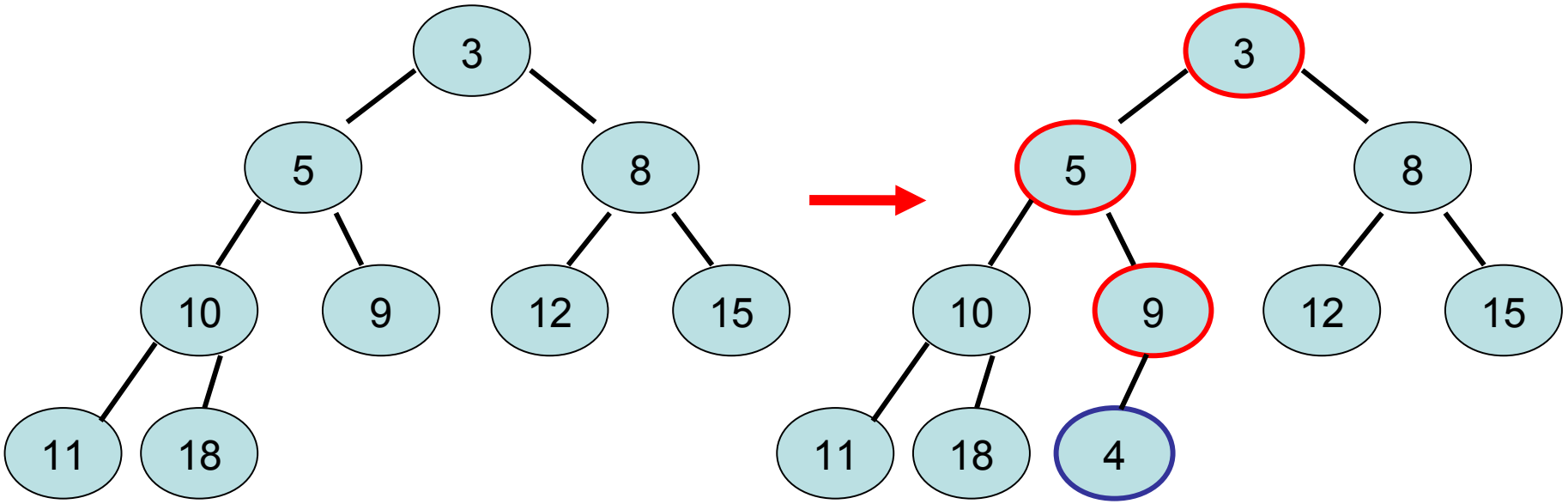
- Form-Invariante: H[n] = e; n++;
- Heap-Invariante: vertausche e mit Vater bis key(H[(k-1)/2]) ≤ key(e) für e in H[k] (oder e in H[0])

Insert Operation

```
void insert(Element e) {
   H[n] = e;
   siftUp(n); n++;
}
void siftUp(int i) {
   while (i>0 && key(H[(i-1)/2])>key(H[i])) {
      H[i] ↔ H[(i-1)/2];
      i=(i-1)/2;
}
```

Laufzeit: O(log n)

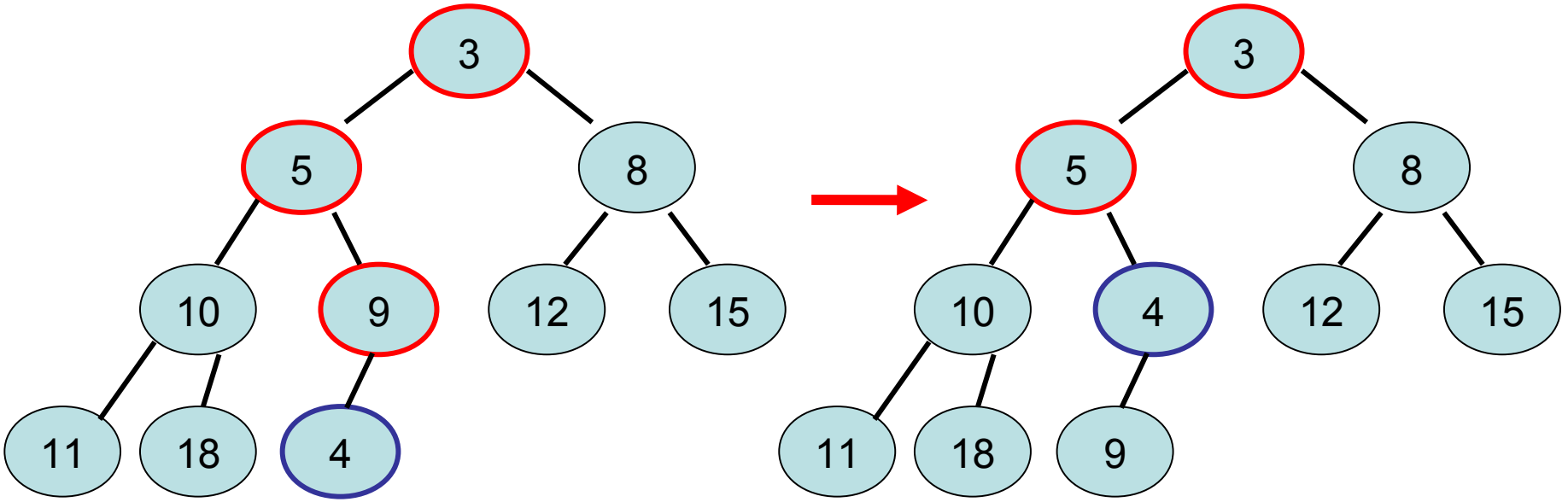
Insert Operation - Korrektheit



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

: Knoten, die Invariante eventuell verletzen

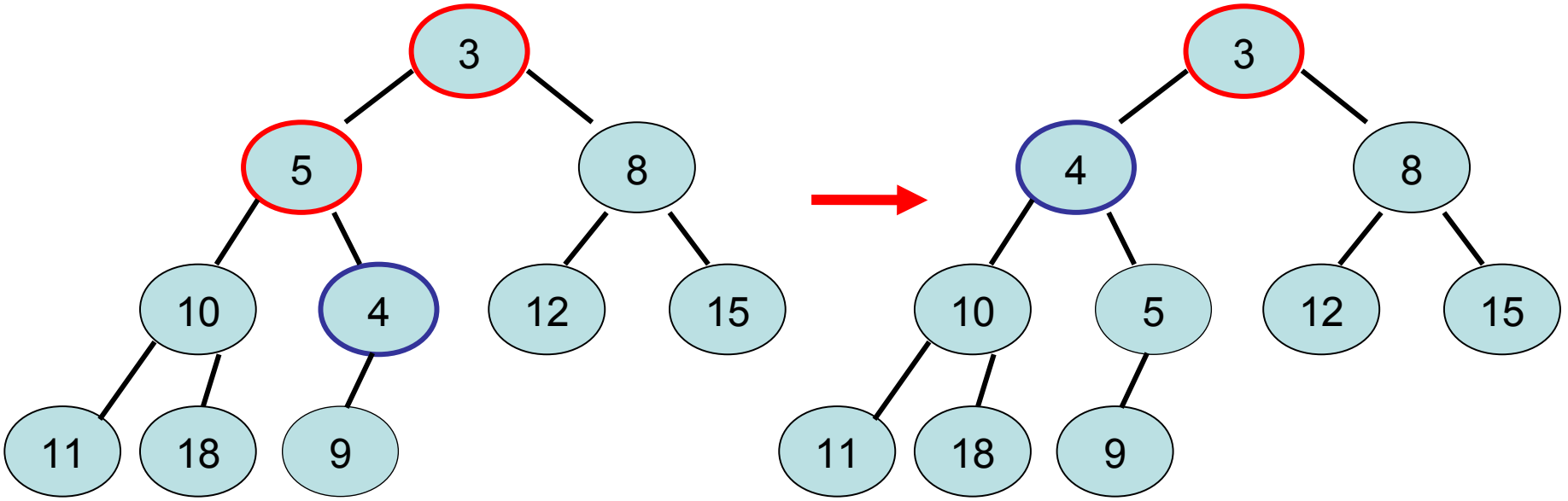
Insert Operation - Korrektheit



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

: Knoten, die Invariante eventuell verletzen

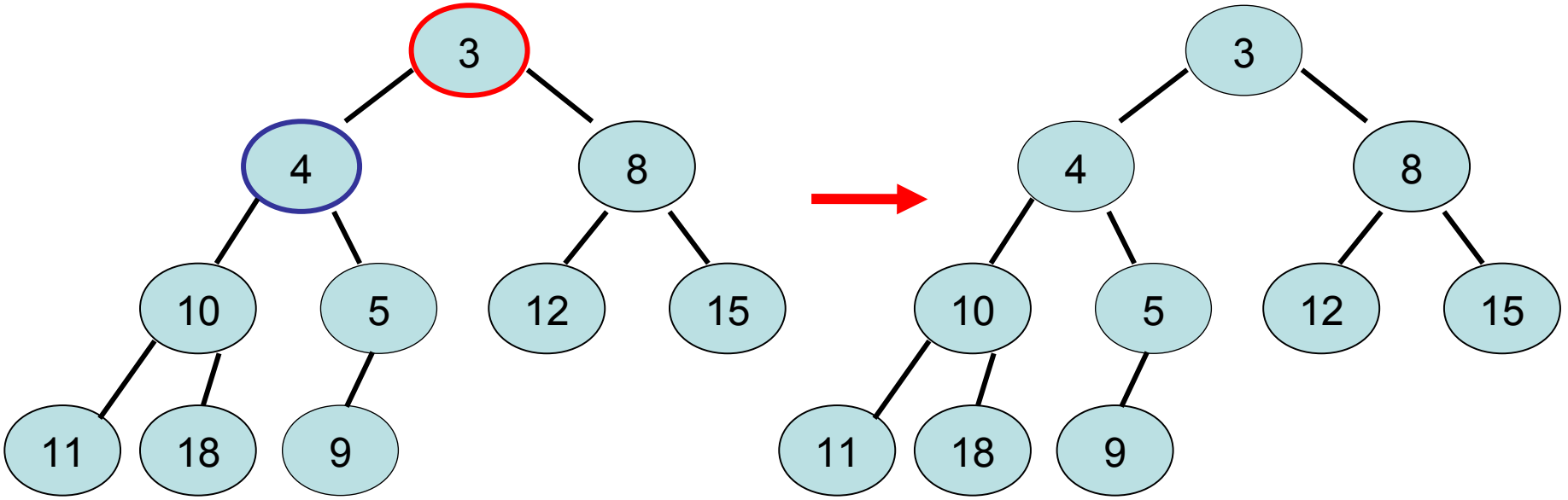
Insert Operation - Korrektheit



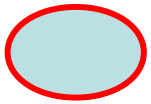
Invariante: H[k] minimal für Teilbaum von H[k]

: Knoten, die Invariante eventuell verletzen

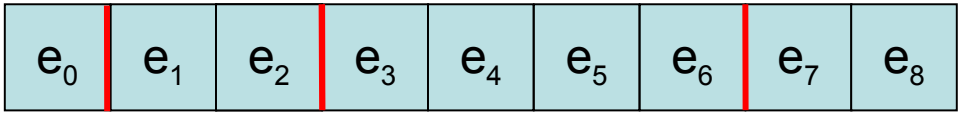
Insert Operation - Korrektheit



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

 : Knoten, die Invariante eventuell verletzen

Binärer Heap



deleteMin:

- Form-Invariante: n--; H[0]=H[n];
- Heap-Invariante: starte mit e in H[0]. Vertausche e mit Kind mit min Schlüssel bis $H[k] \leq \min\{H[2k+1],H[2k+2]\}$ für Position k von e (oder e in Blatt)

Binärer Heap

```
Element deleteMin() {
    Element e = H[0]; n--; H[0] = H[n];
    siftDown(0);
    return e;
}
void siftDown(int i) { int m;
    while (2i+1<n) {
        if (2i+2>=n)  m = 2i+1;  // m: Pos. des min. Kindes
        else if (key(H[2i+1])<key(H[2i+2])  m = 2i+1;
        else m = 2i+2;
        if (key(H[i])<=key(H[m]) return;  // Heap-Inv gilt
        H[i] ↔ H[m]; i = m;
    }
}
```

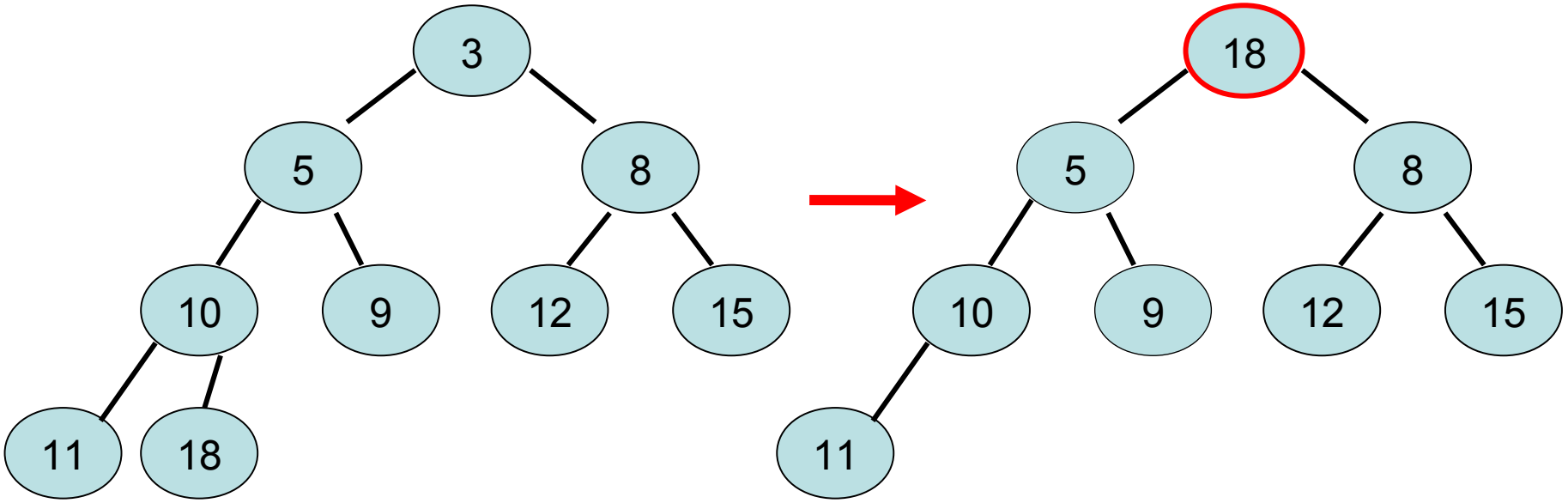
Binärer Heap

Laufzeit von deleteMin: O(log n)

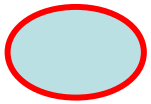
Korrektheit von deleteMin:

- Vor deleteMin: Heap-Invariante gilt für alle i, d.h. key(H[i]) minimal für Teilbaum von H[i]
- Während deleteMin:
  e in Position k: key(H[k´]) minimal für Teilbaum von H[k´] für alle k´ ungleich k (Induktion)
- Nach deleteMin: wie vor deleteMin

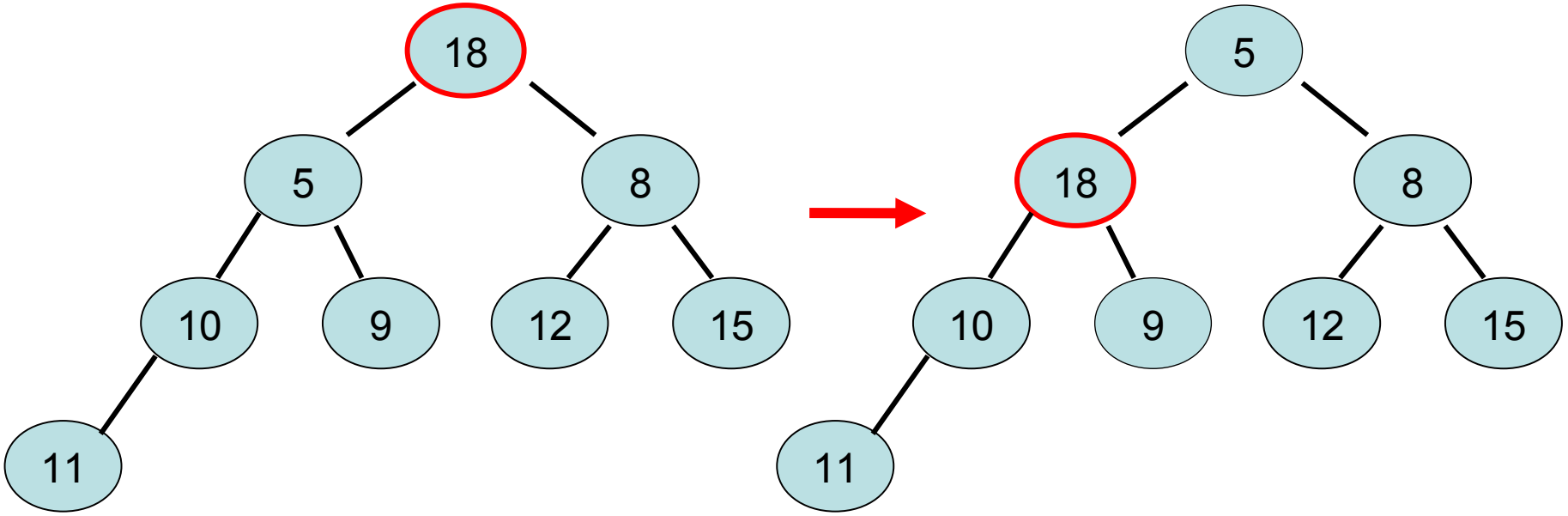
deleteMin Operation - Korrektheit



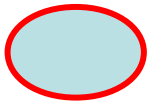
Invariante: H[k] minimal für Teilbaum von H[k]

 : Knoten, die Invariante eventuell verletzen

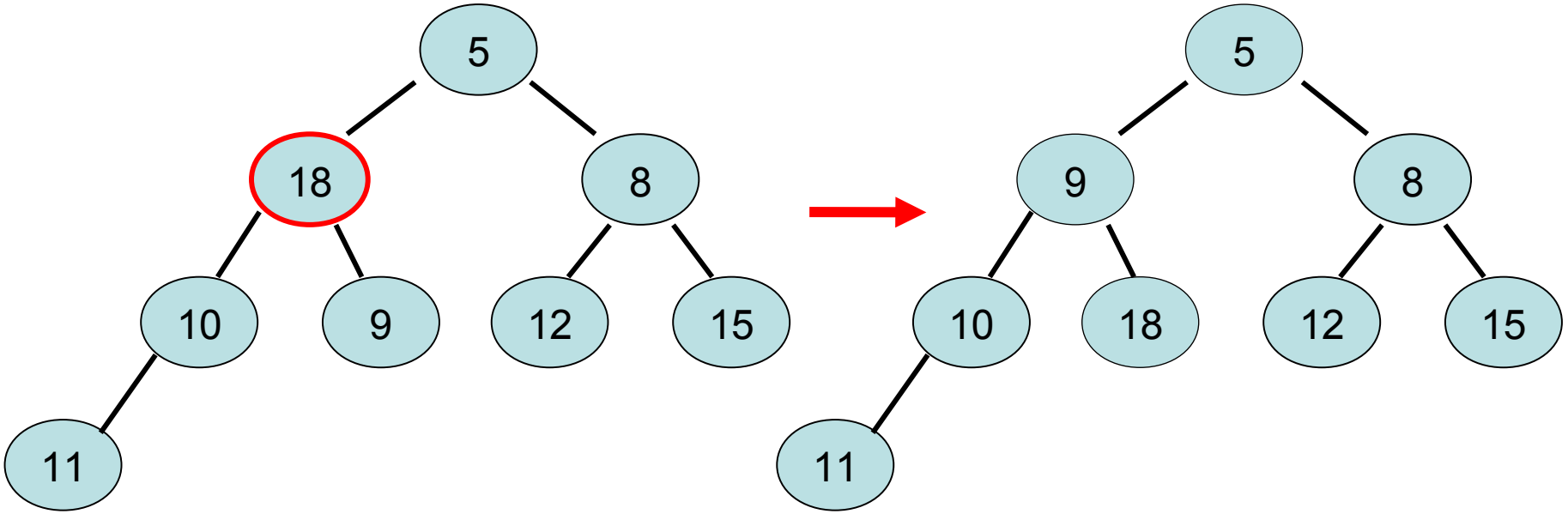
deleteMin Operation - Korrektheit



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

 : Knoten, die Invariante eventuell verletzen

deleteMin Operation - Korrektheit



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

: Knoten, die Invariante eventuell verletzen

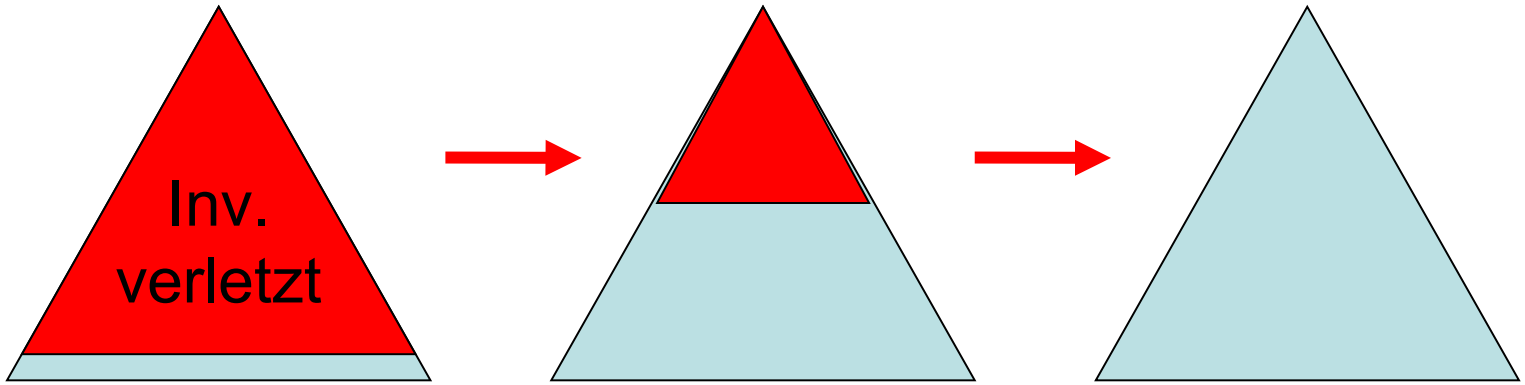
Binärer Heap

build($\{e_0,\ldots,e_{n-1}\}$):

- 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 $i=(n-1)/2$ runter bis 0 auf.
  Aufwand (mit $k=\lceil \log n \rceil$):
  $O(\sum_{1\le l<k} 2^l (k-l)) = O(2^k \sum_{j\ge 1} j/2^j) = O(n)$

Binärer Heap

Setze H[i]=e_i für alle i. Rufe siftDown(i) für i = (n-1)/2 runter bis 0 auf.



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

Binärer Heap

Laufzeiten:

- build($\{e_0,\ldots,e_{n-1}\}$): O(n)
- insert(e): O(log n)
- min: O(1)
- deleteMin: O(log n)

Einsatz in Selection Sort (Heapsort):

Verbessert Laufzeit auf O(n log n)

Heapsort

```
static void heapSort(Seq<Element> s) {
   Heap<Element> M = new Heap<Element>.build(s);
                    // Laufzeit O(n)
   s = ∅;
    while (!M.empty())
       s.pushBack(M.deleteMin());
                    // Laufzeit O(log n)
}
```

Gesamtlaufzeit: O(n log n)

Erweiterte Priority Queue

Zusätzliche Operationen:

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

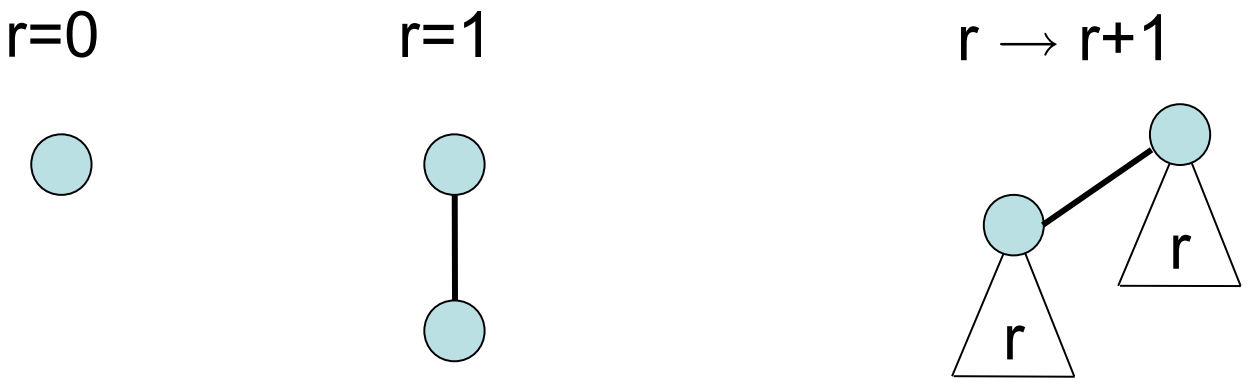
Remove und decreaseKey in Zeit O(log n) in Heap (wenn Position von e bekannt), aber merge ist teuer (Θ (n) Zeit)!

Binomial-Heap

Binomial-Heap basiert auf Binomial-Bäumen

Binomial-Baum muss erfüllen:

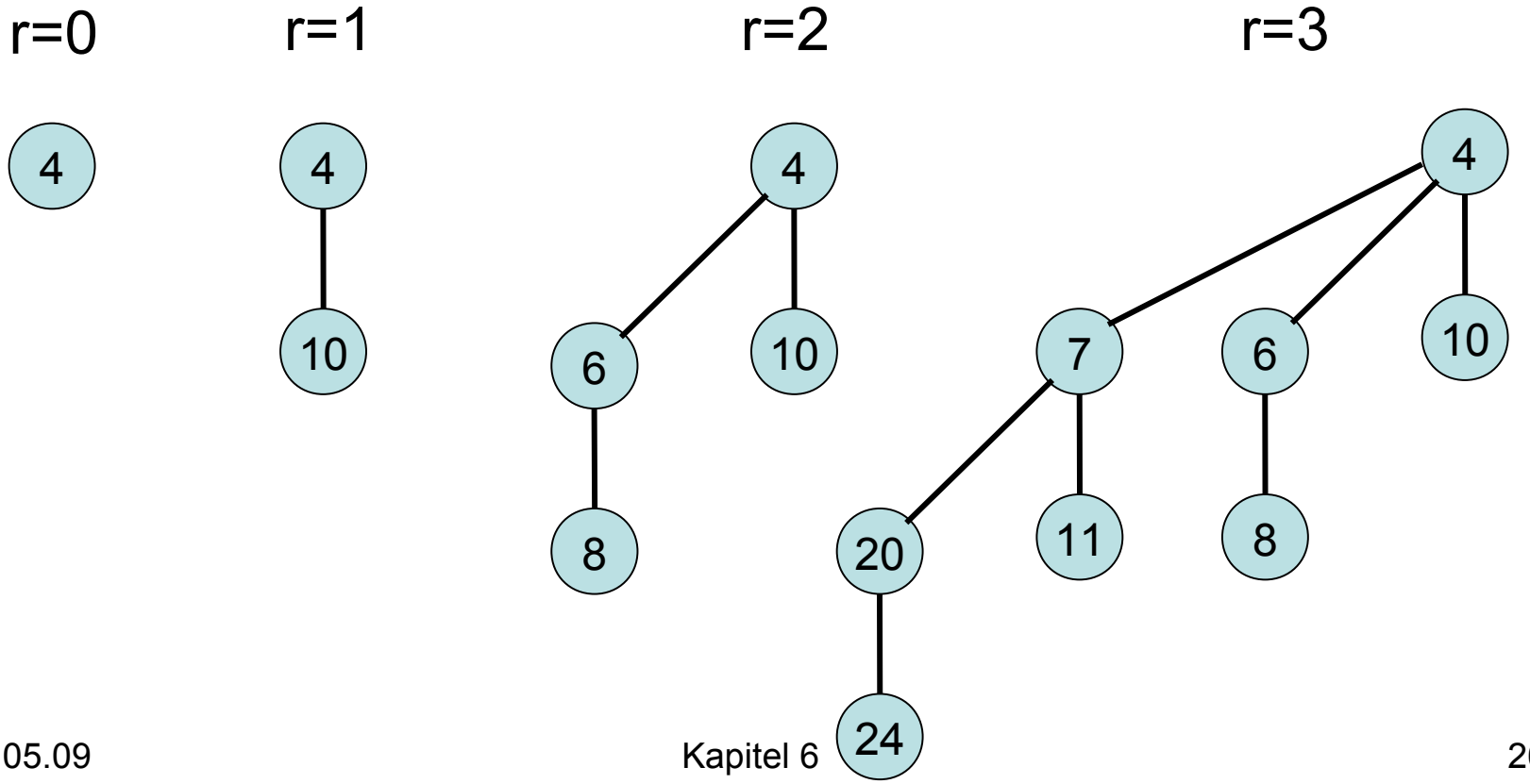
- Form-Invariante (r: Rang):



- Heap-Invariante (key(Vater) ≤ key(Kinder))

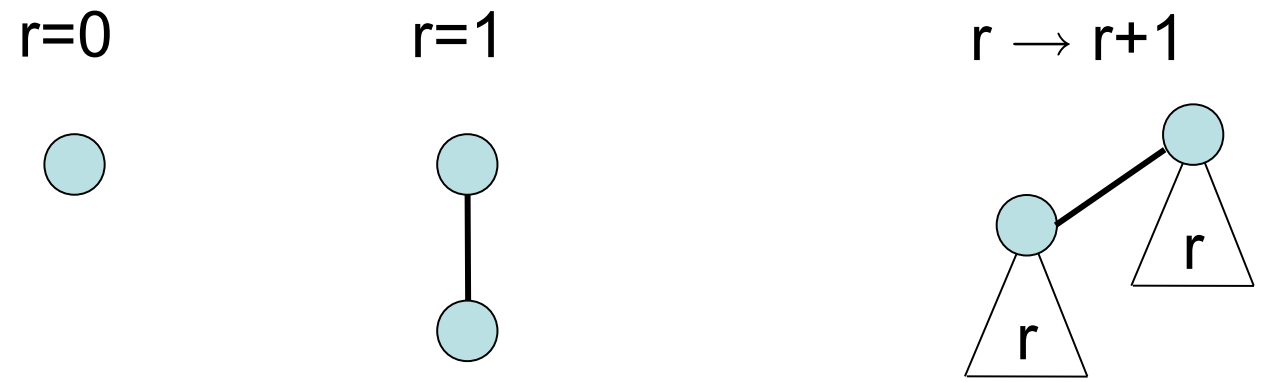
Binomial-Heap

Beispiel für korrekte Binomial-Bäume:



Binomial-Heap

Eigenschaften von Binomial-Bäumen:

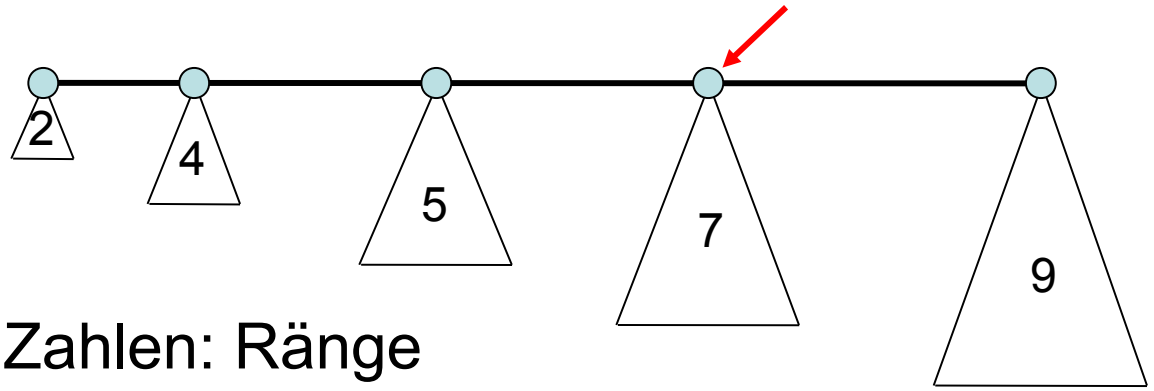


- 2^r Knoten
- maximaler Grad r (bei Wurzel)
- Wurzel weg: zerfällt in Binomial-Bäume mit Rang 0 bis r-1

Binomial-Heap

Binomial-Heap:

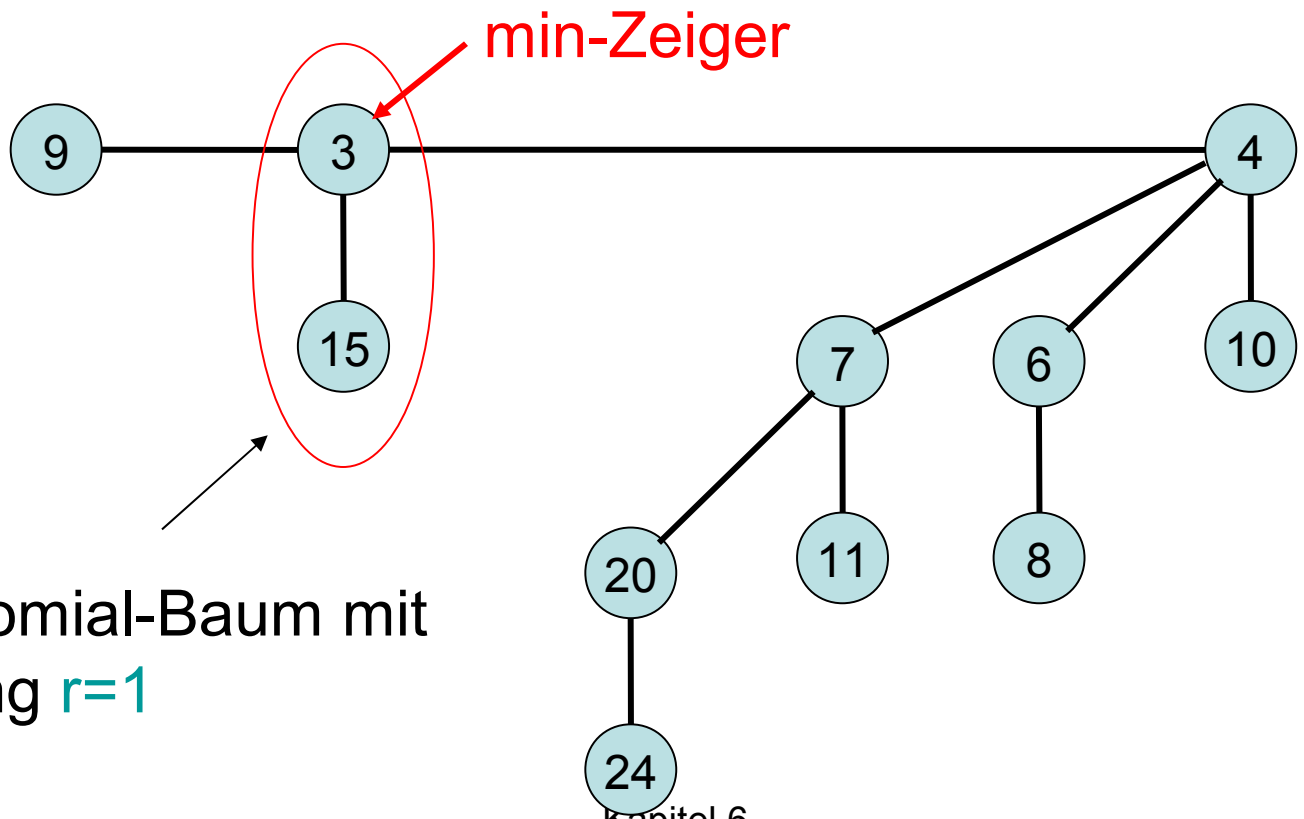
- verkettete Liste von Binomial-Bäumen
- Pro Rang maximal 1 Binomial-Baum
- Zeiger auf Wurzel mit minimalem key



Zahlen: Ränge

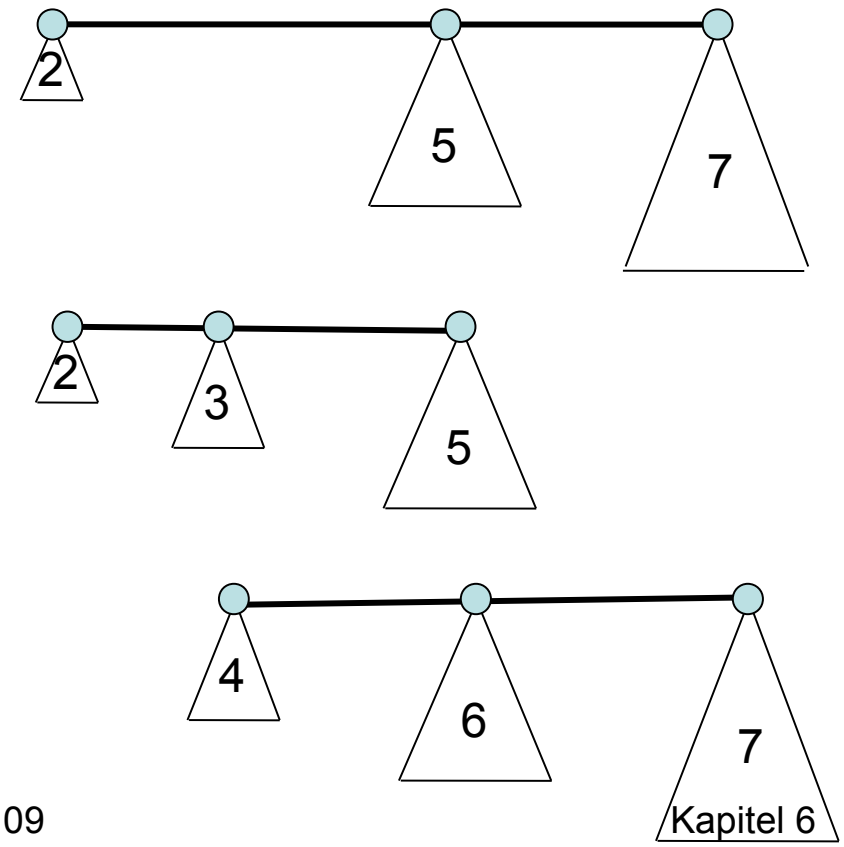
Binomial-Heap

Beispiel eines korrekten Binomial-Heaps:



Binomial-Heap

Merge von Binomial-Heaps H_1 und H_2:



H_1

H_2

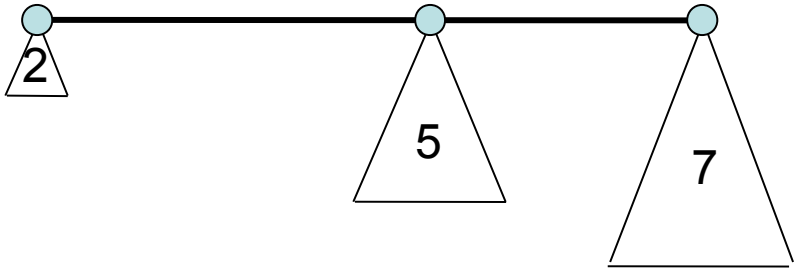
wie Binäraddition

10100100

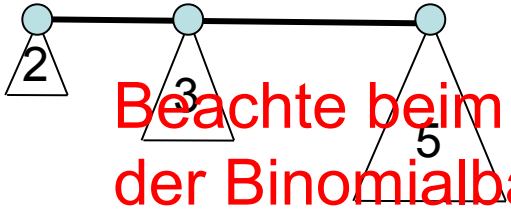
\+ 101100

11010000

Beispiel einer Merge-Operation



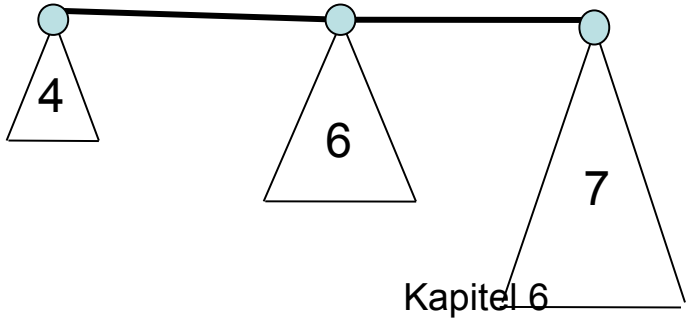
H_1



H_2

Zahlen geben die Ränge an

Beachte beim Mergen der Binomialbäume die Heap-Eigenschaft!



Ergebnis-Heap

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 B_i, durch Löschen von Minimum: $B_i \rightarrow B_0,\ldots,B_{i-1}$, diese zurückmergen in Binomial-Heap, Zeit $O(\log n)$

Binomial-Heap

- decreaseKey(e,Δ): siftUp-Operation in Binomial-Baum von e, Zeit O(log n)

- remove(e): setze key(e)=-∞ und wende siftUp-Operation auf e an bis e in der Wurzel, dann weiter wie bei deleteMin, Zeit O(log n)

Weitere Verbesserungen

Fibonacci-Heap: Verbesserung des Binomial-Heaps, so dass amortisierte Kosten von decreaseKey O(1) sind

Keys ganzzahlig: Priority Queues bekannt, die Zeit O(1) für decreaseKey und insert und Zeit O(log log n) für deleteMin benötigen.

Nächstes Kapitel

Thema: Suchstrukturen

Ziel: Operationen insert, remove und locate mit Laufzeit O(log n) pro Operation.

Locate(k): finde nächsten Nachfolger zu k