Algorithmen und Datenstrukturen 2

Lerneinheit 1: Priority Queues

Prof. Dr. Christoph Karg

Studiengang Informatik
Hochschule Aalen

Sommersemester 2016

Einleitung

Das Thema dieser Lerneinheit sind Priority Queues. Hierunter versteht man Datenstrukturen, in denen Datensätze mit einem Prioritätsschlüssel gespeichert werden.

- Wiederholung: Heap Sort
- Priority Queue auf Basis von Heap Sort
- Binomial Heaps

Heap Sort

Der Erfinder von Heap Sort ist J. W. J. Williams. Der Algorithmus wurde 1964 veröffentlicht und hat folgende Eigenschaften:

- Die Sortierung erfolgt „in place“, d.h., es ist nur konstanter zusätzlicher Speicherplatz notwendig
- Es wird eine Heap Datenstruktur eingesetzt ($\rightsquigarrow$ Interpretation des Arrays als Binärbaum)
- Die Laufzeit von Heap Sort ist $O(n \log_2 n)$

Wissenswertes über Binärbäume

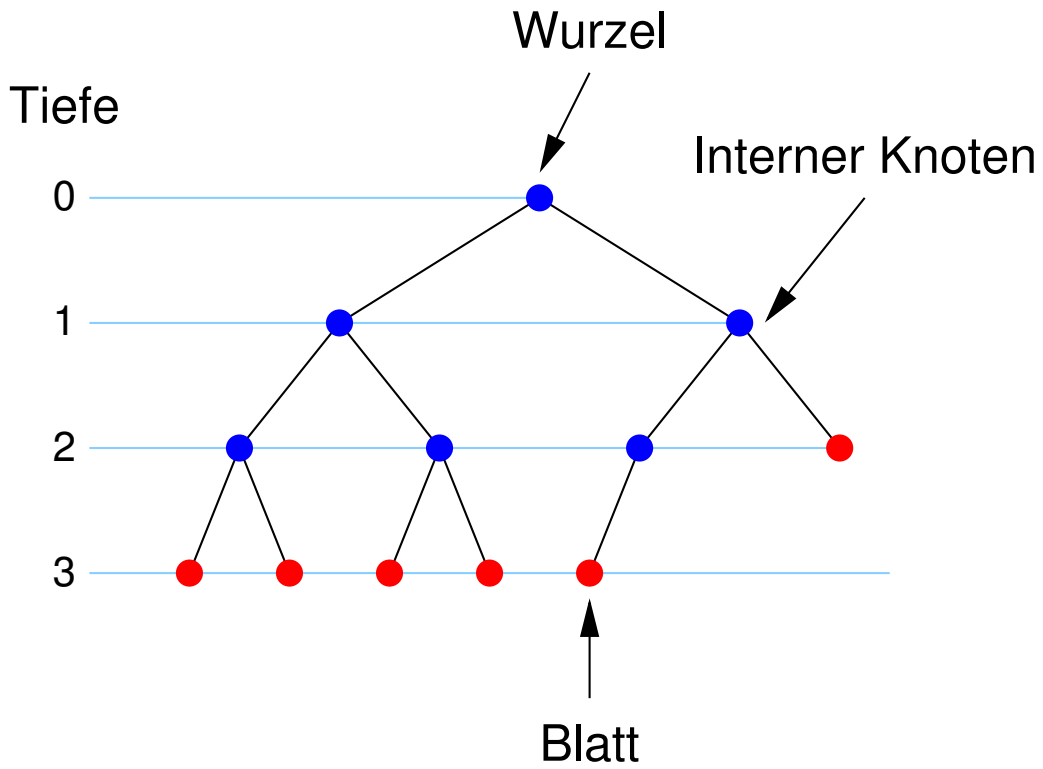
- Ein Binärbaum ist ein Baum mit der Eigenschaft, dass jeder Knoten höchstens zwei Nachfolger hat
- Die Tiefe eines Knotens ist gleich der Anzahl der Kanten des kürzesten Pfads vom Knoten zur Wurzel des Baums
- Die Tiefe des Baums ist gleich der maximalen Tiefe eines Blatts
- Ein Binärbaum heißt balanciert, falls für zwei beliebige Blätter u und v gilt:

$$|\text{Tiefe}(u) - \text{Tiefe}(v)| \leq 1$$

- Es gilt: ein balancierter Binärbaum mit n Knoten hat Tiefe $\lfloor log_2 n \rfloor$

Beispiel Binärbaum

Binärbaum mit 12 Knoten



Binärer Heap

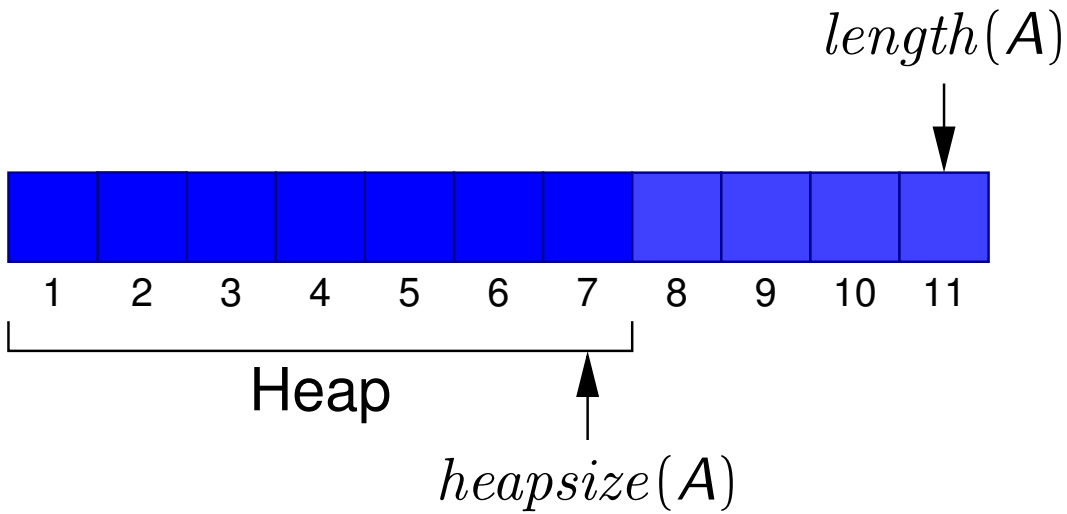
Gegeben: Array A mit n Elementen

Zwei Attribute:

- $length(A)$: Anzahl der Elemente in A
- $heapsize(A)$: Anzahl der Elemente im Heap A

Es muss gelten: $heapsize(A) \leq length(A)$

Anschaulich:



Binärer Heap (Forts.)

Idee: Interpretiere das Array A als balancierten Binärbaum

Eigenschaften:

- Jeder Index steht für einen Knoten
- Die Tiefe des Heaps ist $\lfloor \log_2 n \rfloor = \Theta(\log_2 n)$
- Wurzel des Baums ist der Index 1
- Der Elternknoten von i ist $parent(i) = \lfloor \frac{i}{2} \rfloor$, wobei $i = 1, 2, \ldots, n$
- Das linke bzw. rechte Kind von i, $i = 1, 2, \ldots, \lfloor \frac{n}{2} \rfloor$, ist
  - $left(i) = 2 \cdot i$
  - $right(i) = 2 \cdot i + 1$

Beachte: Die Operationen $\lfloor \frac{i}{2} \rfloor$ und $2 \cdot i$ sind durch einen Rechts- bzw. Linksshift effizient berechenbar

Heap Eigenschaft

- Max-Heap: Für alle Knoten $i = 1, 2, \ldots, n$ gilt:

$$A[i] \leq A[parent(i)]$$

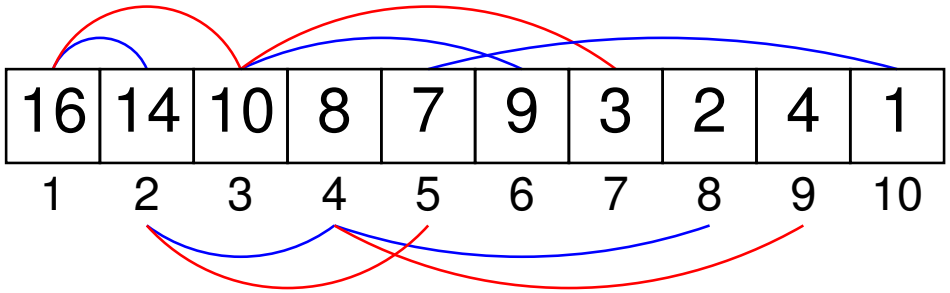
  Die Wurzel des Heaps enthält also das größte Element des Arrays.

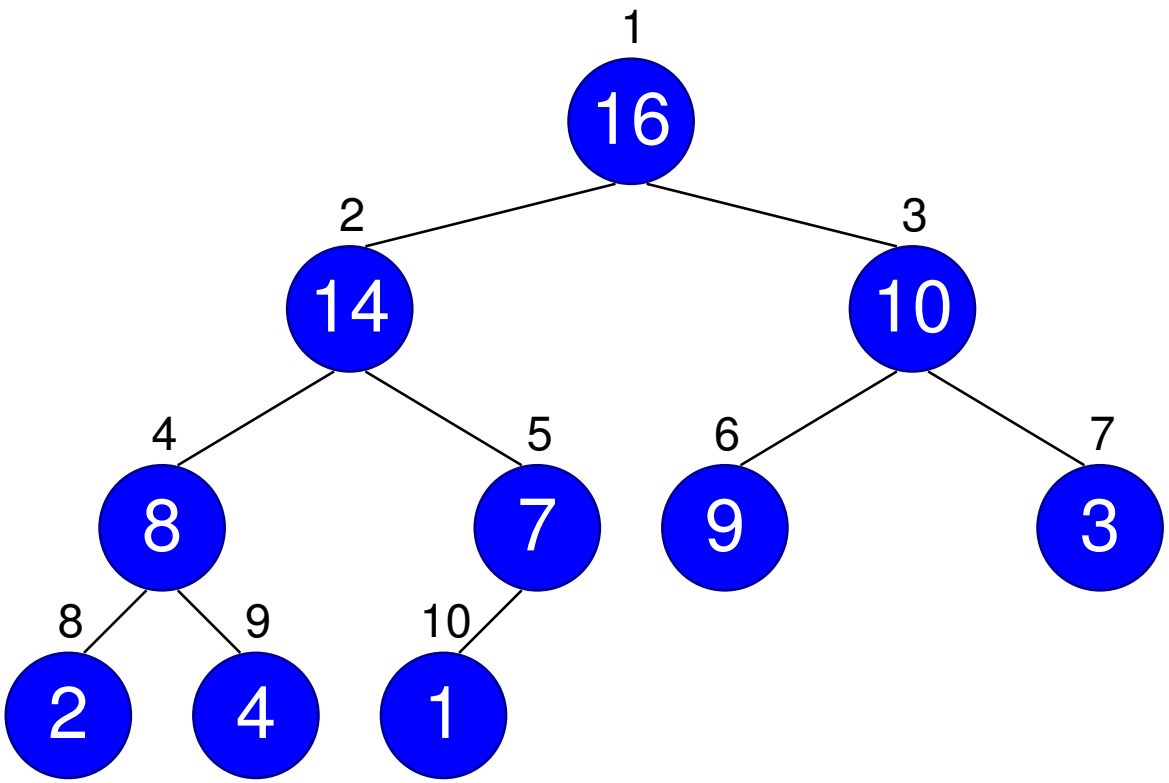
- Min-Heap: Für alle Knoten $i = 1, 2, \ldots, n$ gilt:

$$A[i] \geq A[parent(i)]$$

  Die Wurzel des Heaps enthält also das kleinste Element des Arrays.

Ansicht eines Heaps





Heap Operationen

Heap Sort setzt sich zusammen aus folgenden Algorithmen:

- MaxHeapify: Herstellen der Max-Heap Eigenschaft für einen Teilbaum (Laufzeit $O(\log_2 n)$)
- BuildMaxHeap: Erzeugen eines Max-Heaps aus einem unsortierten Array (Laufzeit $O(n)$)
- HeapSort: Sortierverfahren (Laufzeit $O(n \log_2 n)$)

Herstellen der Max-Heap Eigenschaft

Gegeben: Array A und Index i, wobei die Bäume mit Wurzel $left(i)$ bzw. $right(i)$ Max-Heaps sind

Aufgabe: Stelle für den Heap mit Wurzel i die Max-Heap Eigenschaft her

Idee: Lasse das Element in $A[i]$ „geeignet“ nach unten sinken

MaxHeapify(A, i)

MaxHeapify(A, i)

Input: Array A mit Max-Heaps $A[left(i)]$, $A[right(i)]$

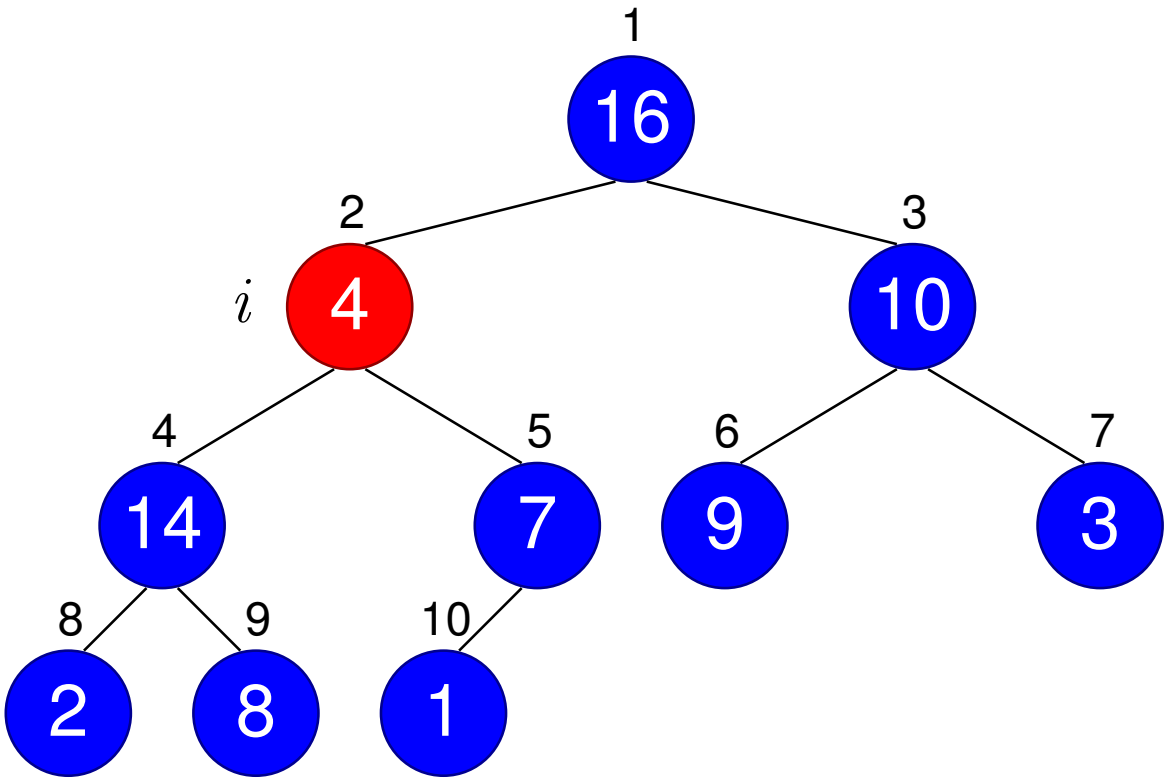
Output: Modifiziertes Array A mit Max-Heap $A[i]$

```
1  l := left(i)
2  r := right(i)
3  if l ≤ heapsize(A) and A[l] > A[i] then
4     largest := l
5  else
6     largest := i
7  if r ≤ heapsize(A) and A[r] > A[largest] then
8     largest := r
9  if largest ≠ i then
10    Vertausche A[i] und A[largest]
11    MaxHeapify(A, largest)
```

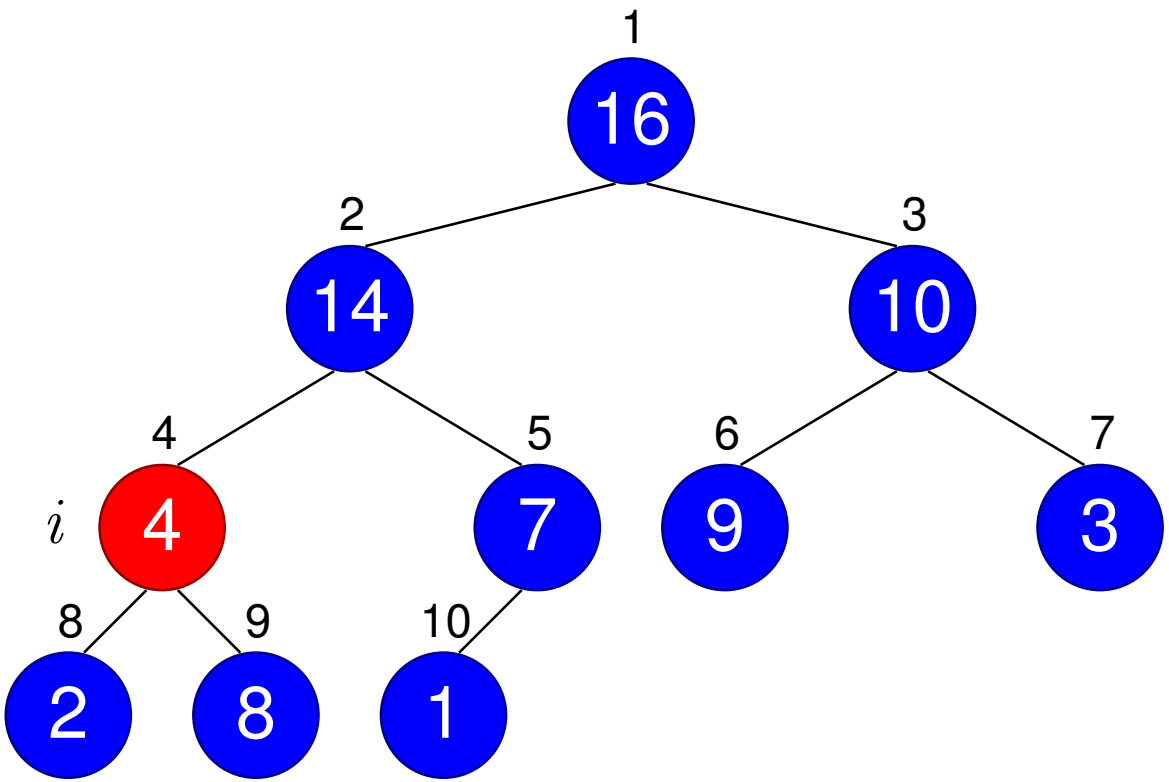
Bemerkungen zu MaxHeapify(A, i)

- In Zeilen 3–8 wird der Index $largest$ des größten Elements in $\{A[i], A[left(i)], A[right(i)]\}$ gesucht
- Zeile 9: Ist $A[i] \neq A[largest]$, dann werden die beiden Elemente vertauscht.
- Zeile 10: Im Falle einer Vertauschung muss für den Baum mit Wurzel $largest$ die Max-Heap Eigenschaft hergestellt werden.

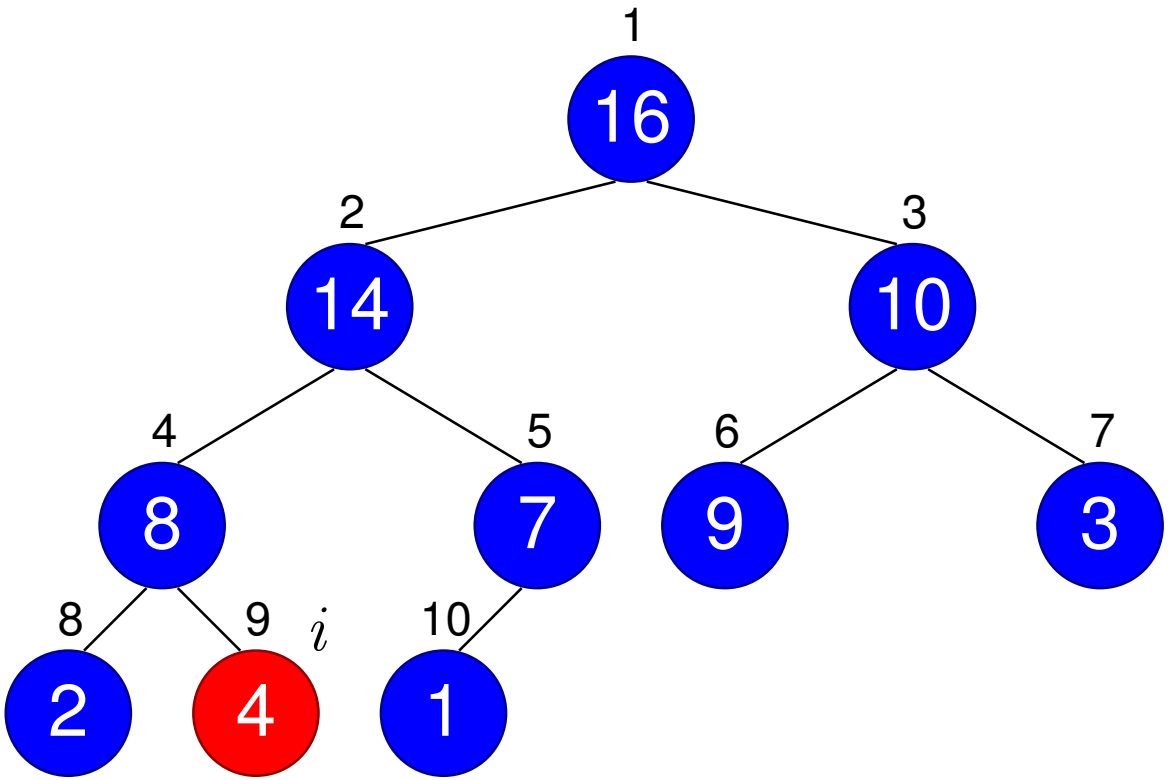
Heap Eigenschaft



Heap Eigenschaft (Forts.)



Heap Eigenschaft (Forts.)



Korrektheit von $\textsc{MaxHeapify}(A, i)$

Induktionsbehauptung: $\textsc{MaxHeapify}(A, i)$ ist für Heaps der Tiefe t korrekt.

Induktionsanfang: $t = 0$, d.h., der Heap besteht aus $A[i]$ ✓

Induktionsschritt: $t \rightsquigarrow t + 1$. Betrachte den Aufruf von *MaxHeapify*(A, i), wobei i die Wurzel eines Baums mit Tiefe $t + 1$ ist.

- Nach Ausführung von Zeile 10 enthält $A[i]$ das größte Element in $\{A[i], A[left(i)], A[right(i)]\}$.
- Wird $\textsc{MaxHeapify}(A, largest)$ in Zeile 11 aufgerufen, dann ist $largest = l$ oder $largest = r$. Die Bäume mit Wurzel l bzw. r haben eine Tiefe $\leq t$. Laut Induktionsannahme ist $largest$ nach Ausführung von $\textsc{MaxHeapify}$ ein Max-Heap.

Somit ist auch i ein Max-Heap. ✓

Laufzeit von $\textsc{MaxHeapify}(A, i)$

- Die Laufzeit von $\textsc{MaxHeapify}(A, i)$ ist linear in Höhe des Heaps mit Wurzel i.
- Ist t die Tiefe des Knotens i, dann ist die Anzahl der rekursiven Aufrufe $\leq \log_2 n - t$, d.h., die Laufzeit von $\textsc{MaxHeapify}(A, i)$ ist $O(\log_2 n - t)$.
- Schlimmstenfalls ist die Tiefe des Knotens i gleich 0. In diesem Fall ist i die Wurzel des Binärbaums.

Fazit: Die Laufzeit von $\textsc{MaxHeapify}$ im schlimmsten Fall $O(\log_2 n)$.

Erstellen eines Max-Heaps

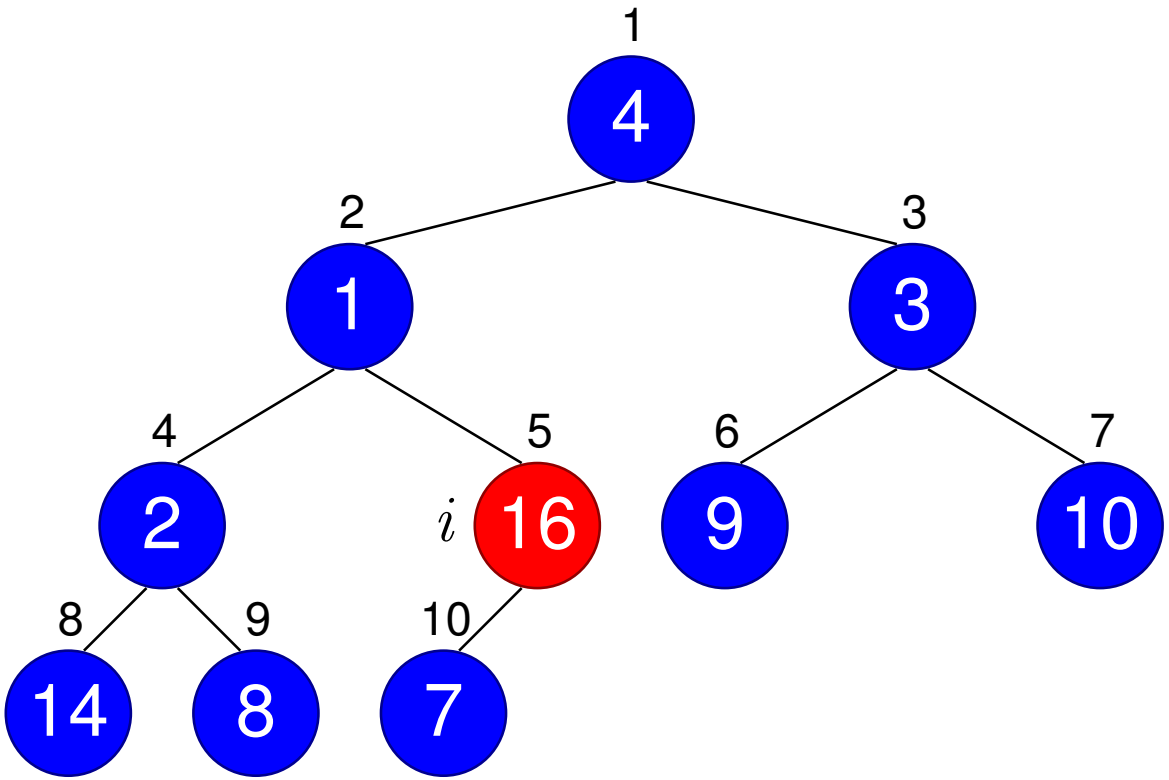
Aufgabe: Erstelle aus einem unsortierten Array einen Max-Heap

Idee: Bearbeite den Heap „von unten nach oben“ mittels

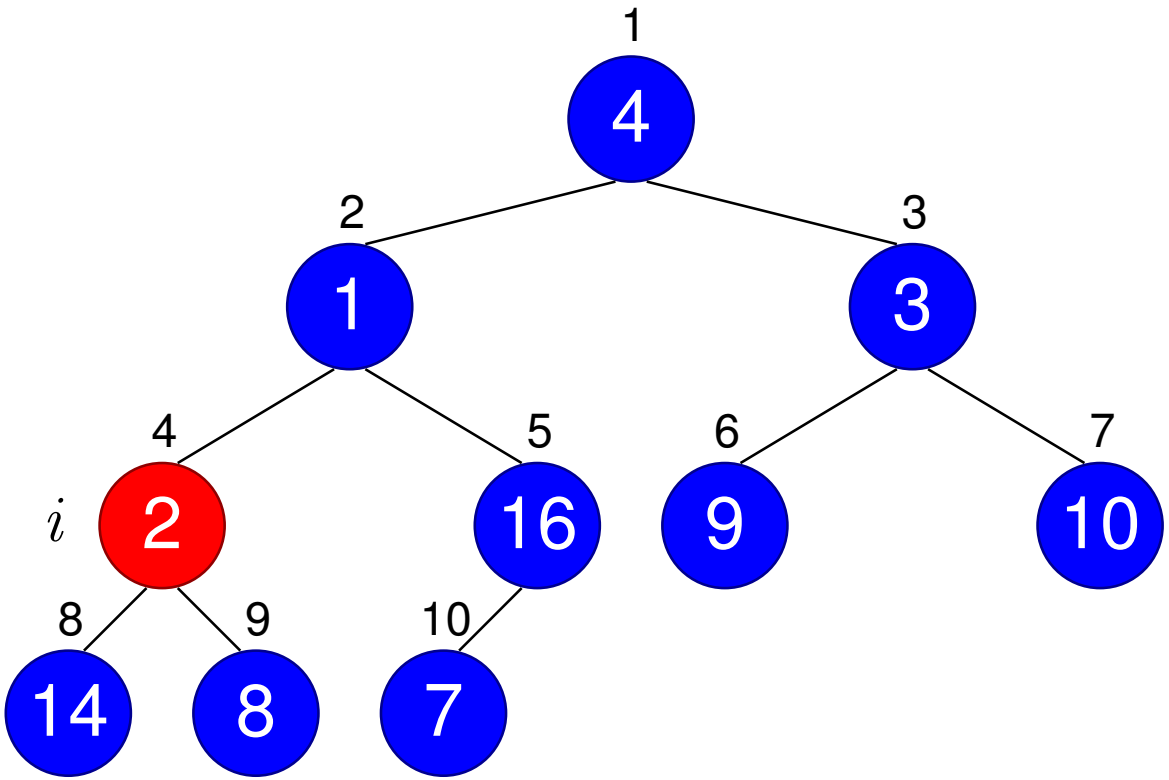
BuildMaxHeap(A)
Input: Array A
Output: Array A' mit Max-Heap Eigenschaft

```
1  heapsize(A) := length(A);
2  for i := ⌊length(A)/2⌋ downto 1 do
3     MaxHeapify(A, i);
```

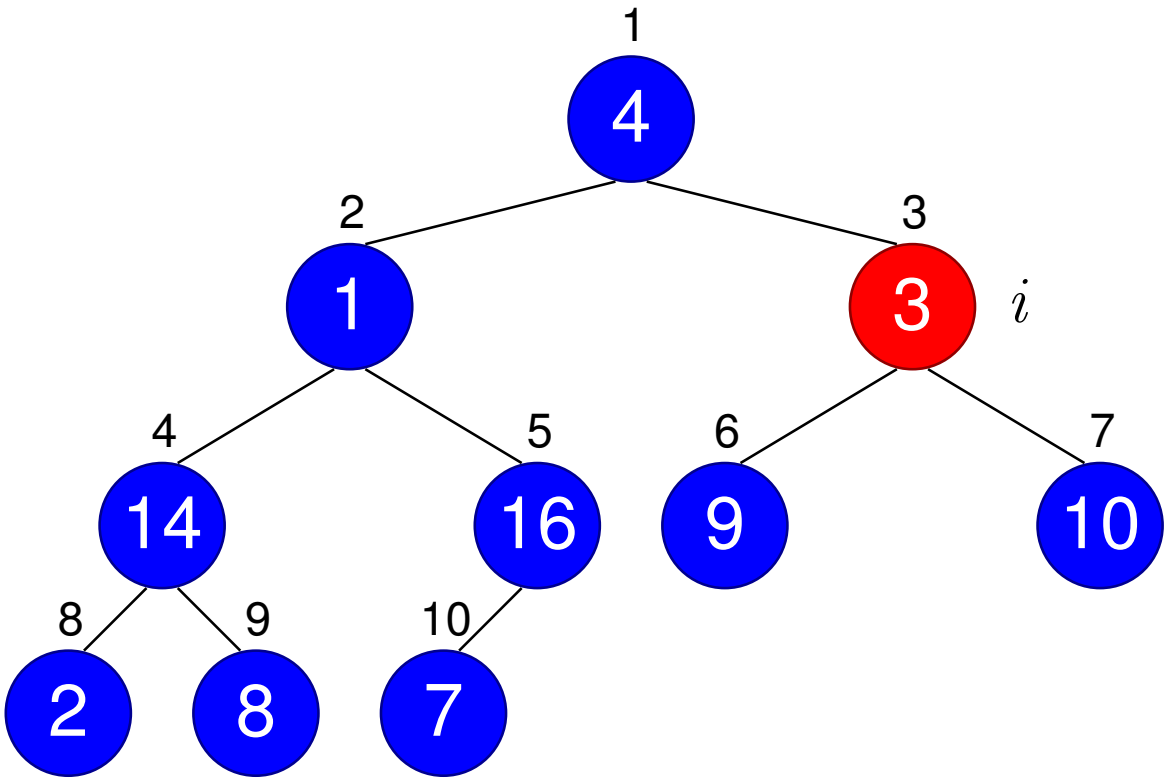
Max-Heap Konstruktion



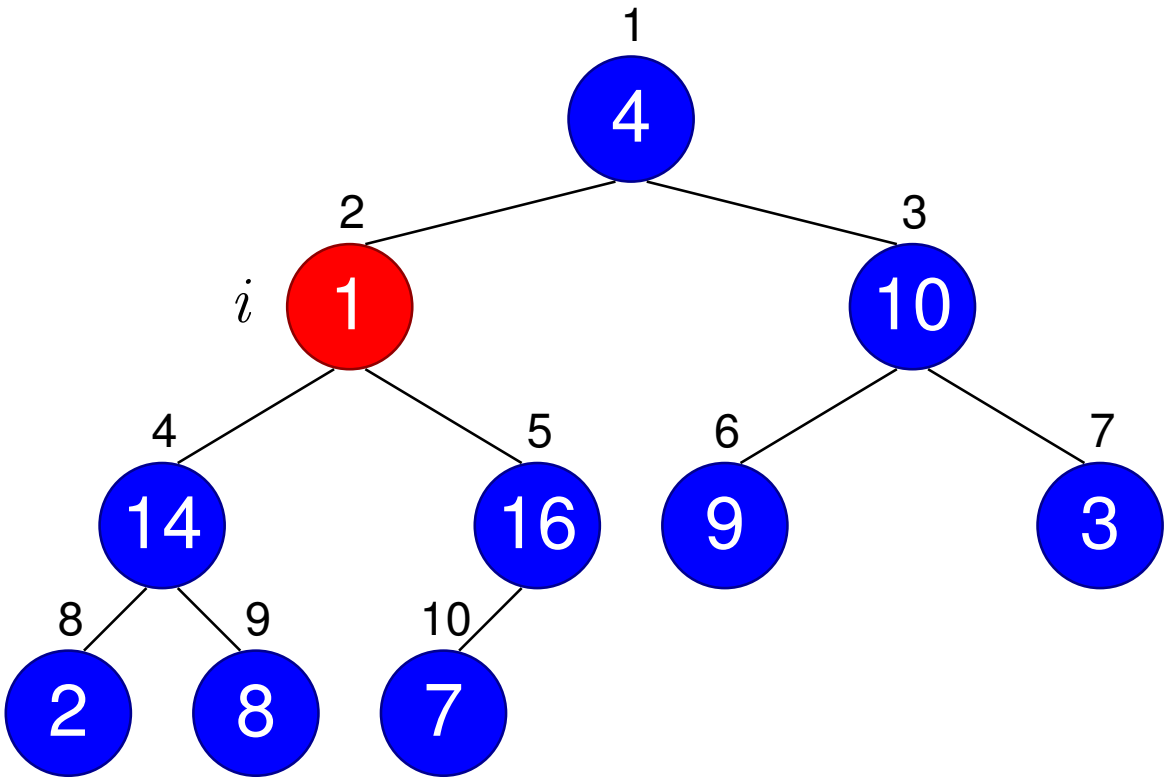
Max-Heap Konstruktion (Forts.)



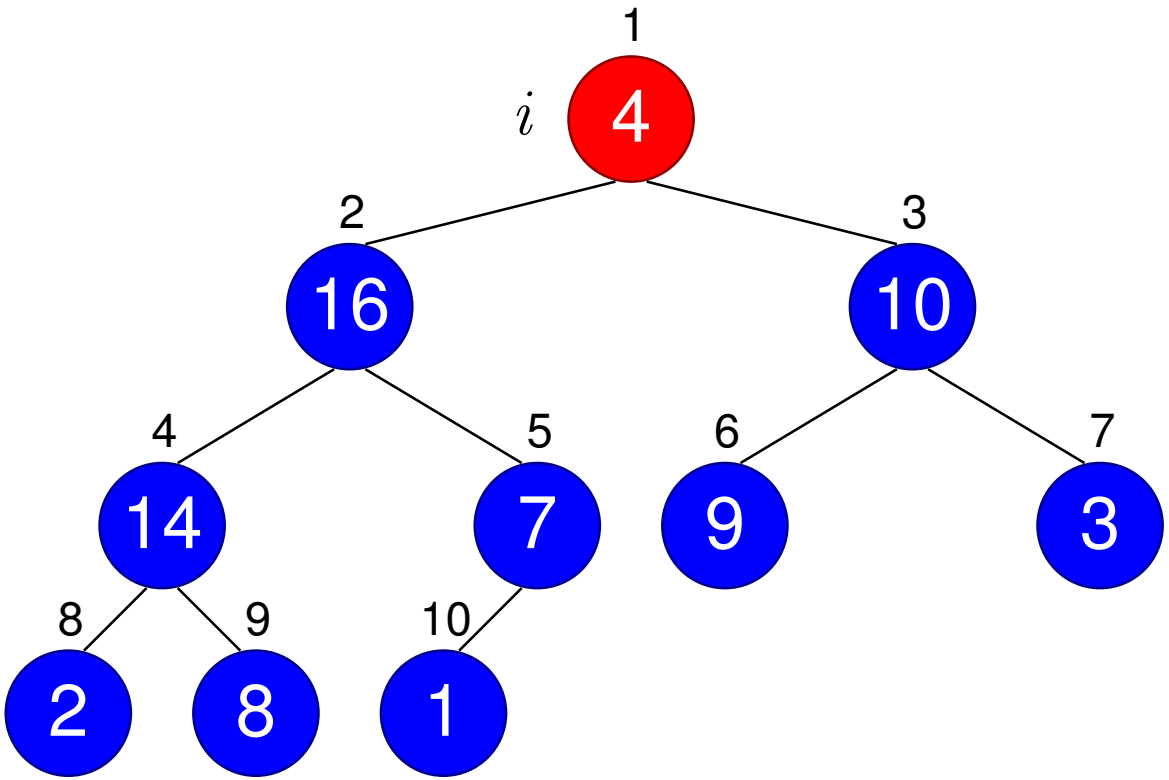
Max-Heap Konstruktion (Forts.)



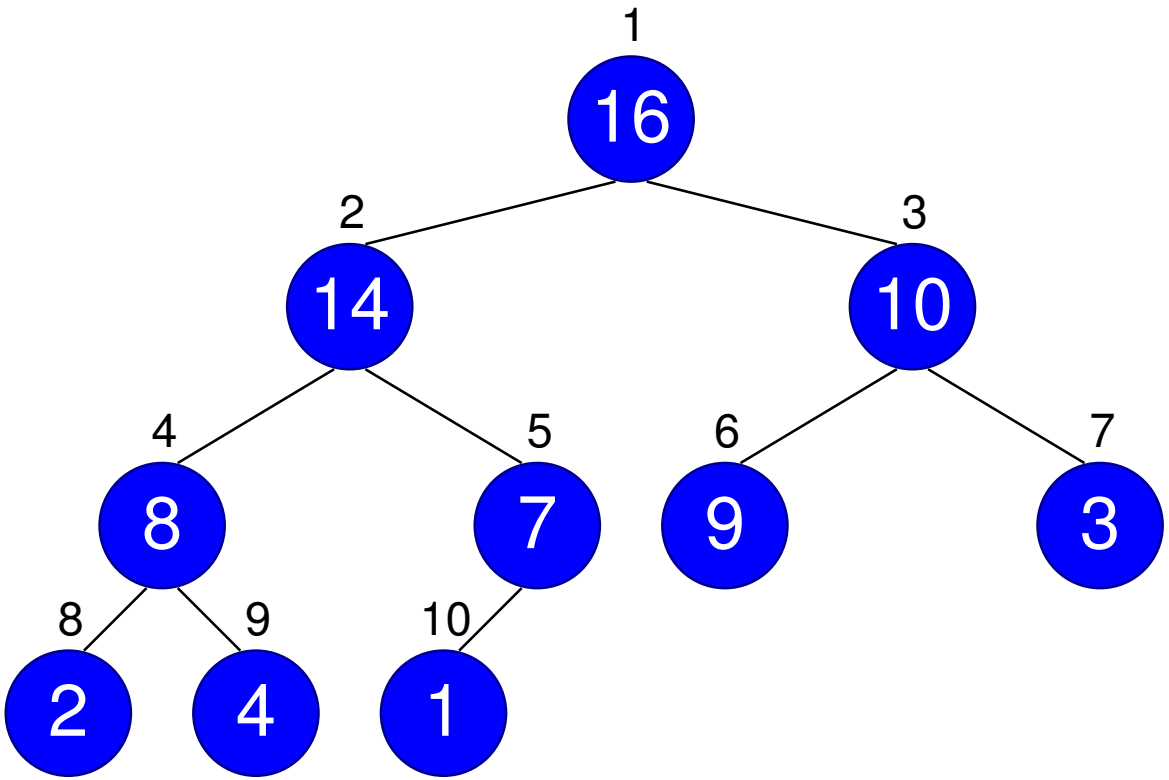
Max-Heap Konstruktion (Forts.)



Max-Heap Konstruktion (Forts.)



Max-Heap Konstruktion (Forts.)



Korrektheit von BuildMaxHeap(A)

Zu zeigen ist folgende Schleifeninvariante:

> Zu Beginn jeder Iteration der for-Schleife in Zeile 2–3 ist jeder der Knoten $i+1, i+2, \ldots, n$ die Wurzel eines Max-Heaps.

Initialisierung: Vor der ersten Iteration ist $i = \left\lfloor \frac{n}{2} \right\rfloor$. Die Bäume unterhalb von $i+1, i+2, \ldots, n$ sind Blätter und somit Max-Heaps.

Korrektheit (Forts.)

Aufrechterhaltung: Betrachte eine Iteration der for-Schleife.

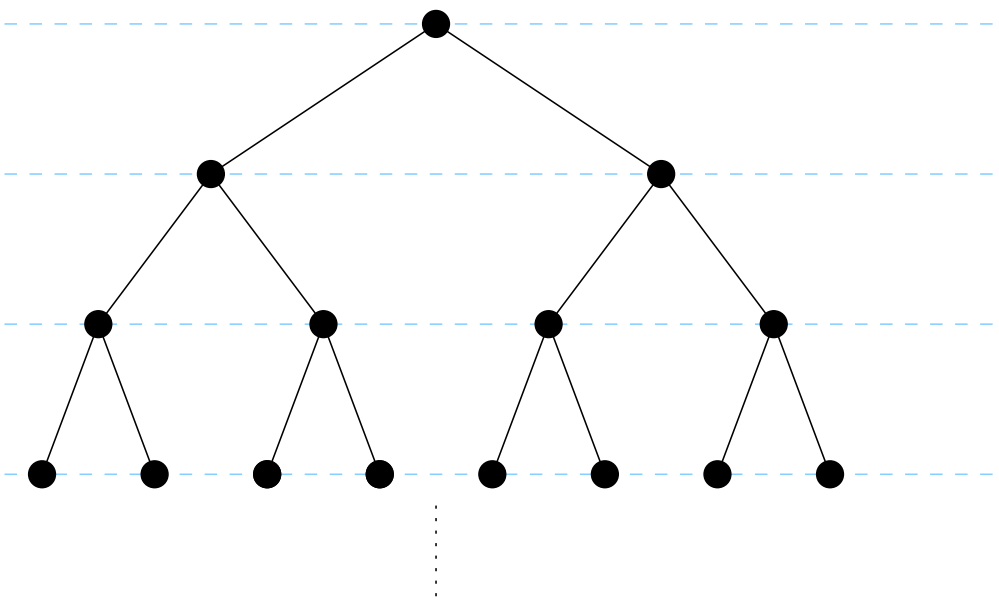
Die Kinder von i sind $left(i) = 2 \cdot i$ und $right(i) = 2 \cdot i + 1$. Laut Invariante sind dies Wurzeln von Max-Heaps.

Somit ist die Eingabe von $\textsc{MaxHeapify}(A, i)$ korrekt. Nach der Ausführung von $\textsc{MaxHeapify}$ ist i also die Wurzel eines Max-Heaps.

Beendigung: Am Ende ist $i = 0$. Somit ist jeder der Knoten $1, 2, \ldots, n$ (also insbesondere Knoten 1) die Wurzel eines Max-Heaps.

Laufzeit von BuildMaxHeap(A)

Beobachtung:



Tiefe	Knoten	Aufwand
0	2^0	$O(\log_2 n - 0)$
1	2^1	$O(\log_2 n - 1)$
2	2^2	$O(\log_2 n - 2)$
3	2^3	$O(\log_2 n - 3)$
i	2^i	$O(\log_2 n - i)$

Laufzeit (Forts.)

Bekannt:

- Ist t die Tiefe von i, dann ist die Laufzeit von $\textsc{MaxHeapify}(A, i)$ gleich $O(\log_2 n - t)$
- Ein balancierter Binärbaum mit n Elementen hat Tiefe $\lfloor \log_2 n \rfloor$
- Die Anzahl der Knoten auf Tiefe t ist 2^t

Abschätzen der Laufzeit durch Aufsummieren:

$$\sum_{t=0}^{\lfloor \log_2 n \rfloor} 2^t O(\log_2 n - t)$$

Laufzeit (Forts.)

Substitution: $h = \lfloor \log_2 n \rfloor - t$ bzw. $t = \lfloor \log_2 n \rfloor - h$

$$\begin{aligned}
\sum_{t=0}^{\lfloor \log_2 n \rfloor} 2^t O(\log_2 n - t) &= \sum_{h=\lfloor \log_2 n \rfloor}^{0} 2^{\lfloor \log_2 n \rfloor - h} O(h) \\
&= \sum_{h=0}^{\lfloor \log_2 n \rfloor} 2^{\lfloor \log_2 n \rfloor} \cdot 2^{-h} O(h) \\
&= 2^{\lfloor \log_2 n \rfloor} \sum_{h=0}^{\lfloor \log_2 n \rfloor} O\left(\frac{h}{2^h}\right) \\
&\leq n \sum_{h=0}^{\lfloor \log_2 n \rfloor} O\left(\frac{h}{2^h}\right)
\end{aligned}$$

Laufzeit (Forts.)

Bekannt: Für alle $|x| < 1$ gilt:

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

Setze $x = \frac{1}{2}$:

$$\sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{\left(\frac{1}{2}\right)^2} = 2$$

Laufzeit (Forts.)

Einsetzen:

$$n \sum_{h=0}^{\lfloor \log_2 n \rfloor} O\left(\frac{h}{2^h}\right) = nO\left(\sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) \leq nO(2) = O(n)$$

Fazit: Laufzeit von $\textsc{BuildMaxHeap}(\mathrm{A})$ ist $O(n)$

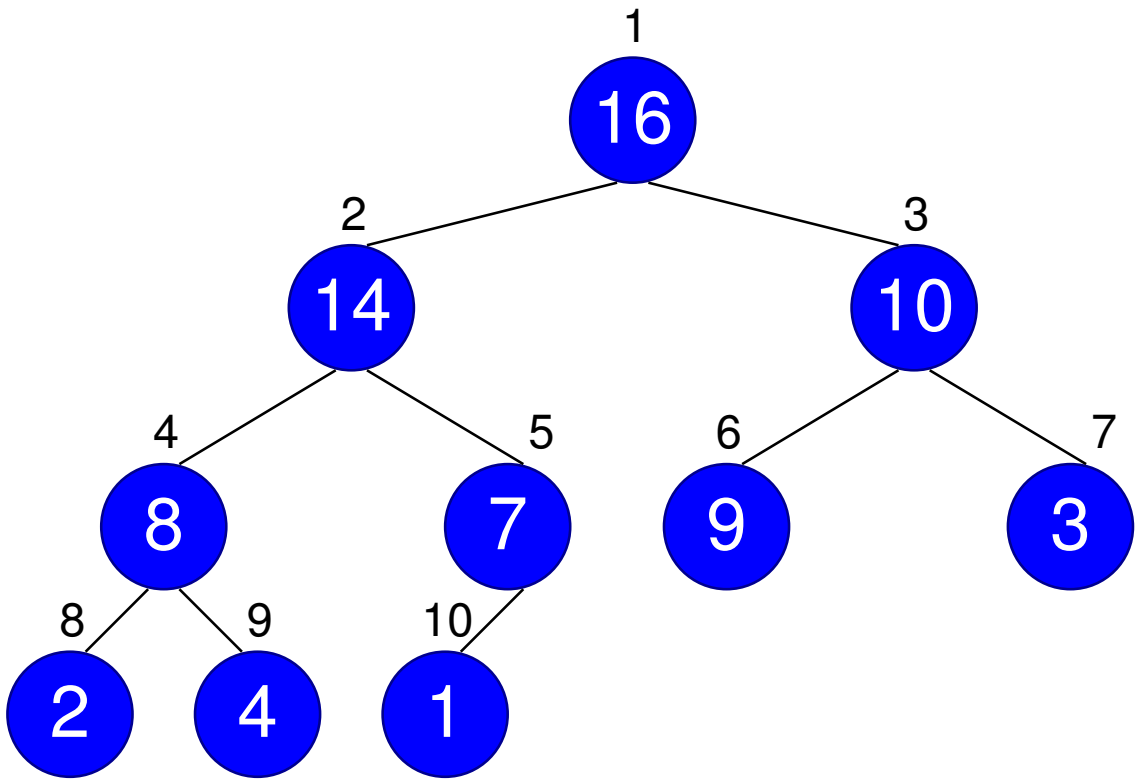
Heap Sort

HeapSort(A)
Input: Array A
Output: Array A in sortierter Form

```
1  BuildMaxHeap(A);
2  for i := length(A) downto 2 do
3     Vertausche A[1] und A[i];
4     heapsize(A) := heapsize(A) − 1;
5     MaxHeapify(A, 1);
```

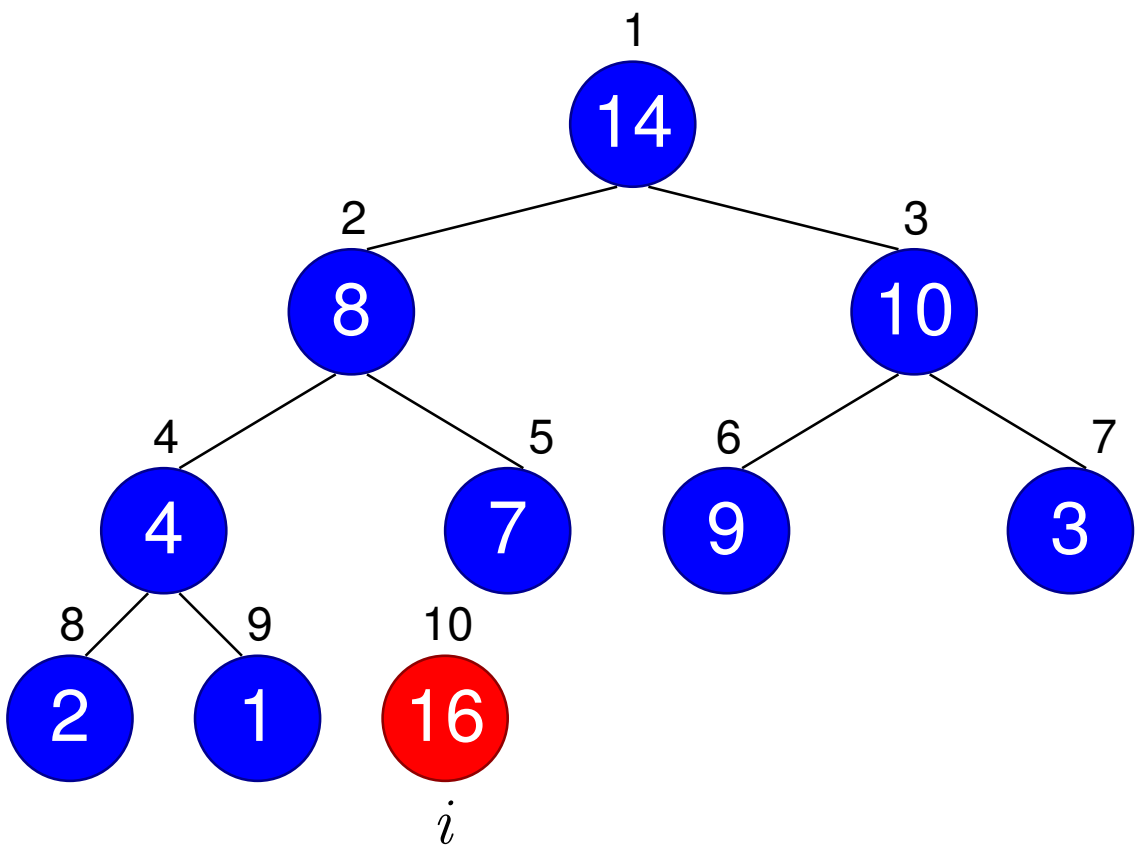
Beispiel für Heap Sort

Zustand des Heaps nach Beendigung von BuildMaxHeap:



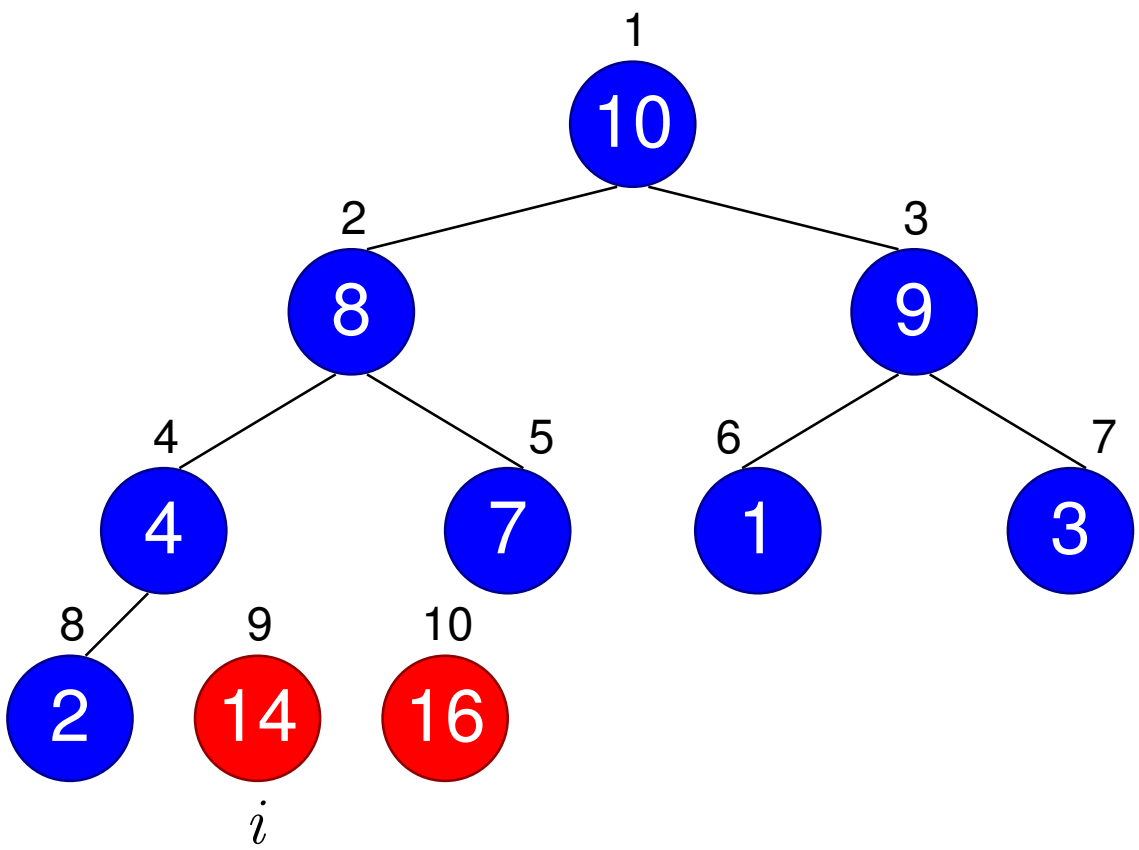
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 1. Schleifendurchlauf:



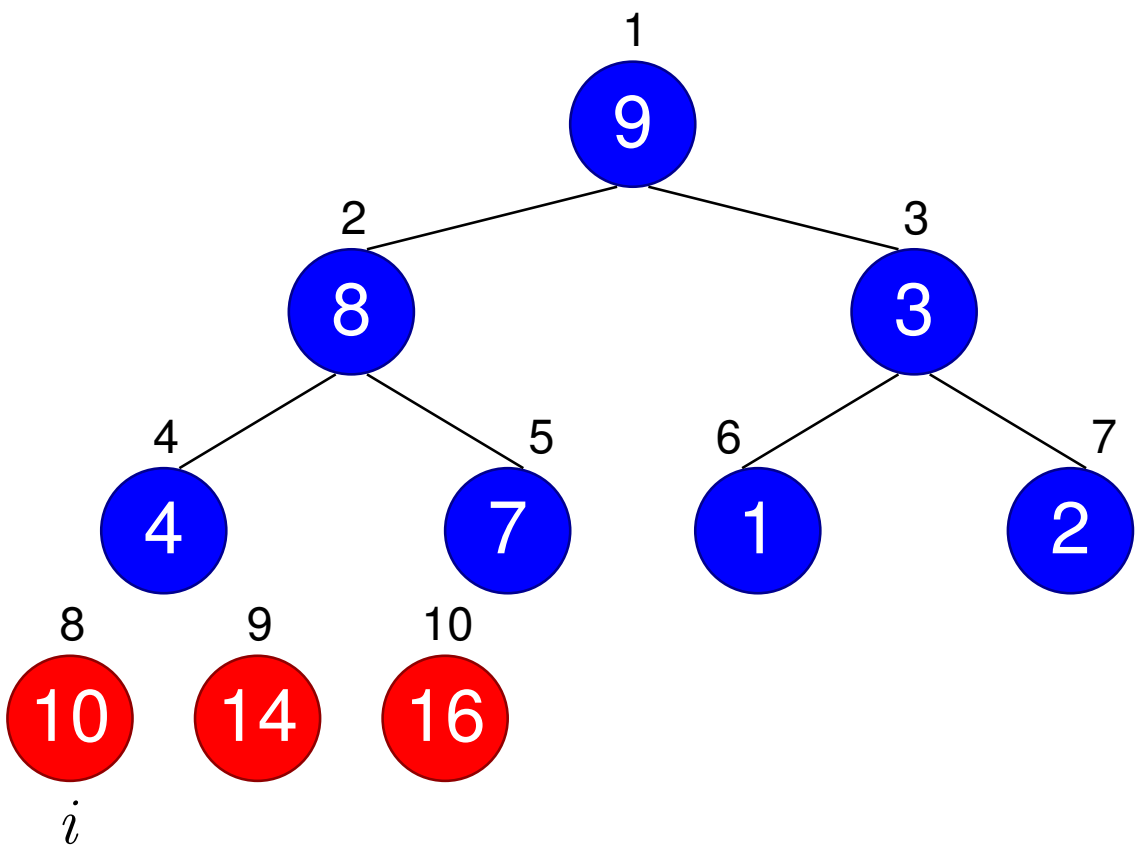
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 2. Schleifendurchlauf:



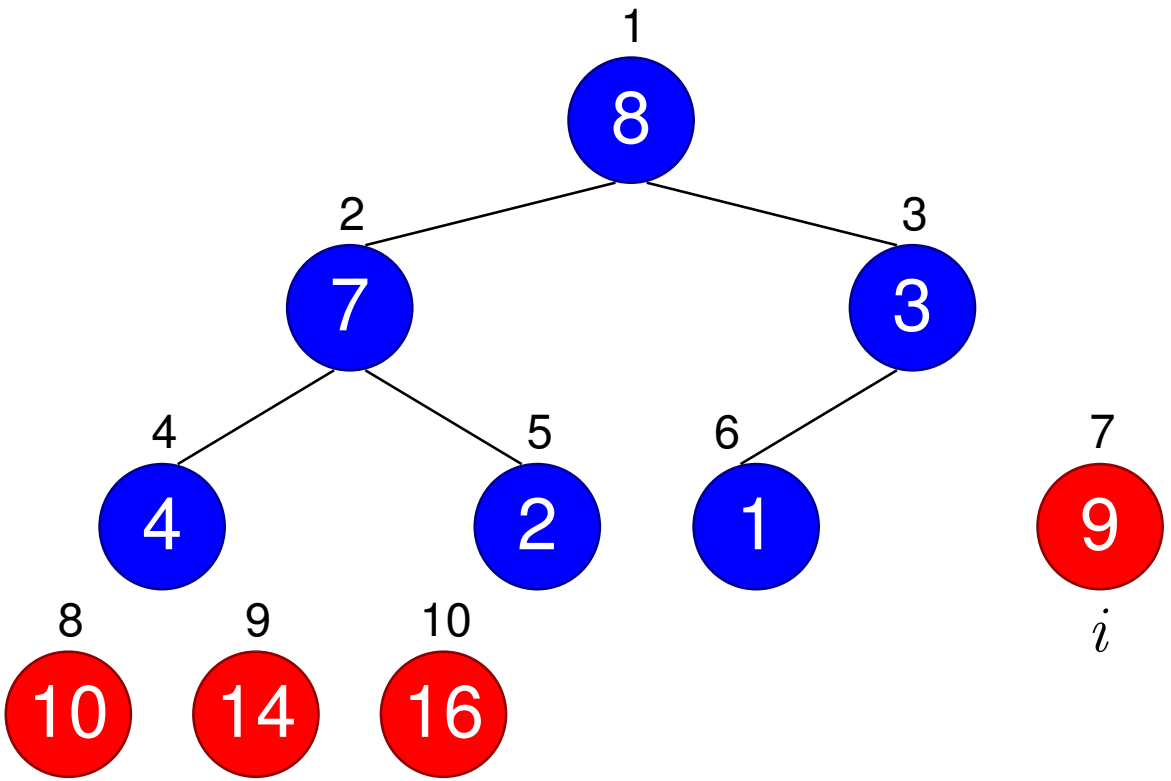
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 3. Schleifendurchlauf:



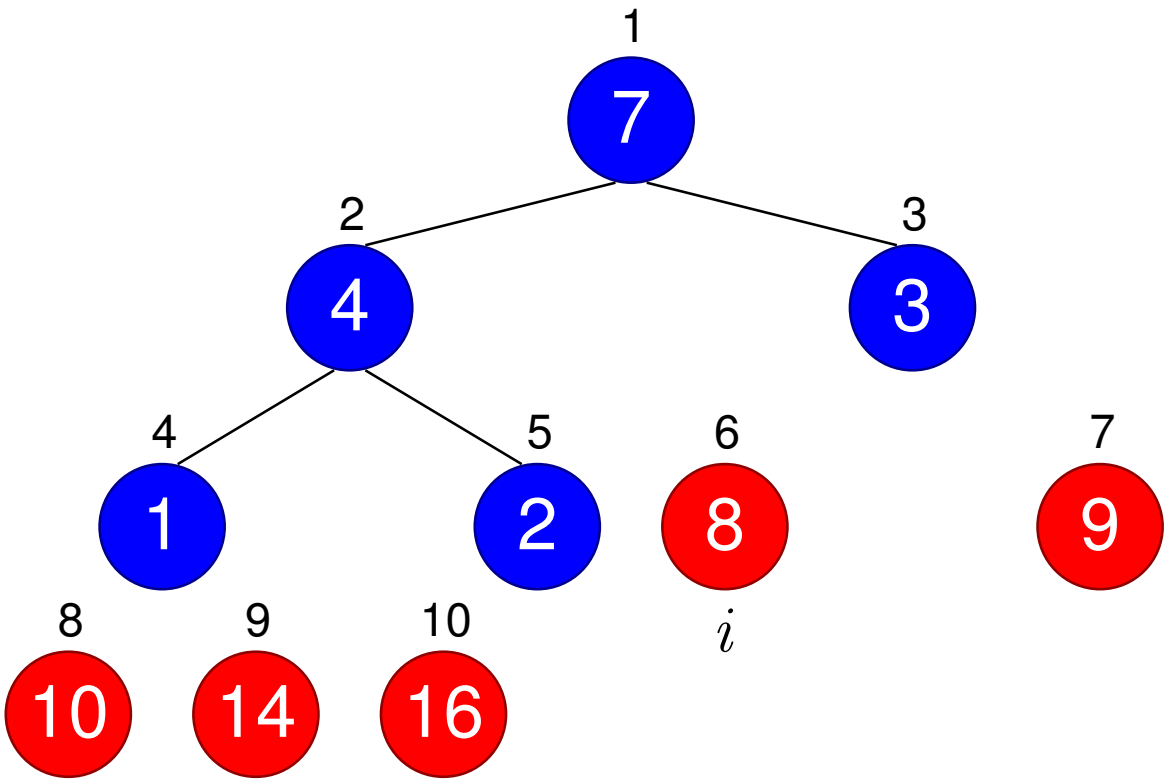
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 4. Schleifendurchlauf:



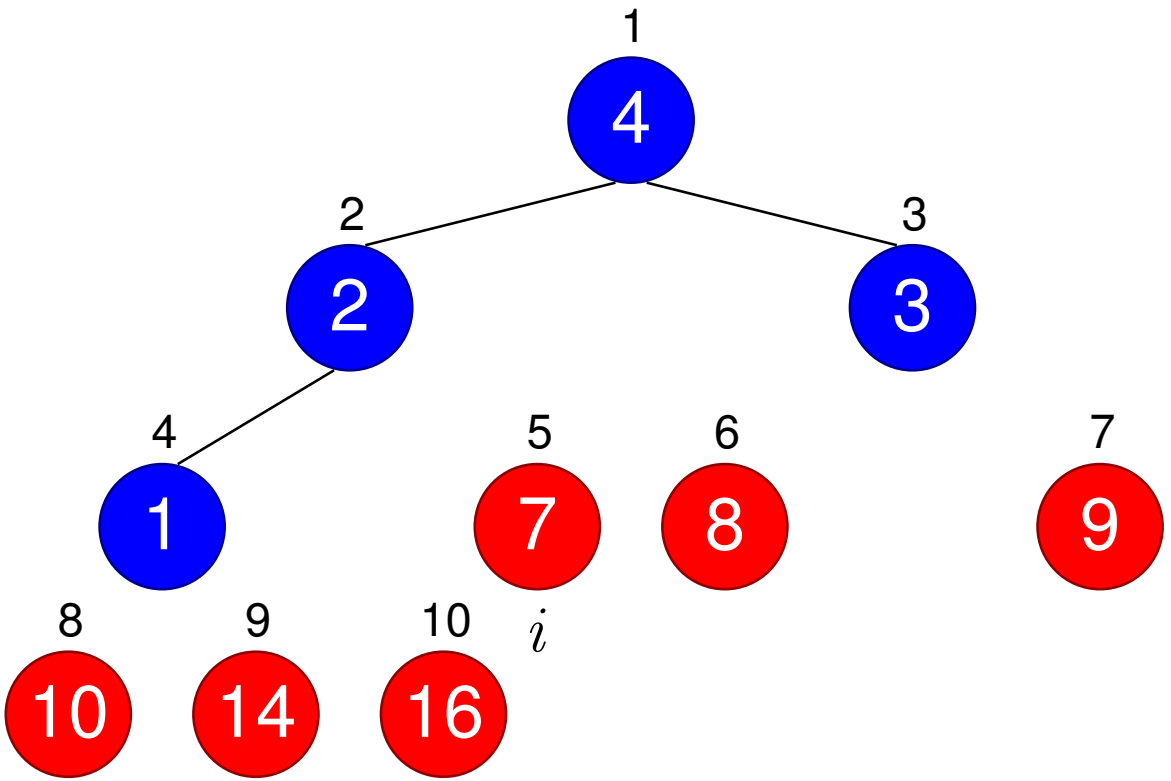
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 5. Schleifendurchlauf:



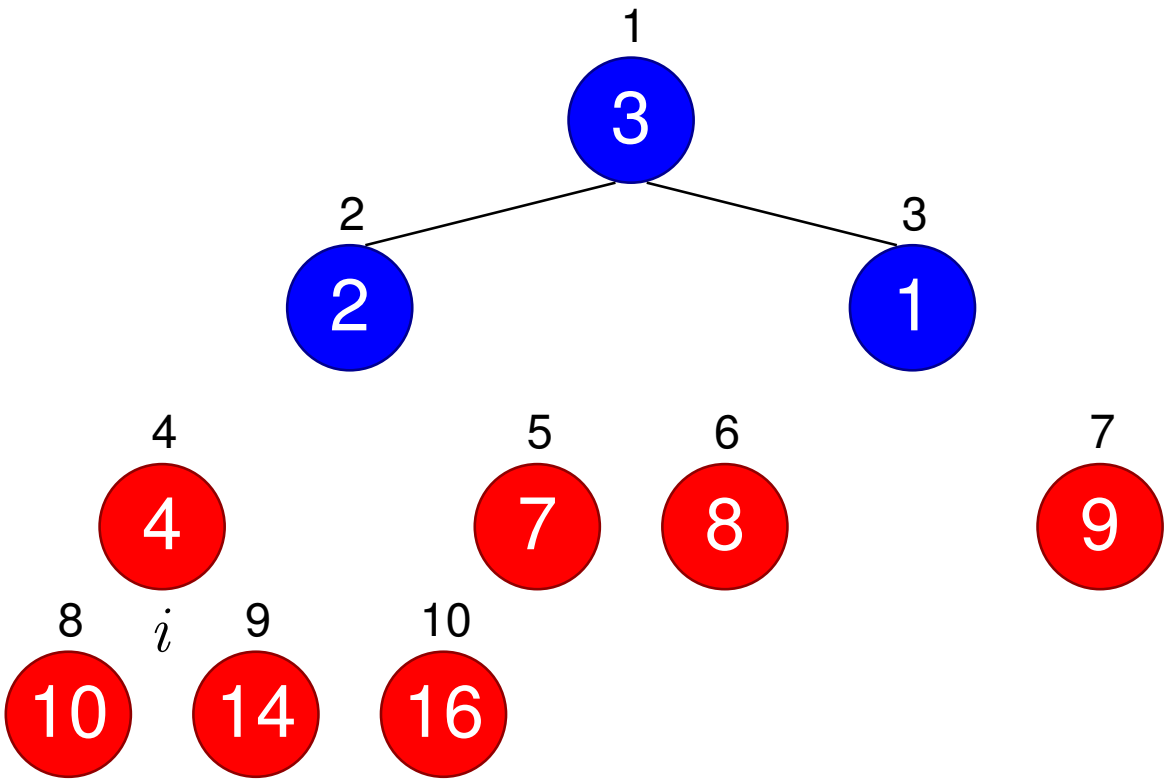
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 6. Schleifendurchlauf:



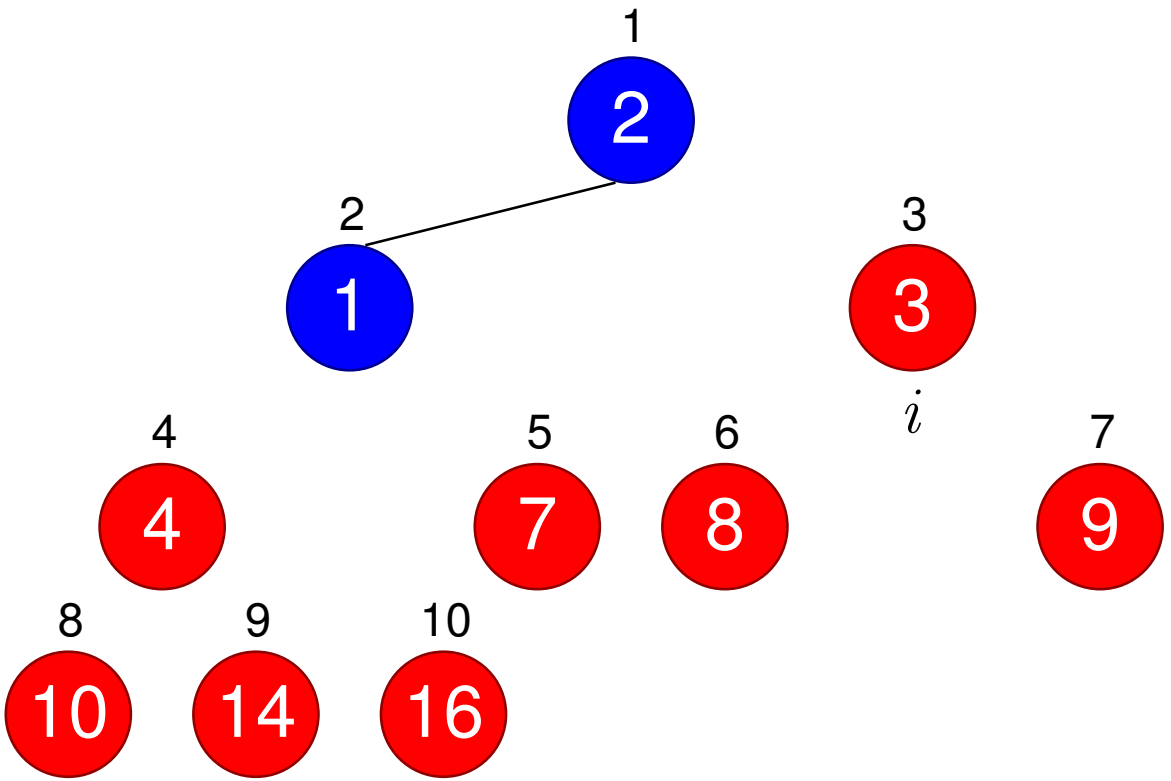
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 7. Schleifendurchlauf:



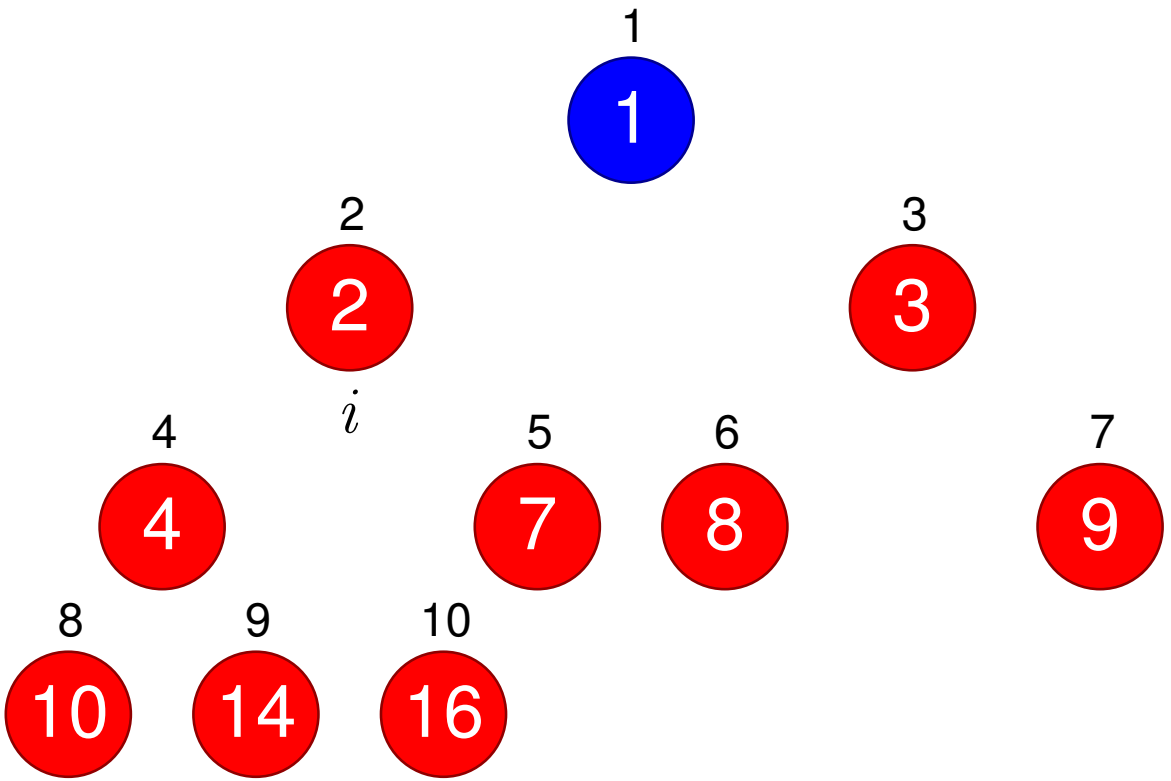
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 8. Schleifendurchlauf:



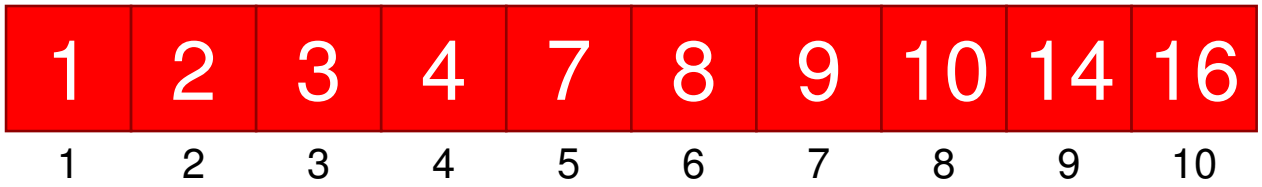
Beispiel für Heap Sort (Forts.)

Zustand des Heaps nach dem 9. Schleifendurchlauf:



Beispiel für Heap Sort (Forts.)

Ergebnis:



Korrektheit von Heap Sort

Schleifeninvariante: Zu Beginn jeder Iteration der for-Schleife in Zeile 2–5 gilt:

- Das Teilarray $A[i+1..n]$ enthält die $n-i$ größten Elemente von $A[1..n]$ in sortierter Form.
- Das Teilarray $A[1..i]$ ist ein Max-Heap und enthält die i kleinsten Elemente von $A[1..n]$.

Initialisierung: Vor der ersten Iteration der for-Schleife ist $i = n = length(A)$.

- Das Teilarray $A[n+1..n]$ ist leer und somit trivialerweise sortiert.
- Da in Zeile 1 $\textsc{BuildMaxHeap}(A)$ aufgerufen wurde, ist $A[1..n]$ ein Max-Heap.

Korrektheit von Heap Sort (Forts.)

Aufrechterhaltung: Betrachte den Zustand des Algorithmus vor Schleifendurchlauf $i-1$. Wegen der Schleifeninvariante gilt:

- Da $A[1..i-1]$ ein Max-Heap ist, ist $A[1]$ ist das größte Element in $A[1..i-1]$.
- $A[1]$ ist kleiner als jedes Element in $A[i..n]$.

Hieraus folgt:

- Nach Vertauschen von $A[1]$ und $A[i-1]$ in Zeile 3 gilt, dass $A[i-1..n]$ die $n-i+1$ größten Elemente von A in sortierter Form enthält.

Korrektheit von Heap Sort (Forts.)

- Nach Ausführung der Zeilen 4 ist $heapsize(A) = i - 2$. Der Aufruf von MaxHeapify garantiert, dass $A[1..i-2]$ ein Max-Heap ist.

Beendigung: Nach Beendigung der for-Schleife ist $i = 1$. Es gilt:

- $A[2..n]$ enthält die $n - 1$ größten (also alle) Elemente von A in sortierter Form.
- $A[1..1]$ ist ein Max-Heap und enthält das kleinste Element von A als Wurzel.

Fazit: $A[1..n]$ enthält die ursprünglichen Elemente in sortierter Form. Also ist die Arbeitsweise von Heap Sort korrekt.

Laufzeit von Heap Sort

Die Laufzeit von Heap Sort für ein Array A der Länge n setzt sich zusammen aus:

1. Ausführung von $\textsc{BuildMaxHeap}(A)$: Laufzeit $O(n)$
2. Durchlaufen der for-Schleife, d.h., n-malige Ausführung von $\textsc{MaxHeapify}(\mathrm{A},1)$: Laufzeit $O(n \log_2 n)$

Ergebnis: Die Laufzeit von Heap Sort ist $O(n \log_2 n)$.

Priority Queues

- Datenstruktur zum Speichern einer Menge S von Datensätzen
- Jedem Element x ist ein Schlüssel $k = key(x)$ zugeordnet
- Operationen:
  - $\textsc{Insert}(S, x) \rightsquigarrow x$ in die Menge S einfügen
  - $\textsc{Maximum}(S) \rightsquigarrow$ Element mit dem größten Schlüssel ermitteln
  - $\textsc{ExtractMax}(S) \rightsquigarrow$ Element mit dem größten Schlüssel ermitteln und aus S löschen
  - $\textsc{IncreaseKey}(S, x, k) \rightsquigarrow$ Den Schlüssel des Elements x auf k erhöhen, wobei $k > key(x)$
- Analog für minimale Schlüssel

Einsatz eines Max-Heaps

- Eine Priority Queue kann mittels eines Max-Heaps A implementiert werden
- Erweiterung: Neben den Schlüsseln müssen auch die Satellitendaten im Array gespeichert werden
- Vorteil: Laufzeit der Queue Operationen ist höchstens linear in der Tiefe des Heaps
- Nachteil: ein Heap hat eine maximale Kapazität

Auslesen des Maximums

```
MAXIMUM(A)
1   return A[1]
```

- Da bei einem Max-Heap das größte Element in der Wurzel des Heaps gespeichert wird, liefert $\textsc{Maximum}(S)$ das korrekte Ergebnis
- Die Laufzeit ist $O(1)$, also konstant unabhängig von der Größe des Heaps

Auslesen und Löschen des Maximums

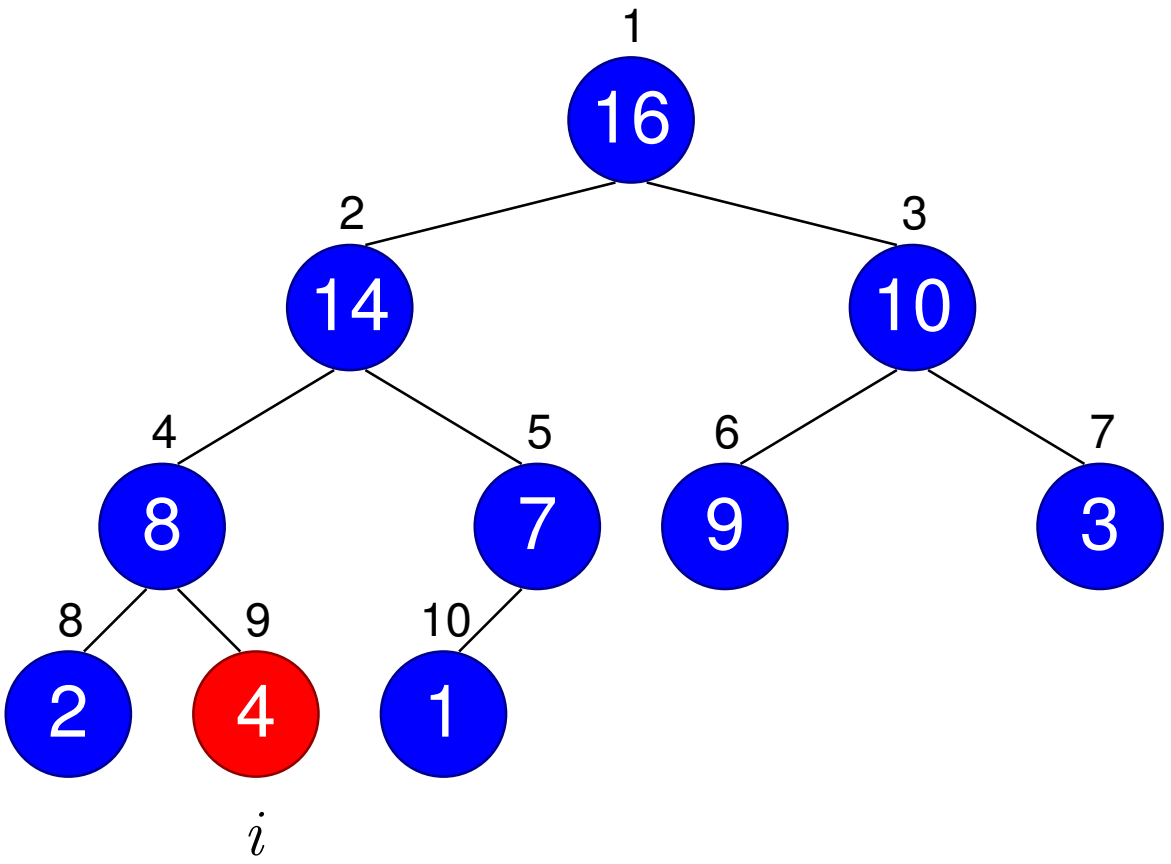
HeapExtractMax(A)

```
1  if heapsize(A) < 1 then
2     error "Heap underflow"
3  max := A[1]
4  A[1] := A[heapsize(A)]
5  heapsize(A) := heapsize(A) − 1
6  MaxHeapify(A, 1)
7  return max
```

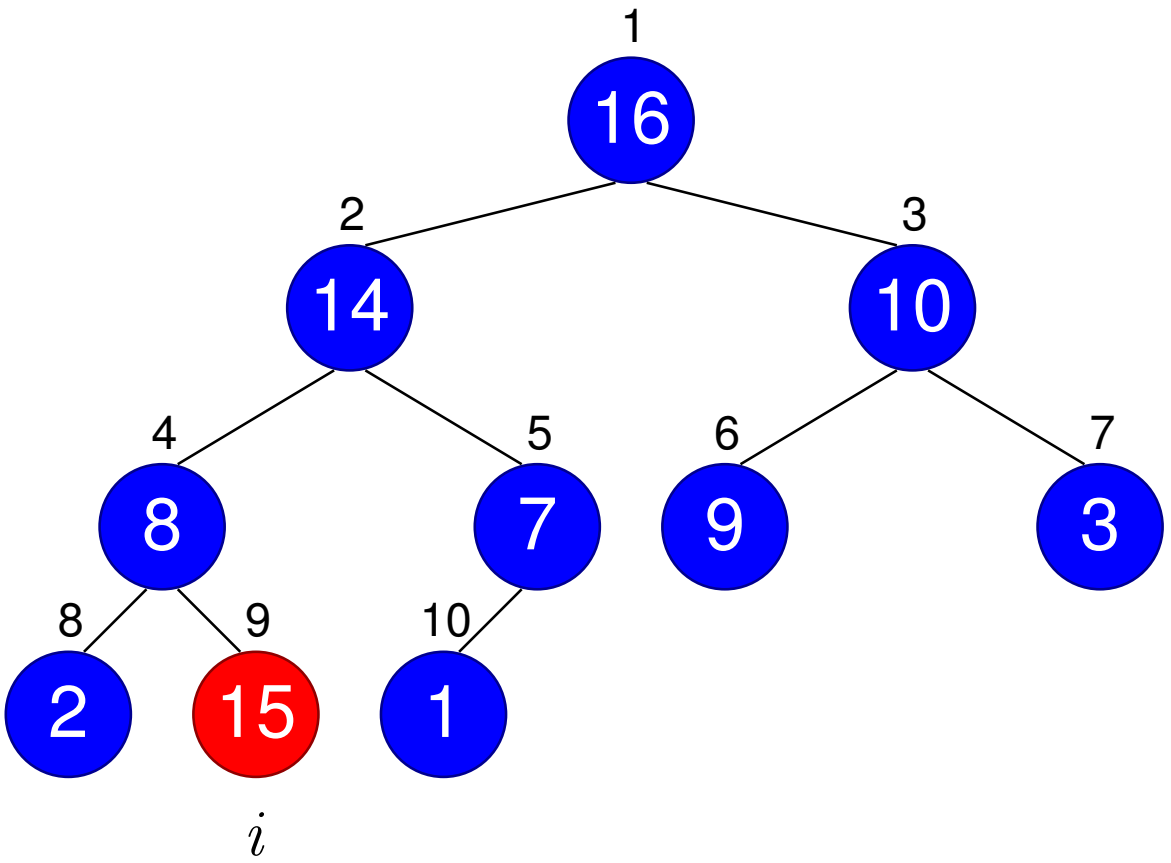
- Es wird das Maximum von A zurückgegeben
- In Zeilen 4–6 wird das Element gelöscht und die Max-Heap Eigenschaft von A wieder hergestellt
- Die Laufzeit ist $O(\log_2 n)$

Erhöhen des Schlüssels eines Elements (Idee)

Erhöhen des Schlüssels von Element $i = 9$ auf den Wert 15:

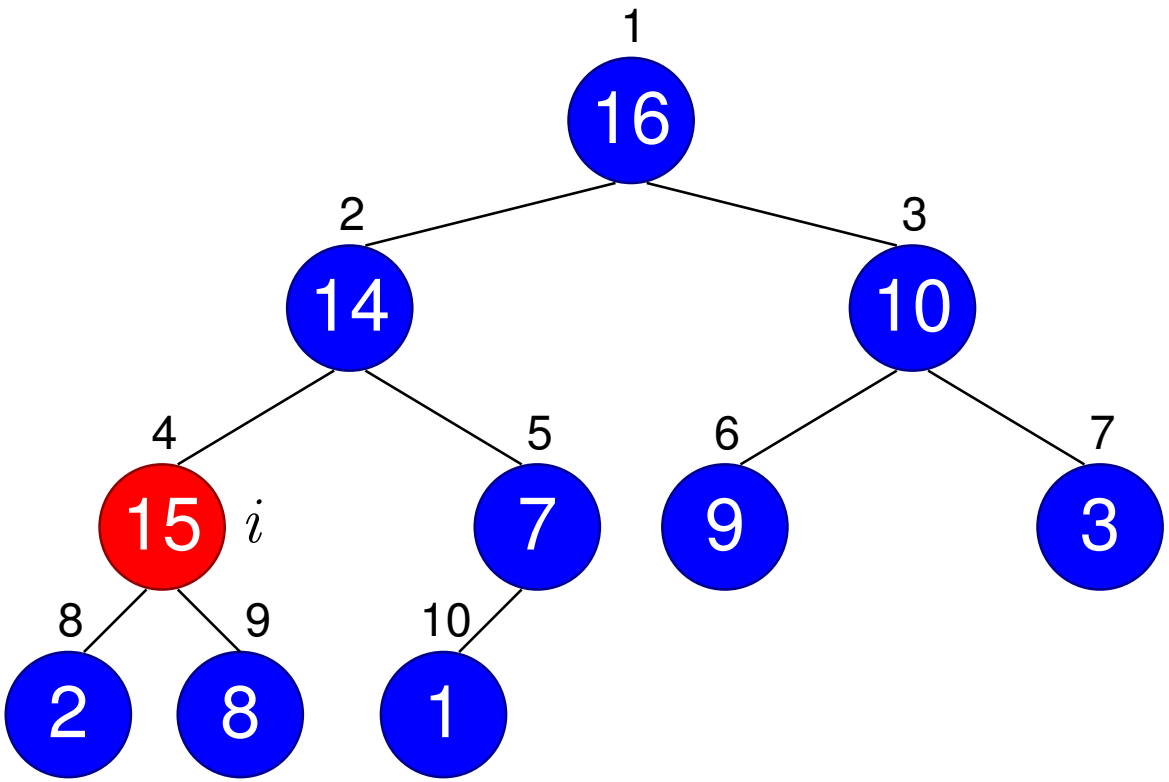


Erhöhen des Schlüssels eines Elements (Idee)



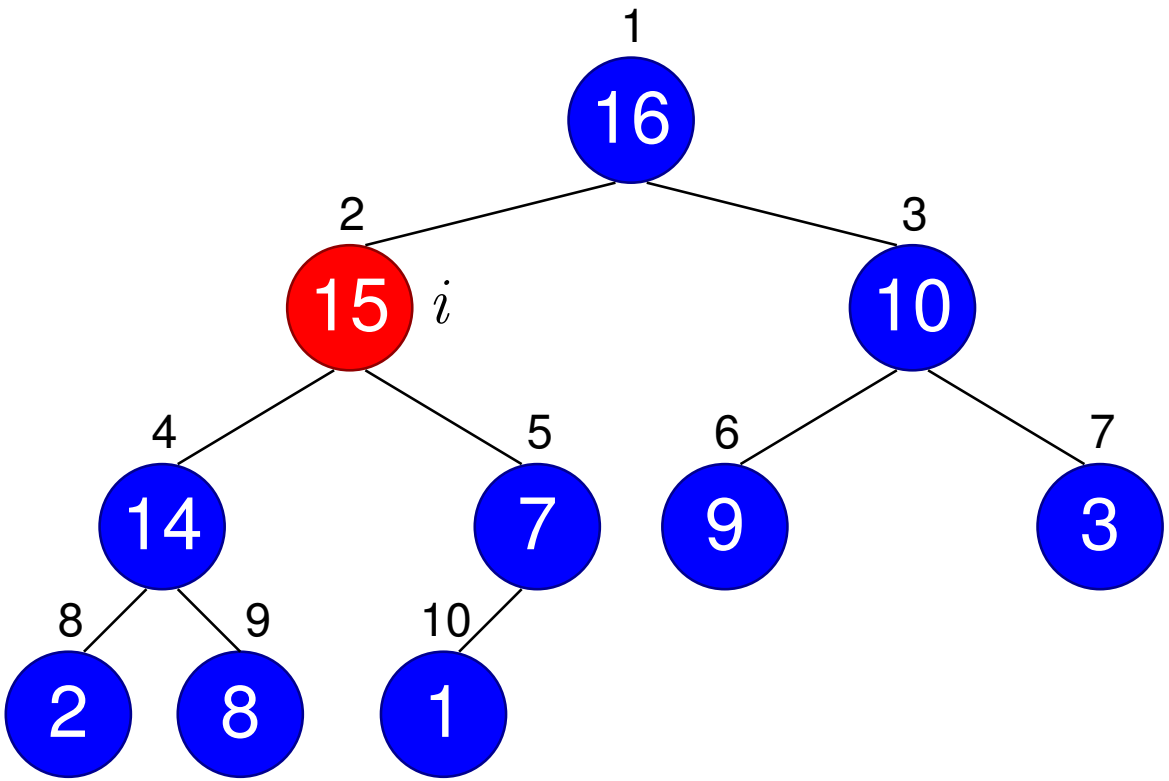
Erhöhen des Schlüssels eines Elements (Idee)

Das Element wandert nach oben ...



Erhöhen des Schlüssels eines Elements (Idee)

. . . bis die Max-Heap Eigenschaft wieder hergestellt ist



Erhöhen des Schlüssels eines Elements

HeapIncreaseKey(A, i, key)

```
1  if key < A[i] then
2     error "new key is smaller than old key"
3  A[i] := key
4  while i > 1 and A[parent(i)] < A[i] do
5     Vertausche A[i] und A[parent(i)]
6     i := parent(i)
```

- Das Element i mit dem vergrößerten Schlüssel wandert solange nach oben, bis die Max-Heap Eigenschaft wieder hergestellt ist
- Die Laufzeit ist linear in der Tiefe des Heaps, also $O(\log_2 n)$

Einfügen eines Elements

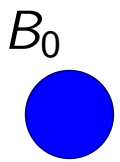
$\textsc{MaxHeapInsert}(A, key)$

1 $heapsize(A) := heapsize(A) + 1$
2 $A[heapsize(A)] := -\infty$
3 $\textsc{HeapIncreaseKey}(A, heapsize(A), key)$

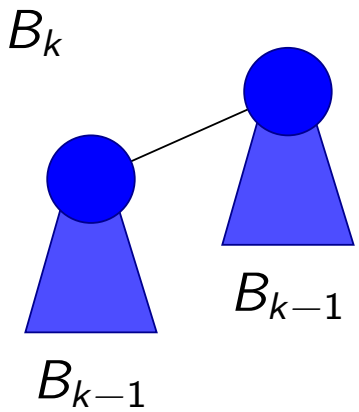
- $A[heapsize(A)]$ wird vorübergehend auf $-\infty$ gesetzt, damit der Aufruf von $\textsc{HeapIncreaseKey}$ keinen Fehler liefert
- Die Laufzeit ist $O(\log_2 n)$

Binomialbäume

- Der Binomialbaum B_k ist ein geordneter Baum
- Der Aufbau von B_k ist rekursiv definiert:
  - B_0 besteht aus einem Knoten



  - B_k besteht aus zwei Binomialbäumen B_{k-1}, wobei der eine links unterhalb des anderen hängt

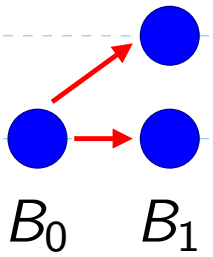


- B_k hat die Tiefe k

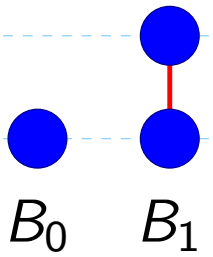
Binomialbäume Beispiel

B_0

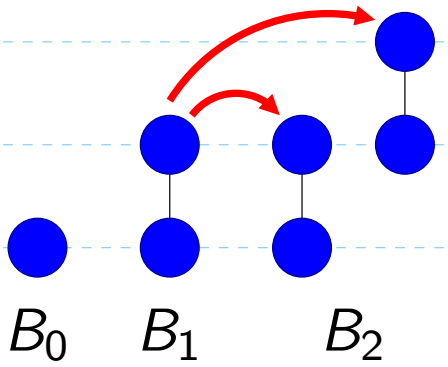
Binomialbäume Beispiel (Forts.)



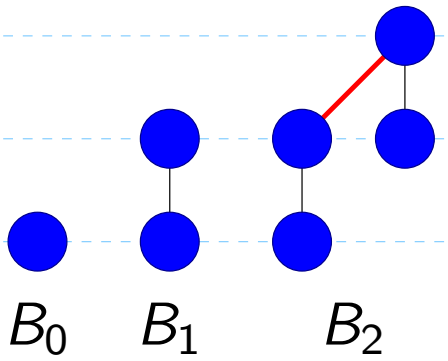
Binomialbäume Beispiel (Forts.)



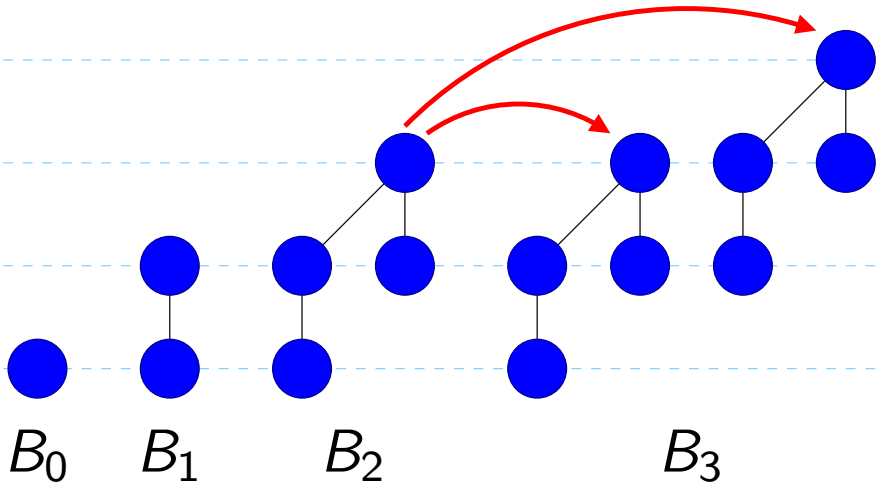
Binomialbäume Beispiel (Forts.)



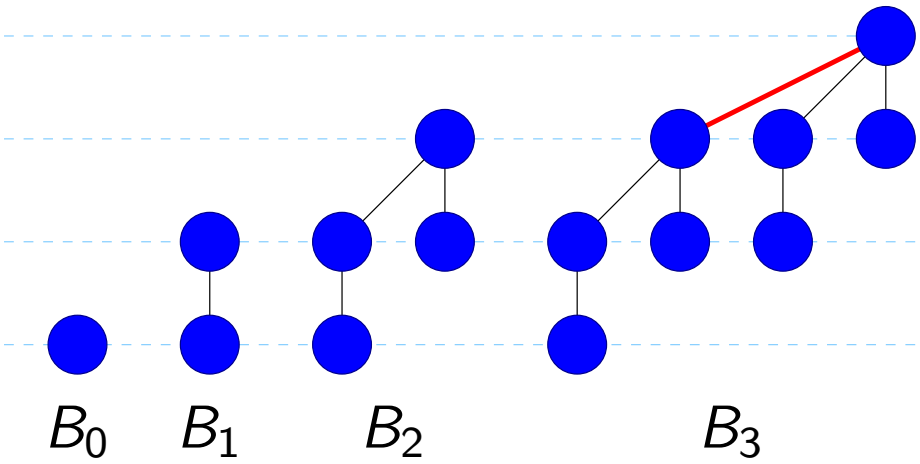
Binomialbäume Beispiel (Forts.)



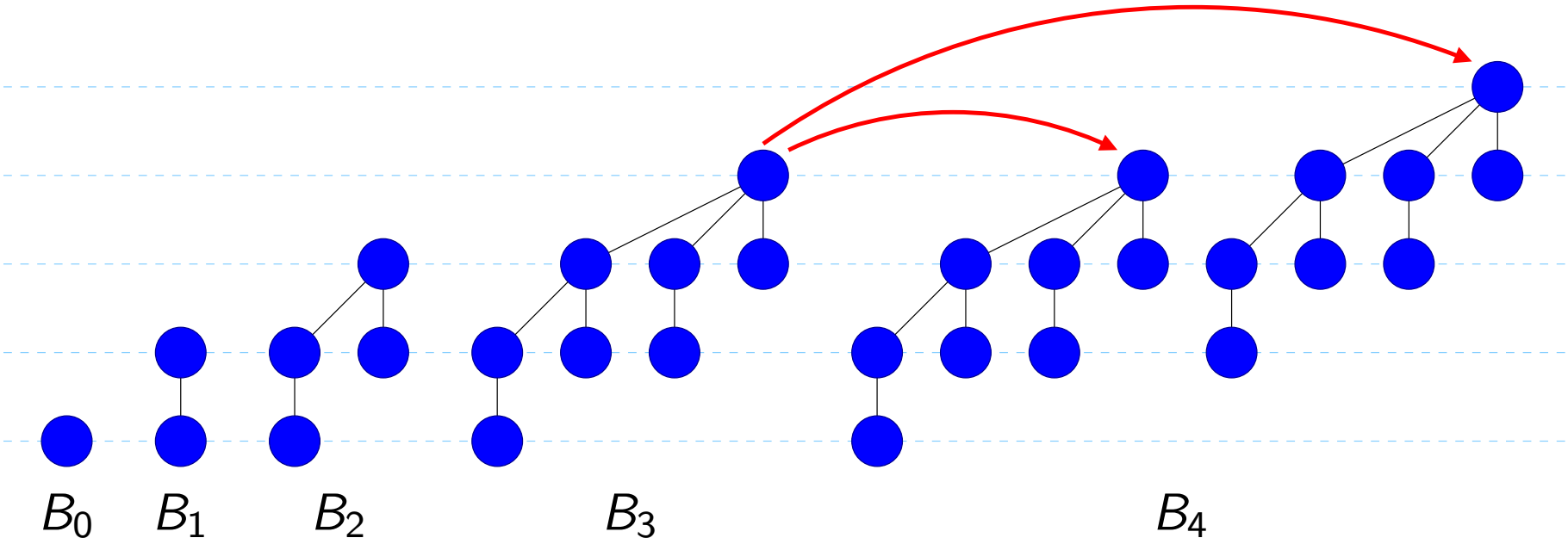
Binomialbäume Beispiel (Forts.)



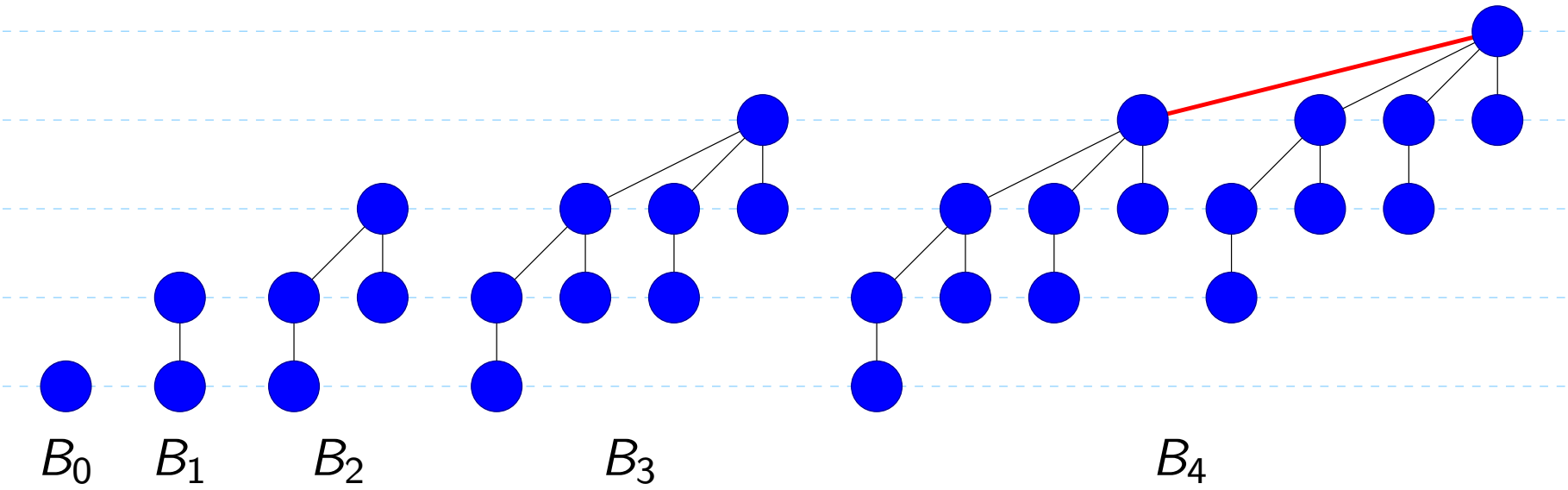
Binomialbäume Beispiel (Forts.)



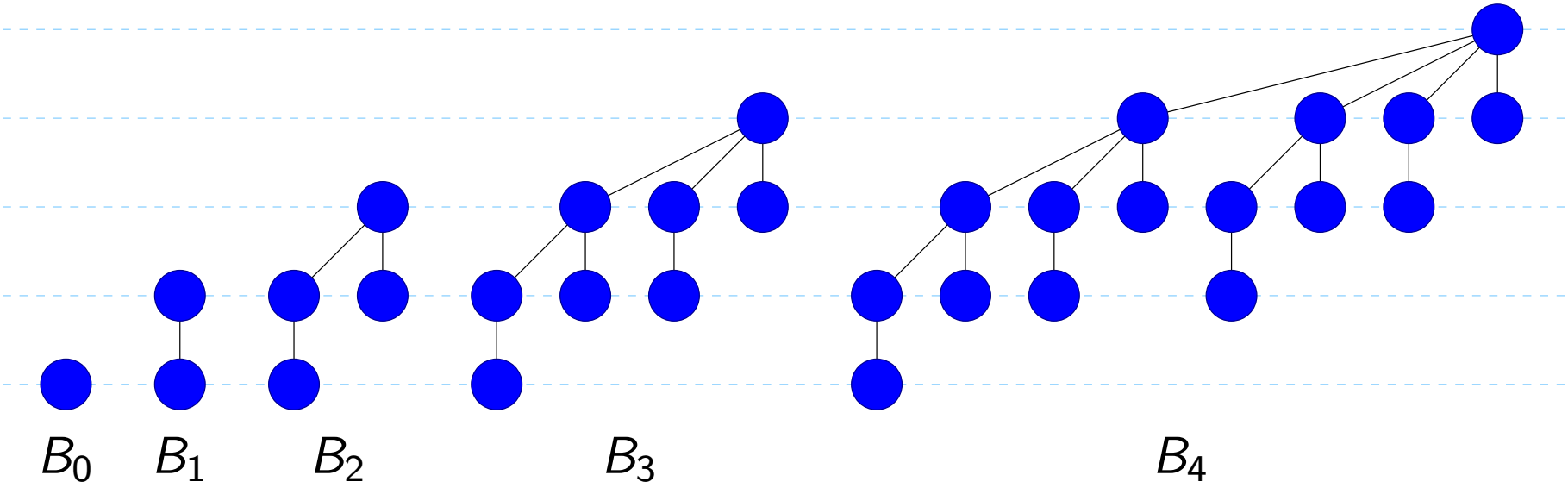
Binomialbäume Beispiel (Forts.)



Binomialbäume Beispiel (Forts.)

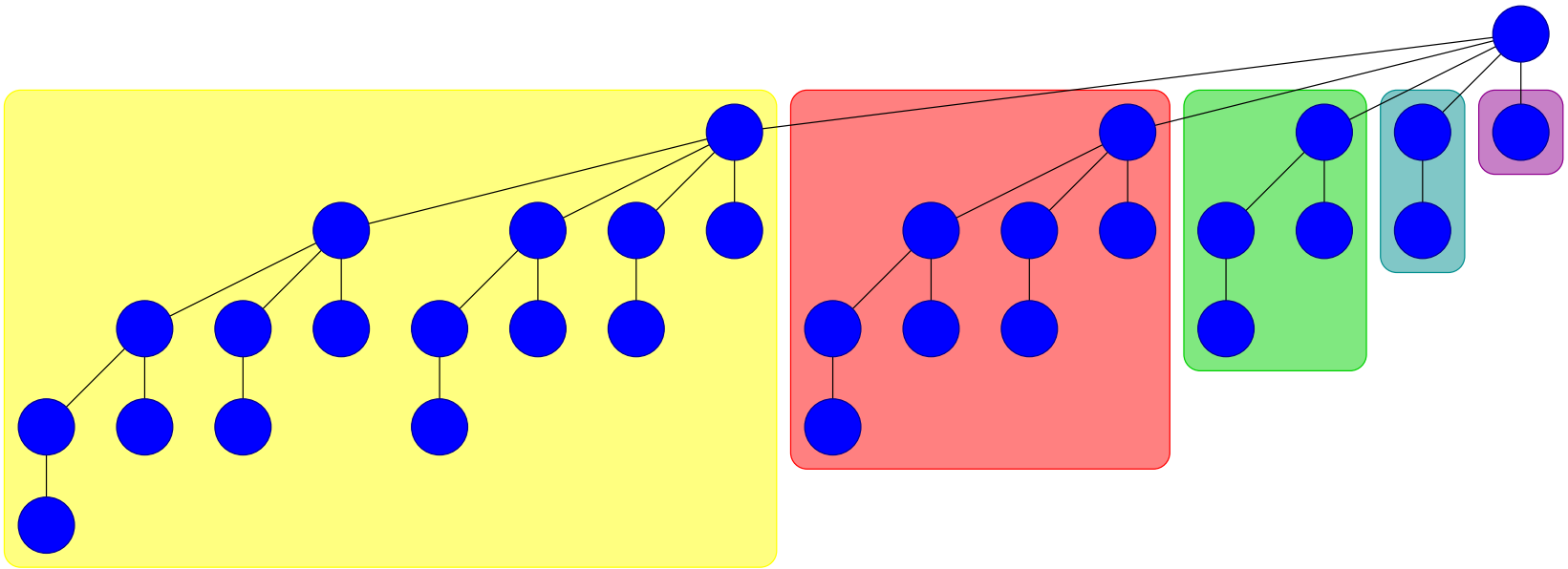


Binomialbäume Beispiel (Forts.)



Binomialbäume Beobachtung

Beobachtung: B_5 enthält B_0, B_1, B_2, B_3 und B_4 als Kindknoten



Allgemein: Der Binomialbaum B_{k+1} enthält die Binomialbäume $B_0, B_1, \ldots, B_k$ als Kinder

Eigenschaften von Binomialbäumen

Lemma. Für alle $k = 0, 1, \ldots$ gilt:

1. B_k besteht aus 2^k Knoten
2. die Tiefe von B_k ist gleich k
3. in B_k gibt auf Tiefe i genau $\binom{k}{i}$ Knoten für alle $i = 0, 1, \ldots, k$
4. Der Grad der Wurzel von B_k ist gleich k und ist größer als der Grad jedes anderen Knotens
5. Werden die Kinder der Wurzel von B_k von links nach rechts von $k-1, k-2, \ldots, 0$ durchnummeriert, dann ist Kind i die Wurzel des Binomialbaums B_i

Eigenschaften von Binomialbäumen (Forts.)

Beweis mittels Induktion über k.

Induktionsanfang: $k = 0$ ✓

Induktionsbehauptung: Obiges Lemma gilt für den Binomialbaum B_{k-1}

Induktionsschritt: $k - 1 \rightsquigarrow k$

1. B_k besteht aus zwei Kopien von B_{k-1}. Unter Einsatz der Induktionsbehauptung ist die Anzahl der Knoten in B_k gleich:

$$2^{k-1} + 2^{k-1} = 2^k.$$

2. Laut IB hat B_{k-1} die Tiefe $k - 1$. Aufgrund der Art und Weise, wie B_k konstruiert wird, hat dieser Baum die Tiefe $(k - 1) + 1 = k$

Eigenschaften von Binomialbäumen (Forts.)

3. Sei $D(k,i)$ die Anzahl der Knoten der Tiefe i des Baums B_k. B_k besteht aus zwei Kopien von B_{k-1}, die um eine Stufe versetzt angeordnet sind. Betrachte einen Knoten von B_{k-1} der Tiefe i. Dieser Knoten kommt in B_k zweimal vor und zwar in Tiefe i und $i+1$.

   Hieraus folgt:

$$\begin{aligned} D(k,i) &= D(k-1,i) + D(k-1,i-1) \\ &= \binom{k-1}{i} + \binom{k-1}{i-1} \\ &= \binom{k}{i} \end{aligned}$$

Eigenschaften von Binomialbäumen (Forts.)

4. Die Wurzel von B_k ist der einzige Knoten, dessen Grad sich von den entsprechenden Knoten in B_{k-1} unterscheidet. Dieser Knoten war vorher die Wurzel von B_{k-1} und erhielt ein zusätzliches Kind. Daher ist sein Grad gleich $(k-1)+1=k$
5. Laut IB sind die Kinder der Wurzel von B_{k-1} die Binomialbäume $B_{k-2}, B_{k-3}, \ldots, B_0$. Bei der Konstruktion von B_k wird eine Kopie von B_{k-1} als linkes Kind unterhalb der Wurzel eingefügt. Somit hat die Wurzel von B_k exakt die im Lemma angegebenen Kinder

Konsequenz: Der maximale Grad eines jeden Knotens in einem Binomialbaum der Größe n ist $\log_2 n$

Binomial Heaps

Binomial Heap $H \rightsquigarrow$ Menge von Binomialbäumen

Eigenschaften:

1. Jeder Binomialbaum in H erfüllt die Min-Heap Eigenschaft, d.h., der Schlüssel eines Knotens ist immer kleiner als die Schlüssel seiner Kinder
2. Für jede Zahl $k = 0, 1, 2, \ldots$ ist in H höchstens ein Binomialbaum B_k enthalten
3. Die Wurzeln der Binomialbäume werden in einer verketteten Liste gespeichert, deren Elemente nach aufsteigendem Grad k sortiert sind

Binomial Heaps (Forts.)

Konsequenz:

- Die Wurzel eines Binomialbaums enthält das Element mit dem kleinsten Schlüssel
- Ein Binomial Heap mit n Knoten besteht aus $\lfloor \log_2 n \rfloor + 1$ Binomialbäumen
- Anhand der Binärdarstellung von n kann man die in H enthaltenen Binomialbäume ermitteln

Binomial Heap Priority Queue

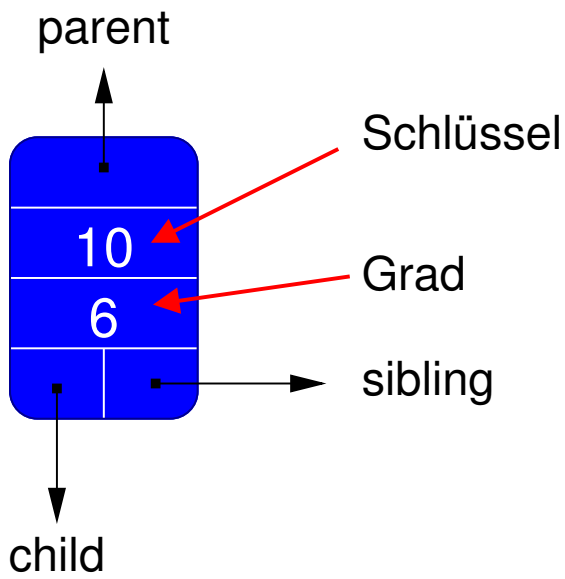
- MakeBinomialHeap() $\rightsquigarrow$ Erstellen eines leeren Binomial Heaps
- BinomialHeapMinimum(H) $\rightsquigarrow$ Zeiger auf einen Knoten mit minimalem Schlüssel zurückgeben
- BinomialLink(y, z) $\rightsquigarrow$ zwei Binomialbäume mit Tiefe k zu einem Binomialbaum der Tiefe $k+1$ verknüpfen
- BinomialHeapUnion(H_1, H_2) $\rightsquigarrow$ Vereinigung der Heaps H_1 und H_2 zu einem Heap

Binomial Heap Priority Queue (Forts.)

- BinomialHeapInsert(H, x) $\rightsquigarrow$ Einfügen des Elements x in den Heap H
- BinomialHeapExtractMin(H) $\rightsquigarrow$ aus dem Heap H das Element mit dem kleinsten Schlüssel entfernen
- BinomialHeapDelete(H, x) $\rightsquigarrow$ Element x aus dem Heap H entfernen
- BinomialHeapDecreaseKey(x, k) $\rightsquigarrow$ Verkleinern des Schlüssels des Elements x auf den Wert k

Knoten innerhalb eines Binomialheaps

Modellierung der Knoten als Zeigerstruktur:

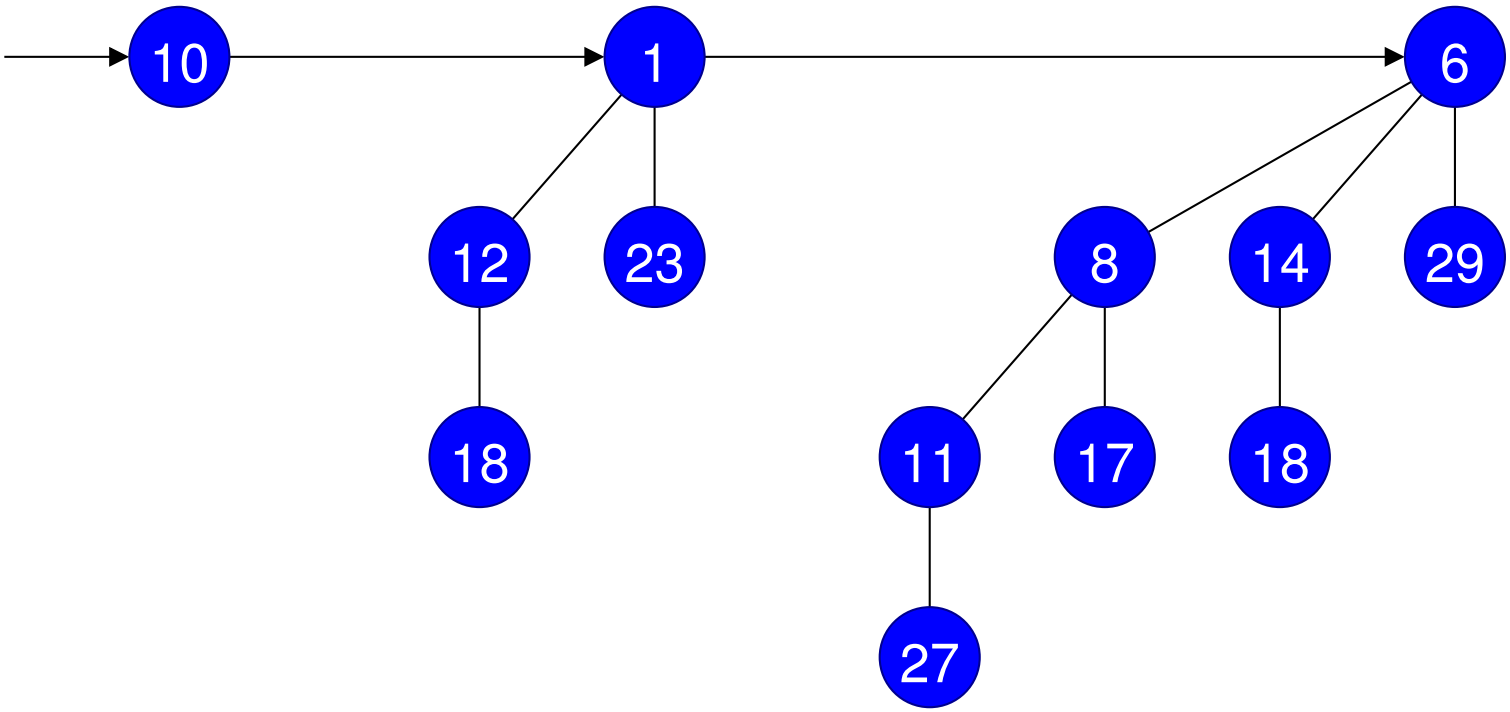


Vereinbarung:

- Globale Variable $head(H) \rightsquigarrow$ Zeiger auf die Wurzel des ersten Binomialbaums im Heap
- Wurzelknoten bilden eine verkettete Liste, die nach aufsteigendem Grad sortiert ist

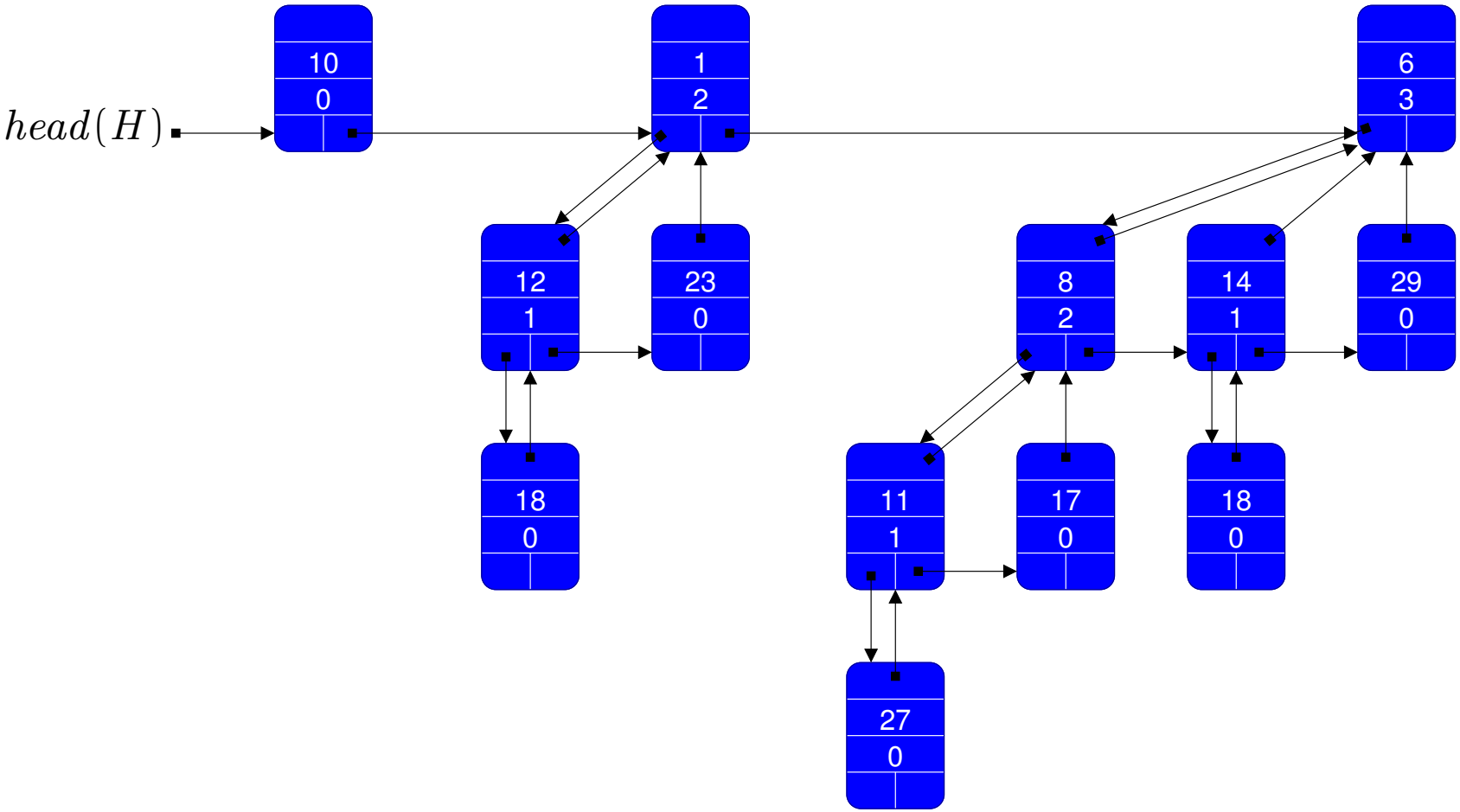
Binomial Heap Beispiel

Ein Binomial Heap . . .



Binomial Heap Beispiel (Forts.)

. . . und dessen Realisierung im Hauptspeicher



Suche eines Elements mit minimalen Schlüssel

Beobachtung: Da jeder Binomialbaum die Min-Heap Eigenschaft erfüllt, befindet sich alle Elemente mit minimalem Schlüssel in den Wurzelknoten der Binomialbäume

Idee: Durchlaufe alle Wurzelknoten und suche dort nach einem minimalen Element

Laufzeit:

- Bekannt: Ein Binomial Heap mit n Elementen besteht aus höchstens $\lfloor \log_2 n \rfloor$ Binomialbäumen
- Die Liste der Wurzelknoten enthält $\lfloor \log_2 n \rfloor$ Knoten

$\rightsquigarrow$ Laufzeit $O(\log_2 n)$

Algorithmus BinomialHeapMinimum(H)

BinomialHeapMinimum(H)

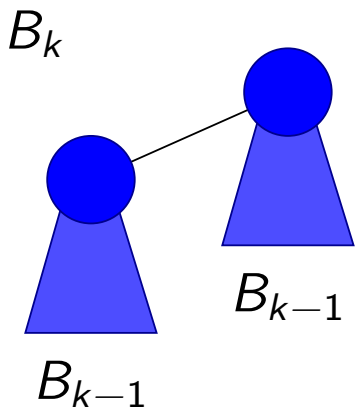
Input: Binomial Heap H

Output: Element mit minimalem Schlüssel

```
1 y := NIL
2 x := head(H)
3 min := ∞
4 while x ≠ NIL
5     if key(x) < min then
6         min := key(x)
7         y := x
8     x := sibling(x)
9 return y
```

Verknüpfung von Binomialbäumen

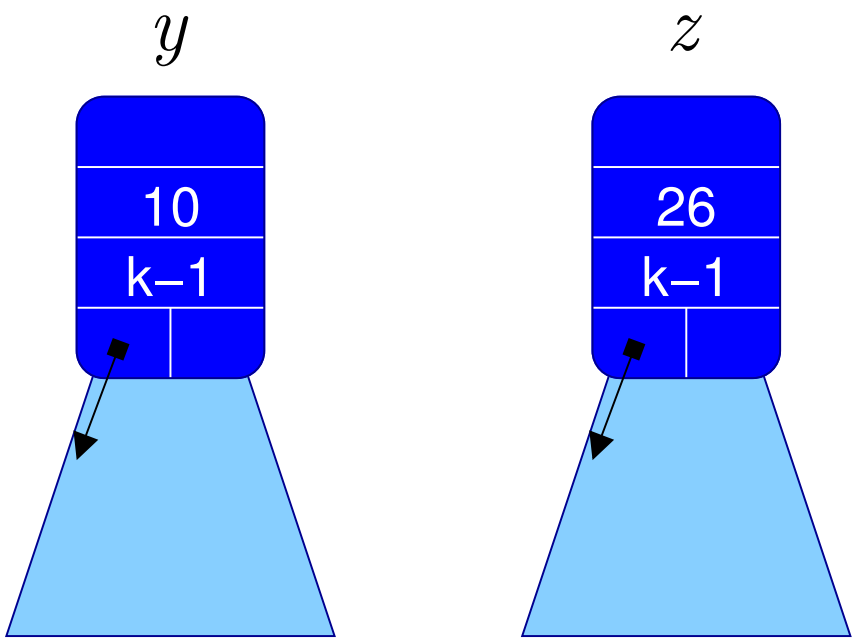
Aufgabe: Vereinige zwei Binomialbäume B_{k-1} zu einem Binomialbaum B_k



Ansatz: Zeiger "umbiegen"

Verknüpfung von Binomialbäumen (Forts.)

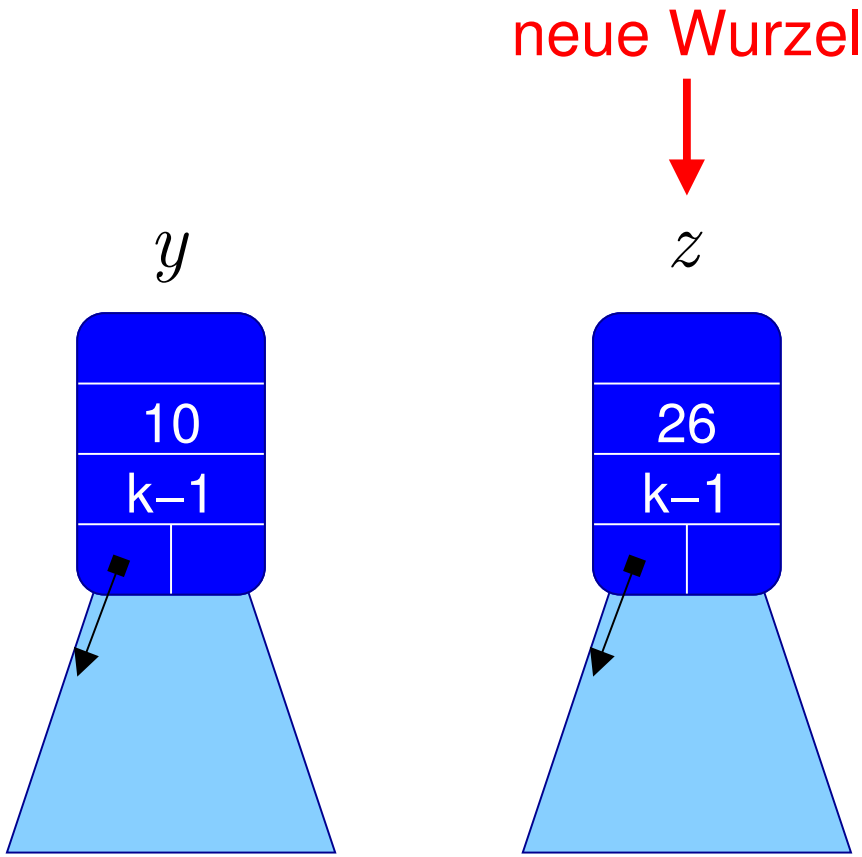
Gegeben: Zeiger y und z auf Wurzelknoten von B_{k-1}



Verknüpfung von Binomialbäumen (Forts.)

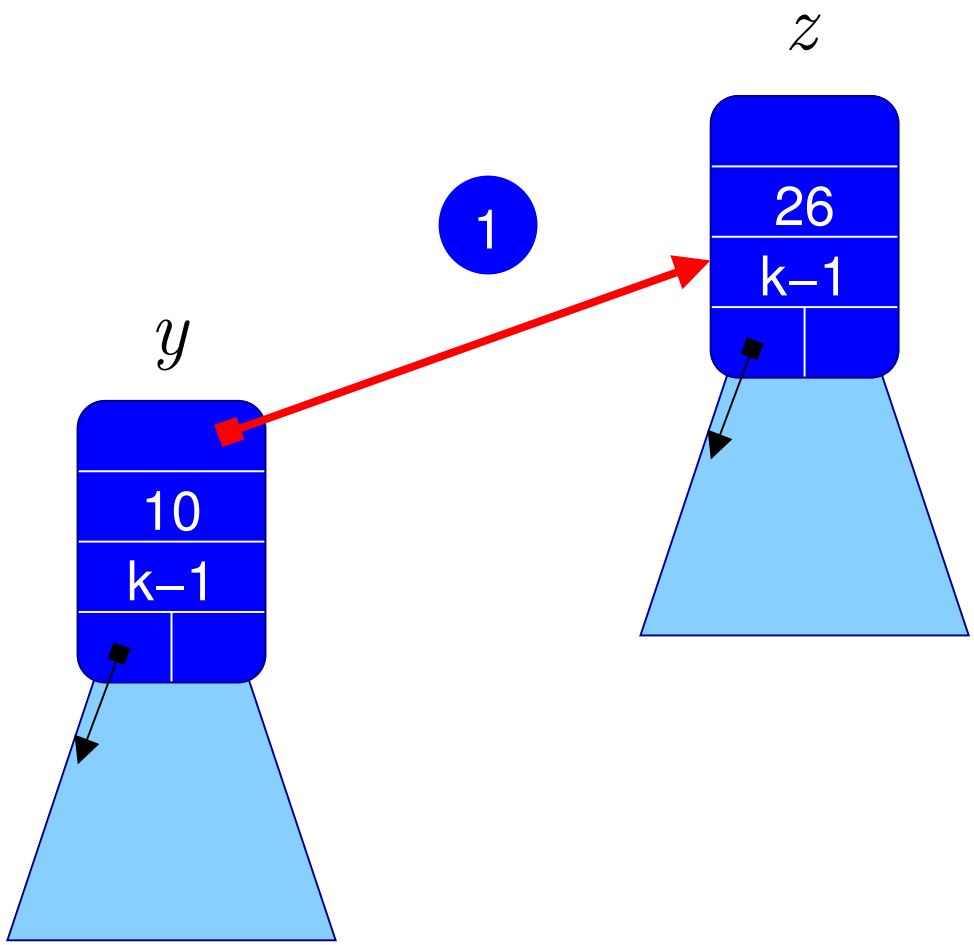
Ziel:

- z wird die Wurzel eines Binomialbaums B_k
- y ist das erste Kind von z



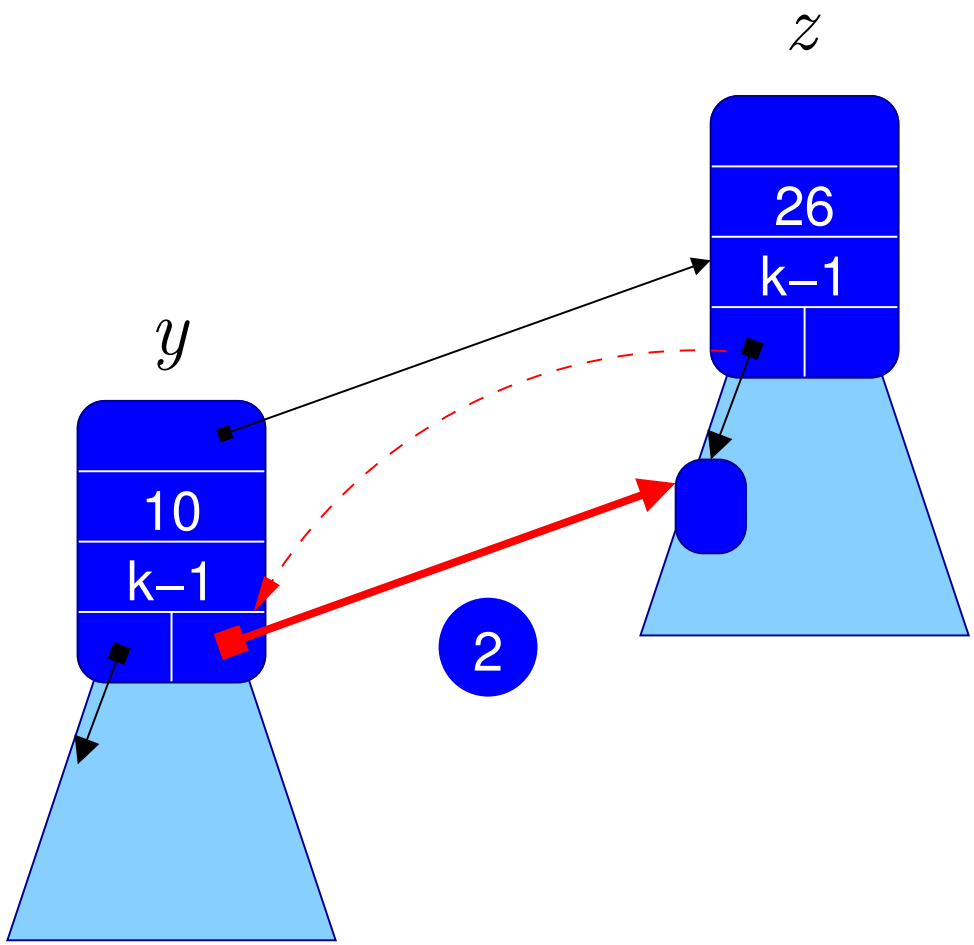
Verknüpfung von Binomialbäumen (Forts.)

Schritt 1: z wird der Elternknoten von y



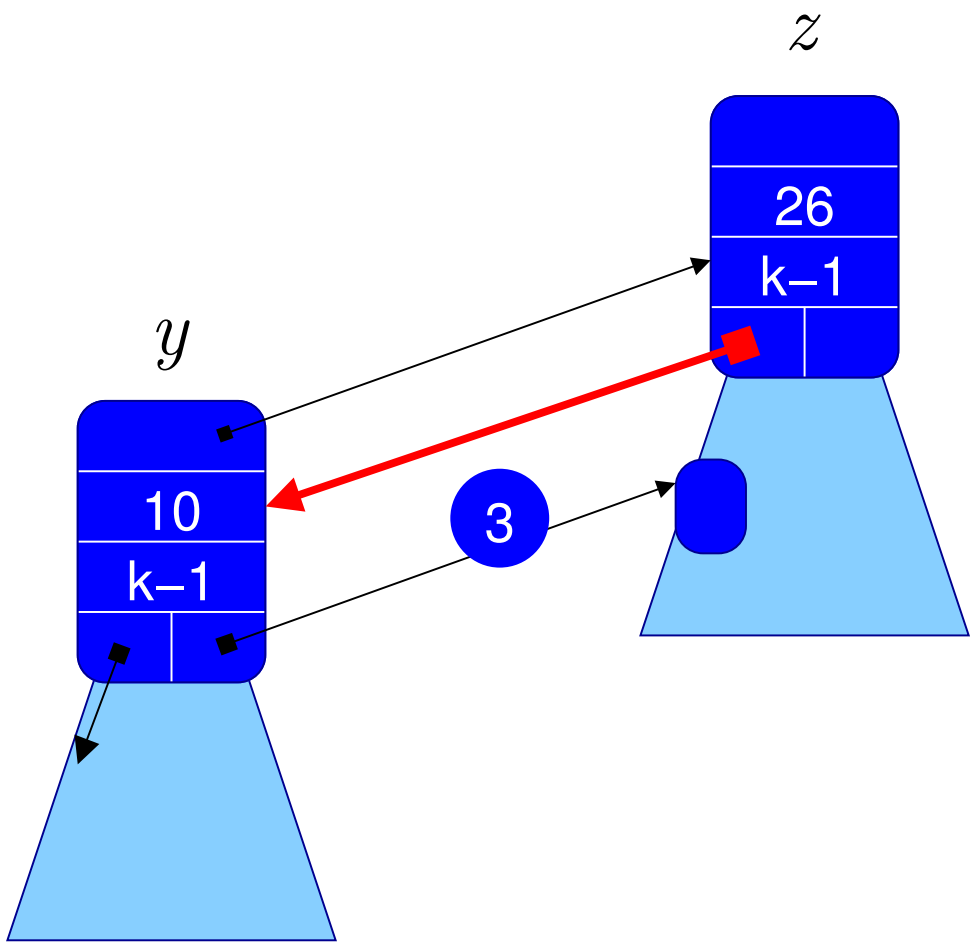
Verknüpfung von Binomialbäumen (Forts.)

Schritt 2: Das erste Kind von z wird der Geschwisterknoten von y



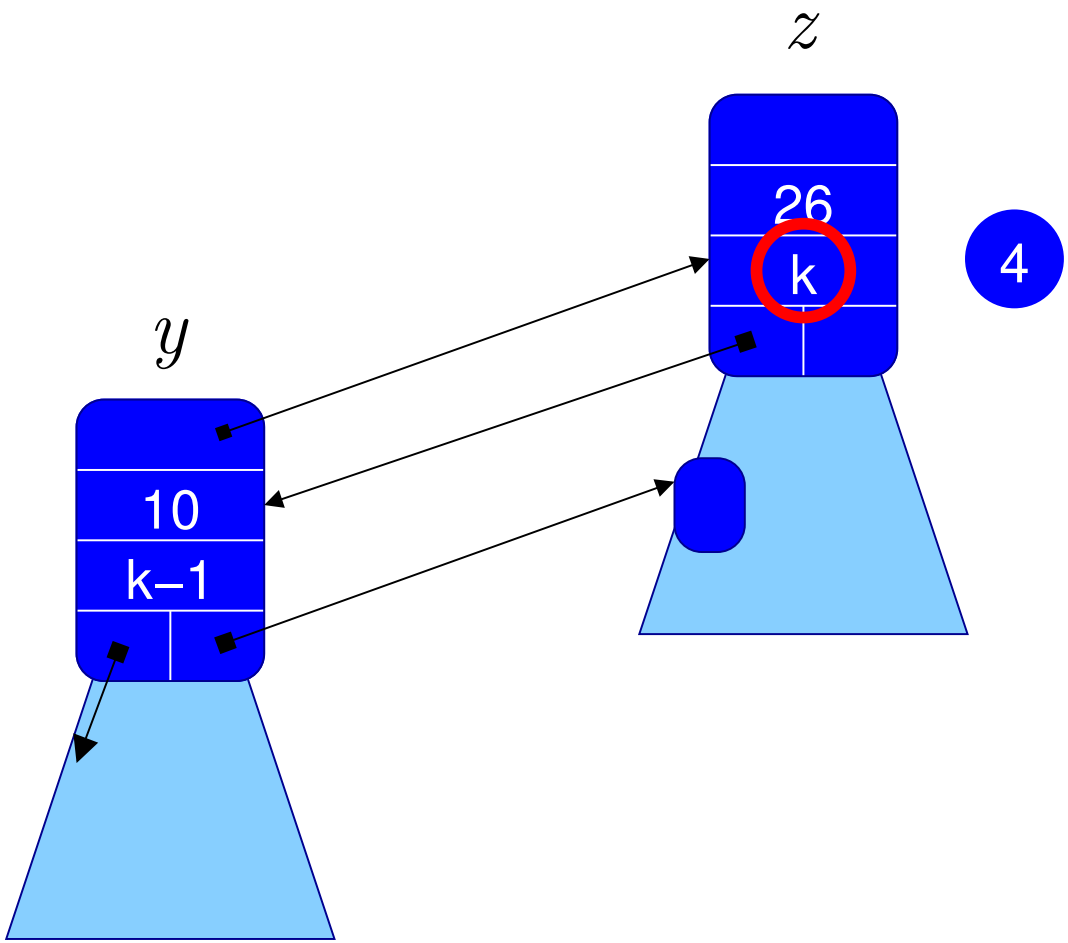
Verknüpfung von Binomialbäumen (Forts.)

Schritt 3: y wird das erste Kind von z



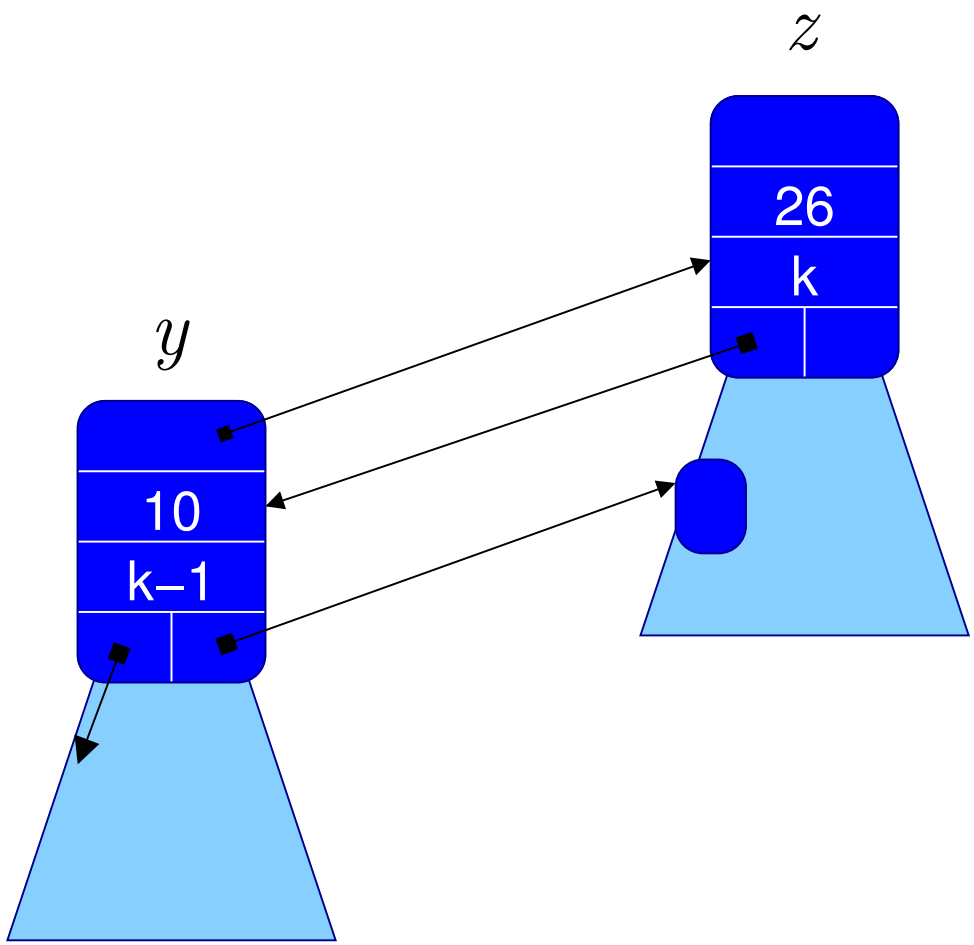
Verknüpfung von Binomialbäumen (Forts.)

Schritt 4: Die Tiefe von z wird um 1 erhöht



Verknüpfung von Binomialbäumen (Forts.)

Ergebnis:



Algorithmus BinomialLink(y, z)

BinomialLink(y, z)

Input: Wurzeln y und z zweier Binomialbäume B_{k-1}

Output: Binomialbaum B_k mit Wurzel z

1 $parent(y) := z$

2 $sibling(y) := child(z)$

3 $child(z) := y$

4 $degree(z) := degree(z) + 1$

Laufzeit: $O(1)$

Merge Operation für Heaps

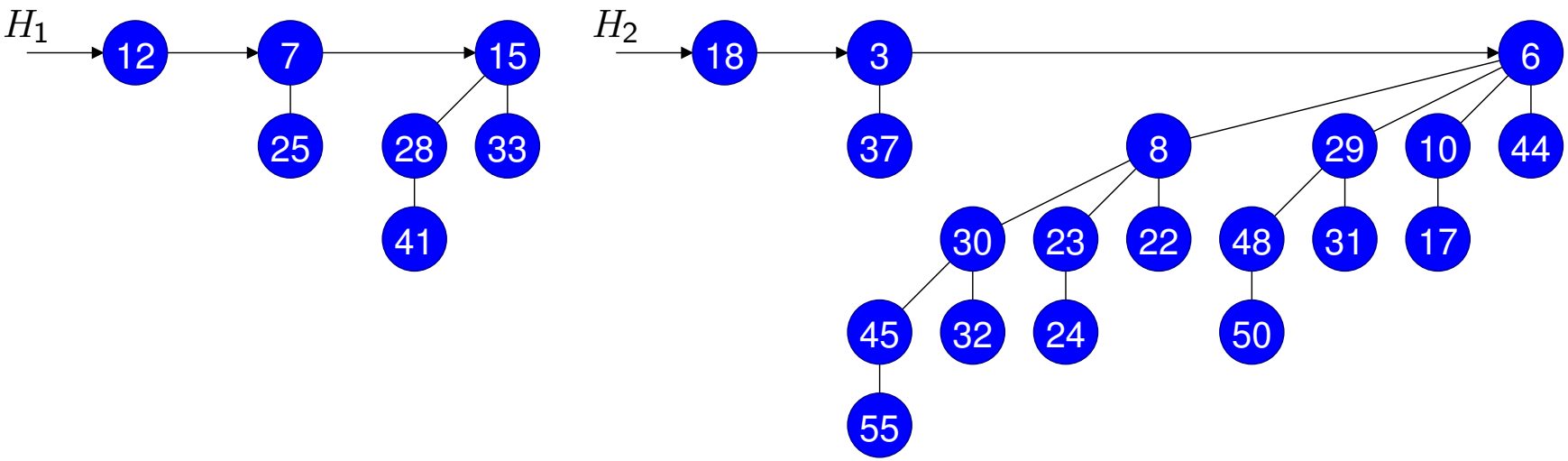
Aufgabe: Füge die Wurzellisten der Binomial Heaps H_1 und H_2 zu einer Liste zusammen

Nebenbedingung: Die Liste soll anhand der Grade der Wurzeln aufsteigend sortiert sein

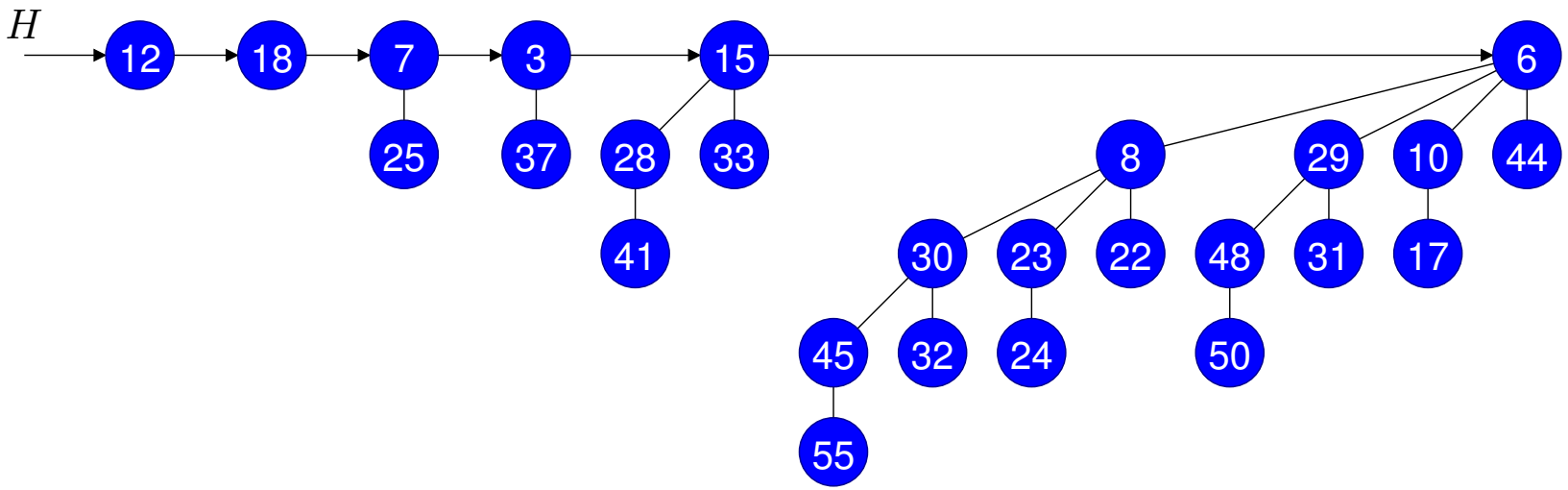
Realisierung: Algorithmus $\textsc{BinomialHeapMerge}(H_1, H_2)$
$\rightsquigarrow$ Übungsaufgabe

Laufzeit: $O(\log_2 n_1 + \log_2 n_2)$, wobei n_1 und n_2 die Anzahl der Elemente im Heap H_1 bzw. H_2 ist

Beispiel Merge Operation



Beispiel Merge Operation (Forts.)



Vereinigung von Binomial Heaps

Aufgabe: Vereinige zwei Binomial Heaps H_1 und H_2

Vorverarbeitungsschritt: Merge Operation

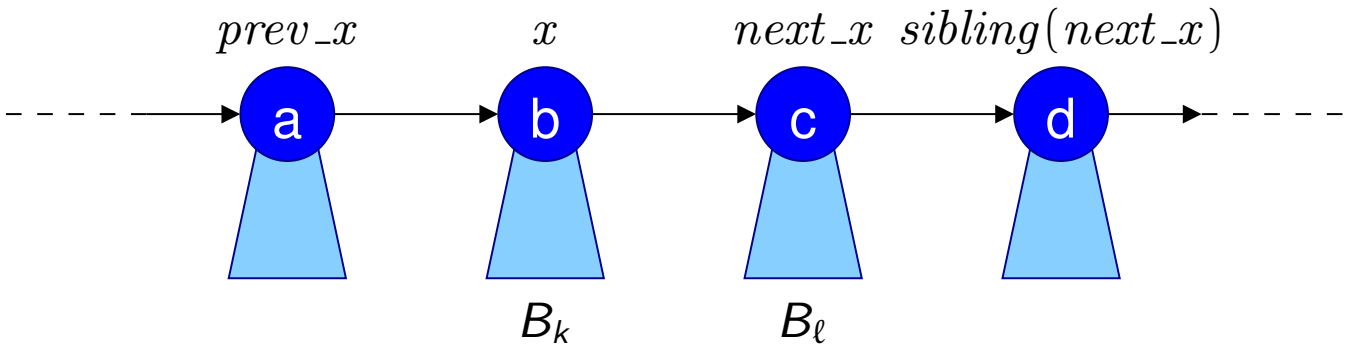
Ansatz: Durchlaufe die von $\textsc{BinomialHeapMerge}(H_1, H_2)$ gelieferte Liste und verknüpfe Binomialbäume gleicher Größe

Variablen:

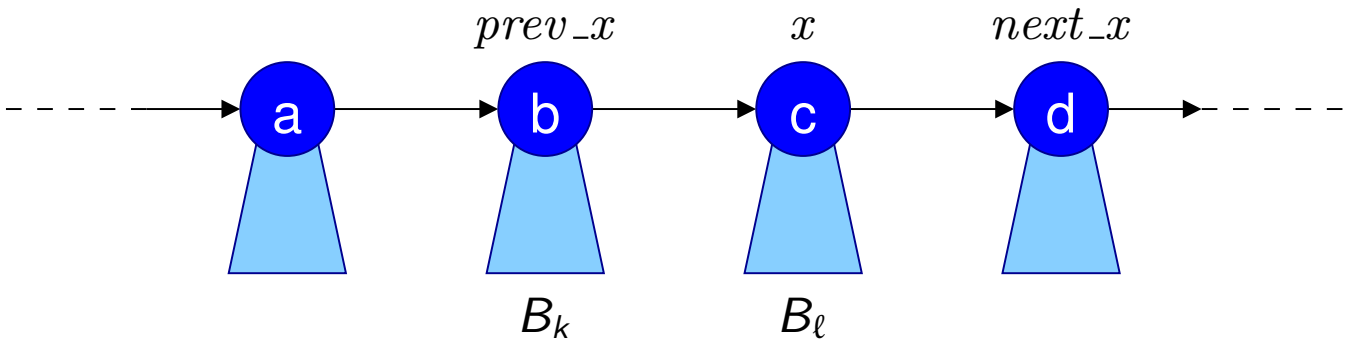
- $x \rightsquigarrow$ Zeiger auf die Wurzel des aktuellen Binomialbaums
- $prev\_x \rightsquigarrow$ Zeiger auf den Vorgänger von x
- $next\_x \rightsquigarrow$ Zeiger auf den Nachfolger von x

Vereinigung von Binomial Heaps (Fall 1)

Fall 1: $degree(x) \neq degree(next\_x)$

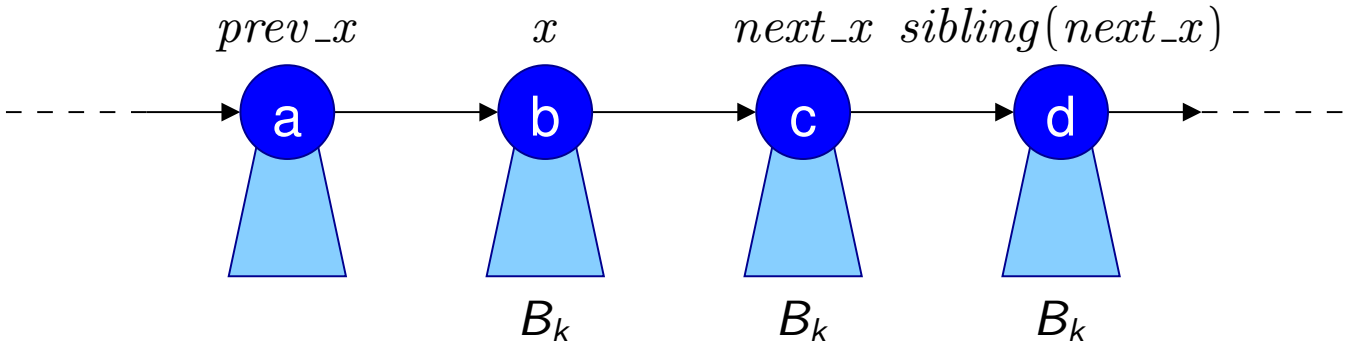


Lösung: alle Zeiger nach rechts verschieben

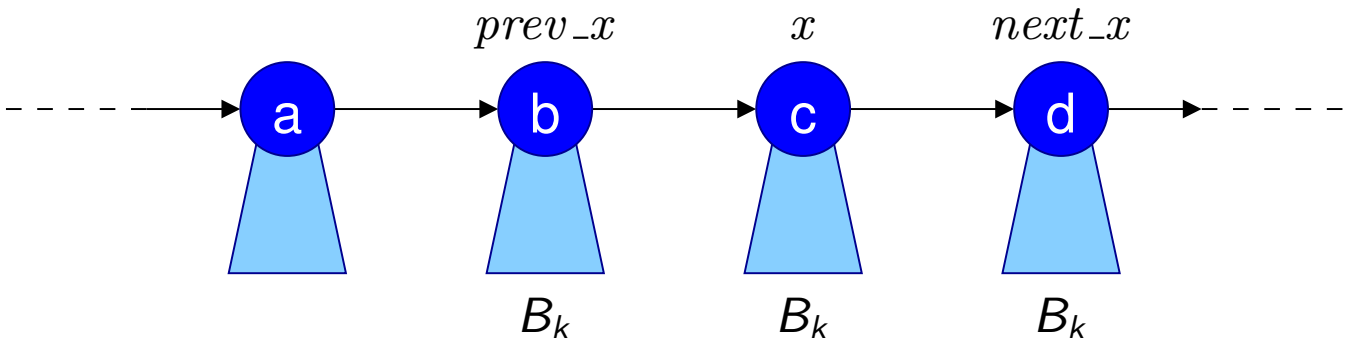


Vereinigung von Binomial Heaps (Fall 2)

Fall 2: $degree(x) = degree(next\_x) = degree(sibling(next\_x))$

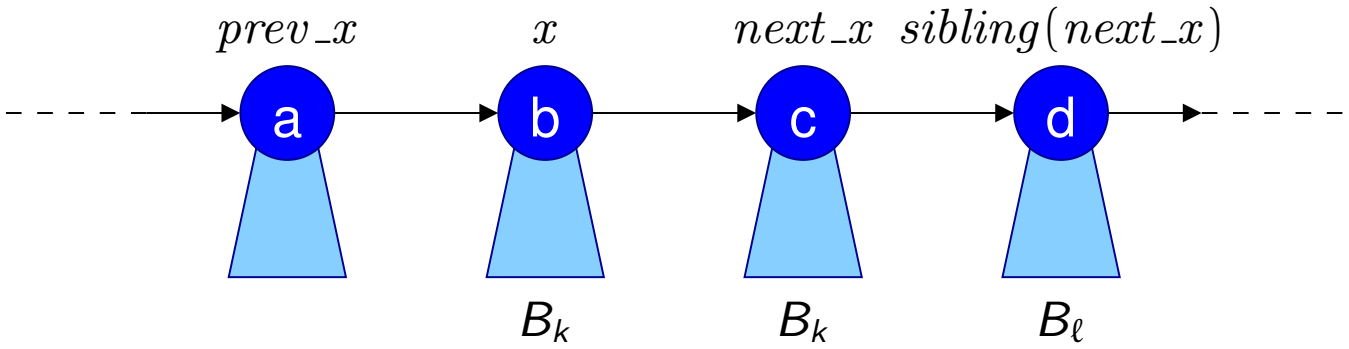


Lösung: alle Zeiger nach rechts verschieben

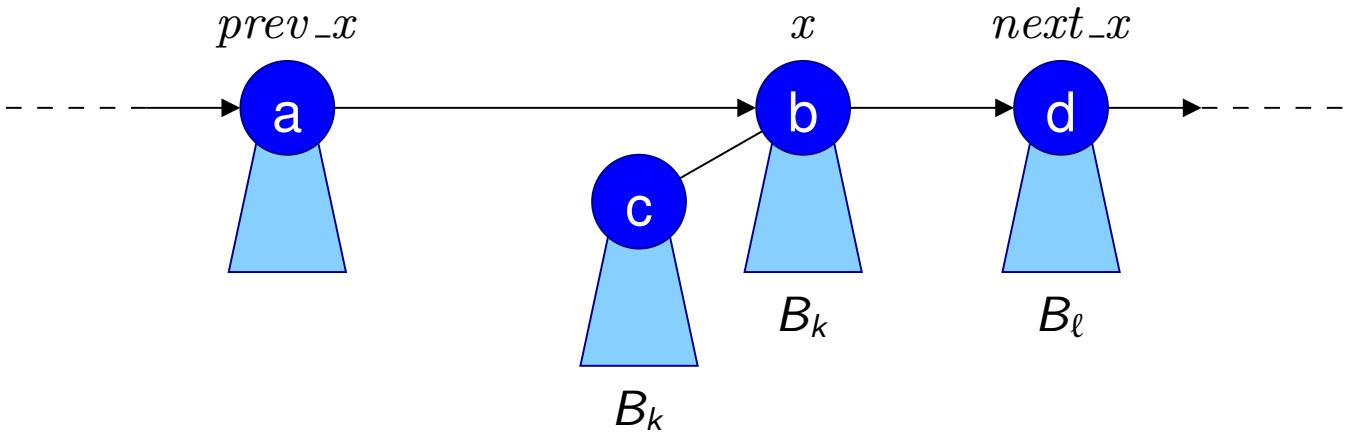


Vereinigung von Binomial Heaps (Fall 3)

Fall 3: $degree(x) = degree(next\_x) \neq degree(sibling(next\_x))$ und $key(x) \leq key(next\_x)$

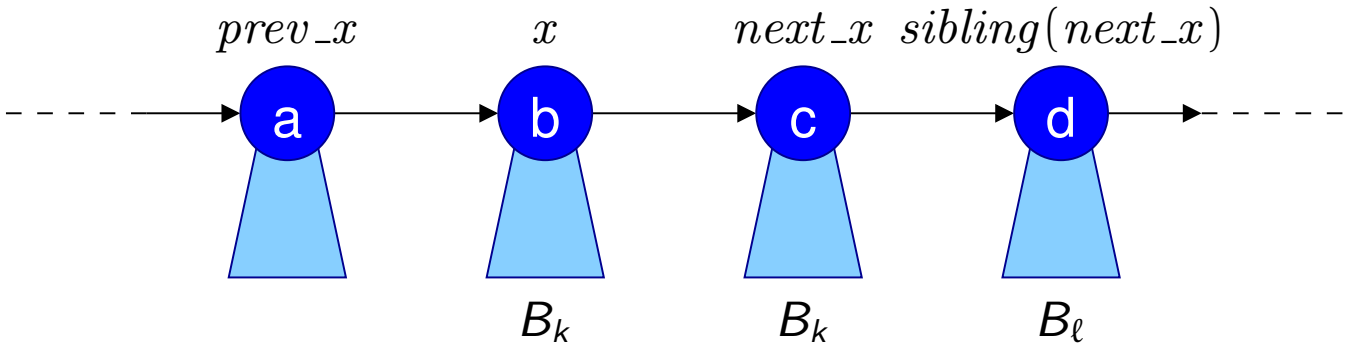


Lösung: Füge $next\_x$ unterhalb von x ein

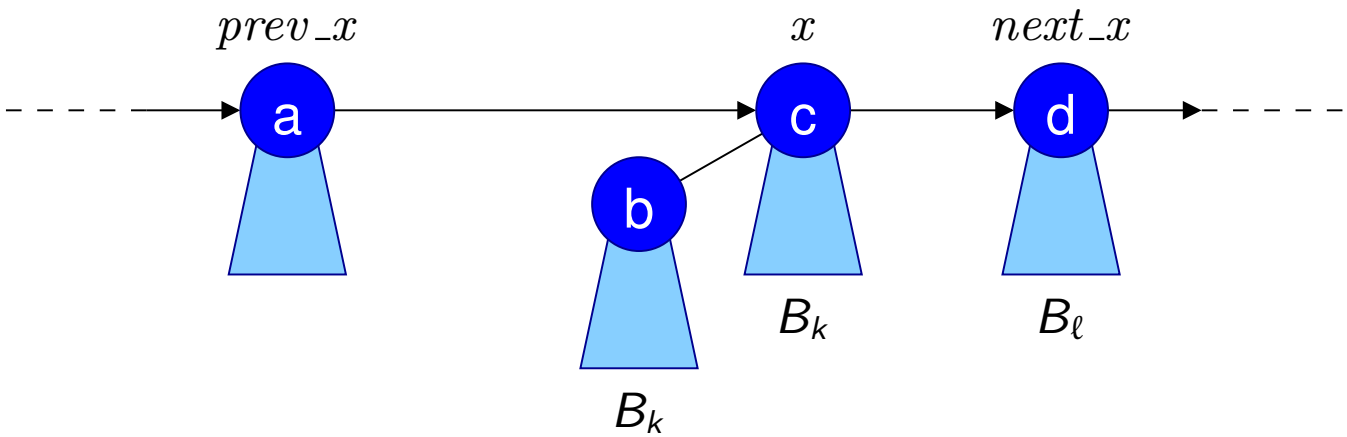


Vereinigung von Binomial Heaps (Fall 4)

Fall 4: $degree(x) = degree(next\_x) \neq degree(sibling(next\_x))$ und $key(x) > key(next\_x)$



Lösung: Füge x unterhalb von $next\_x$ ein



Algorithmus BinomialHeapUnion(H_1, H_2)

BinomialHeapUnion(H_1, H_2)
Input: Binomial Heaps H_1 und H_2
Output: Vereinigung H der Heaps H_1 und H_2

```
1  H := MakeBinomialHeap()
2  head(H) := BinomialHeapMerge(H_1, H_2)
3  Gib den Speicher der Knoten H_1 und H_2 frei
4  if head(H) = NIL then
5     return H
6  prev_x := NIL
7  x := head(H)
8  next_x := sibling(x)
```

BinomialHeapUnion(H_1, H_2) (Forts.)

```
 9 while next_x ≠ NIL do
10   if degree(x) ≠ degree(next_x) or
11       (sibling(next_x) ≠ NIL and
12       degree(sibling(next_x)) = degree(x)) then
13     prev_x := x
14     x := next_x
15   else
16     if key(x) ≤ key(next_x) then
17       sibling(x) := sibling(next_x)
18       BinomialLink(next_x, x)
```

BinomialHeapUnion(H_1, H_2) (Forts.)

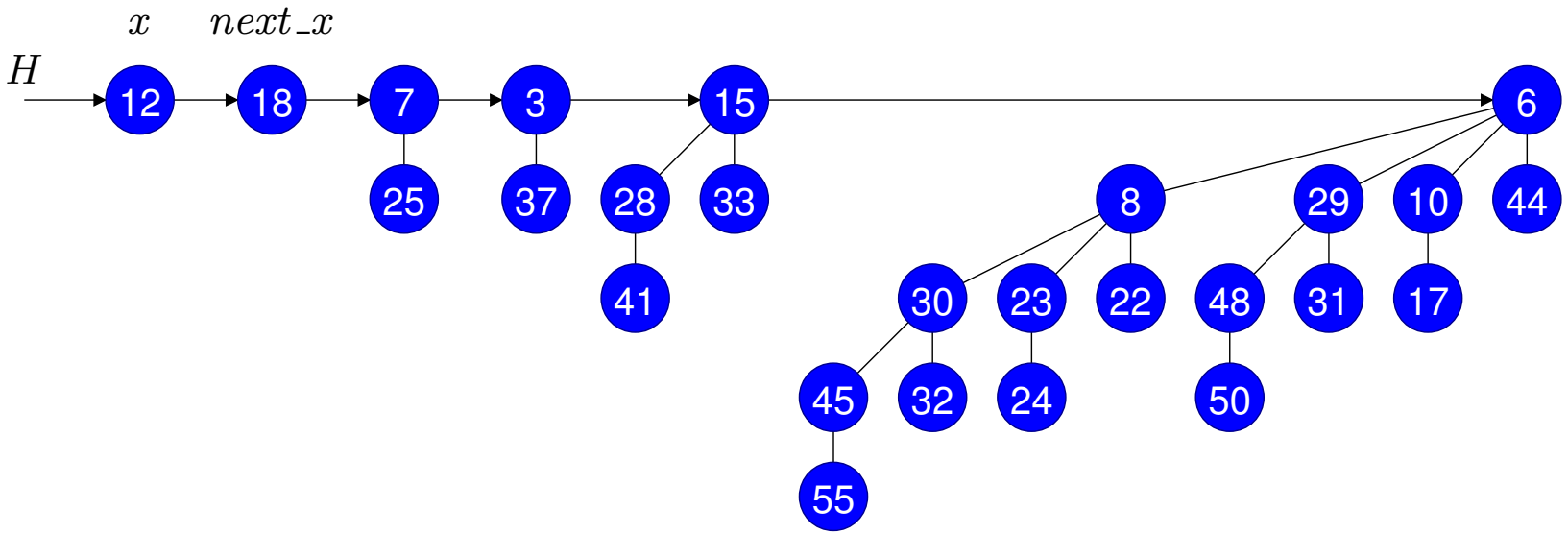
```
19        else
20            if prev_x = NIL then
21                head(H) := next_x
22            else
23                sibling(prev_x) := next_x
24            BinomialLink(x, next_x)
25            x := next_x
26        next_x := sibling(x)
27    return H
```

Bemerkungen

- Behandlung von
  - Fall 1 und 2 $\rightsquigarrow$ Zeilen 10 bis 14
  - Fall 3 $\rightsquigarrow$ Zeilen 17 bis 18
  - Fall 4 $\rightsquigarrow$ Zeilen 20 bis 25
- In jedem Durchlauf der Schleife wandern die Zeiger entweder um eine Position nach rechts oder die Wurzelliste wird um einen Knoten verkürzt
- Laufzeit: $O(\log_2 n)$

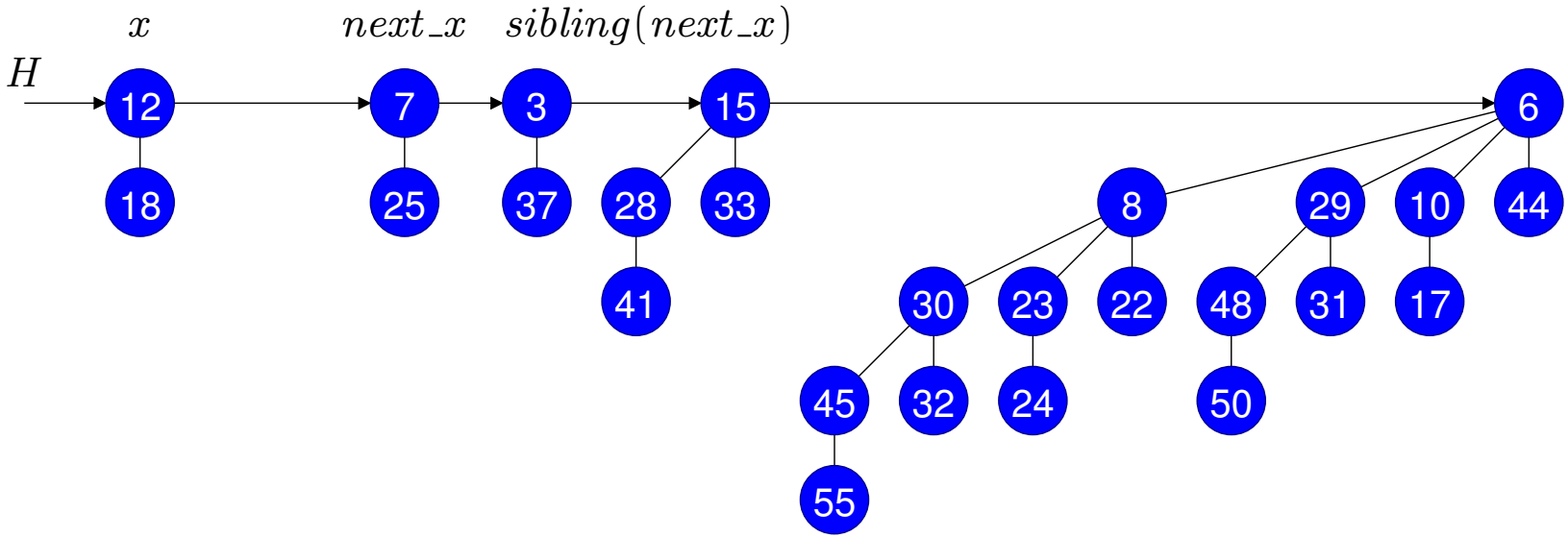
Beispiel zur Heap Vereinigung

Ausgangspunkt: Obiges Beispiel nach der Merge-Operation



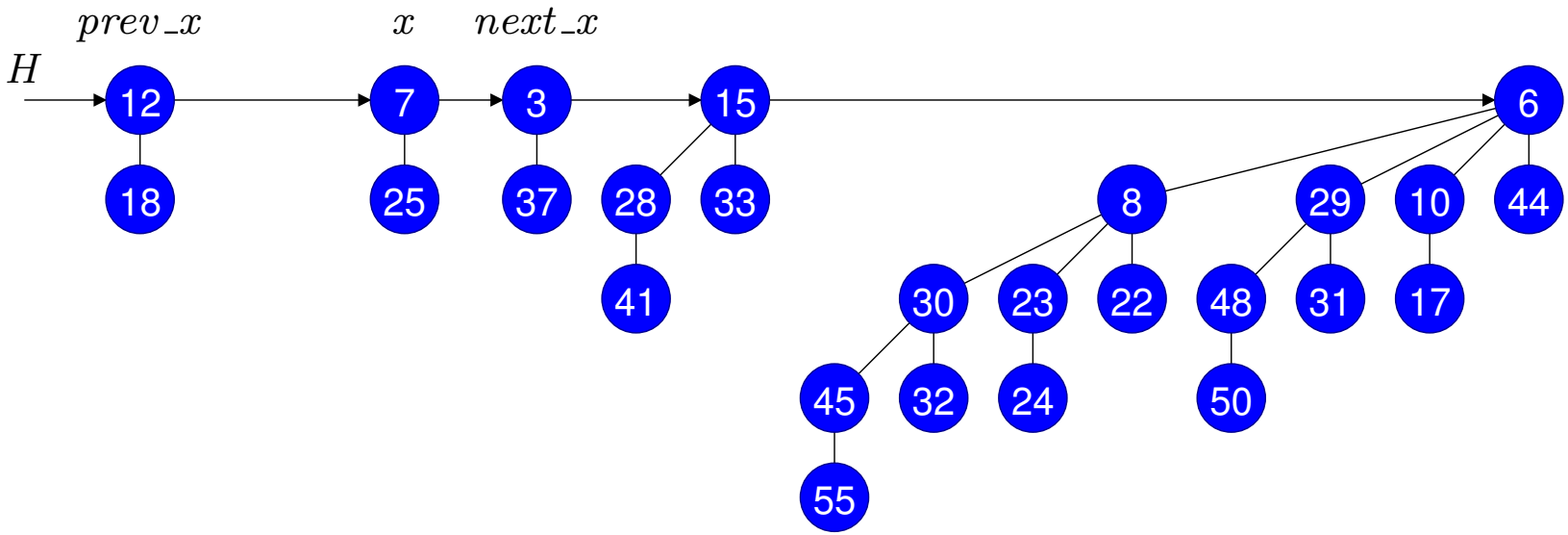
Situation: $degree(x) = degree(next\_x)$ und $key(x) \leq key(next\_x) \rightsquigarrow$ Fall 3

Beispiel zur Heap Vereinigung (Forts.)



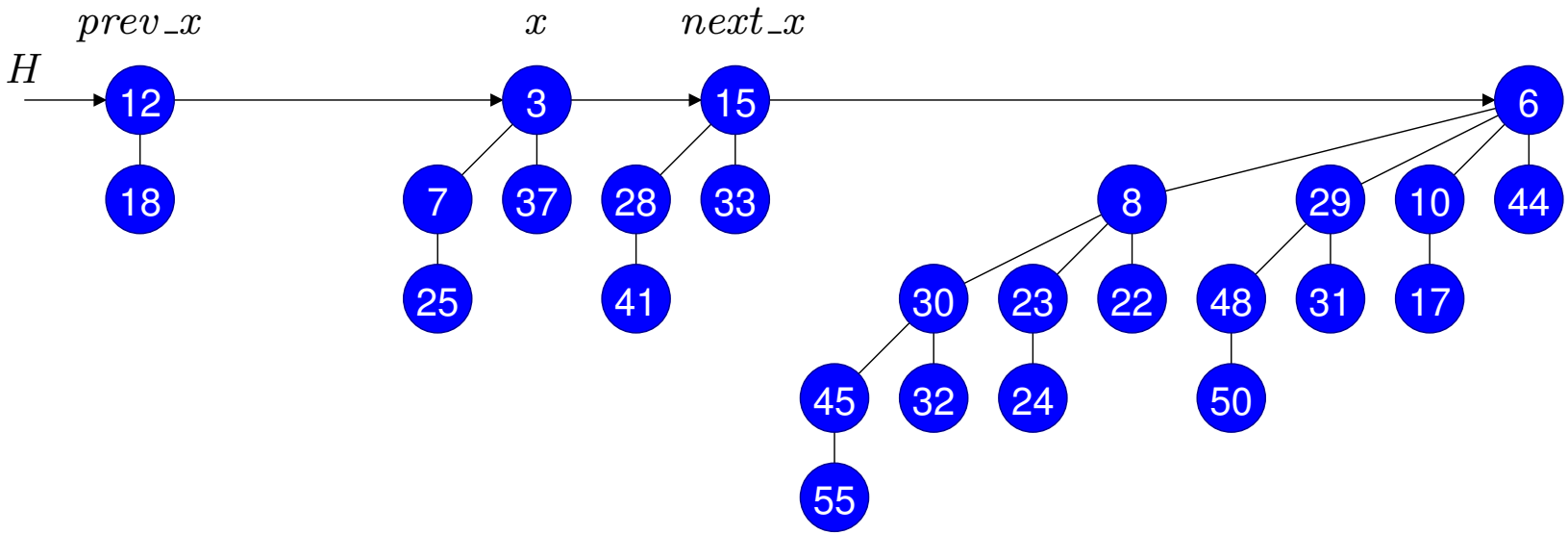
Situation: $degree(x) = degree(next\_x) = degree(sibling(next\_x)) \rightsquigarrow$ Fall 2

Beispiel zur Heap Vereinigung (Forts.)



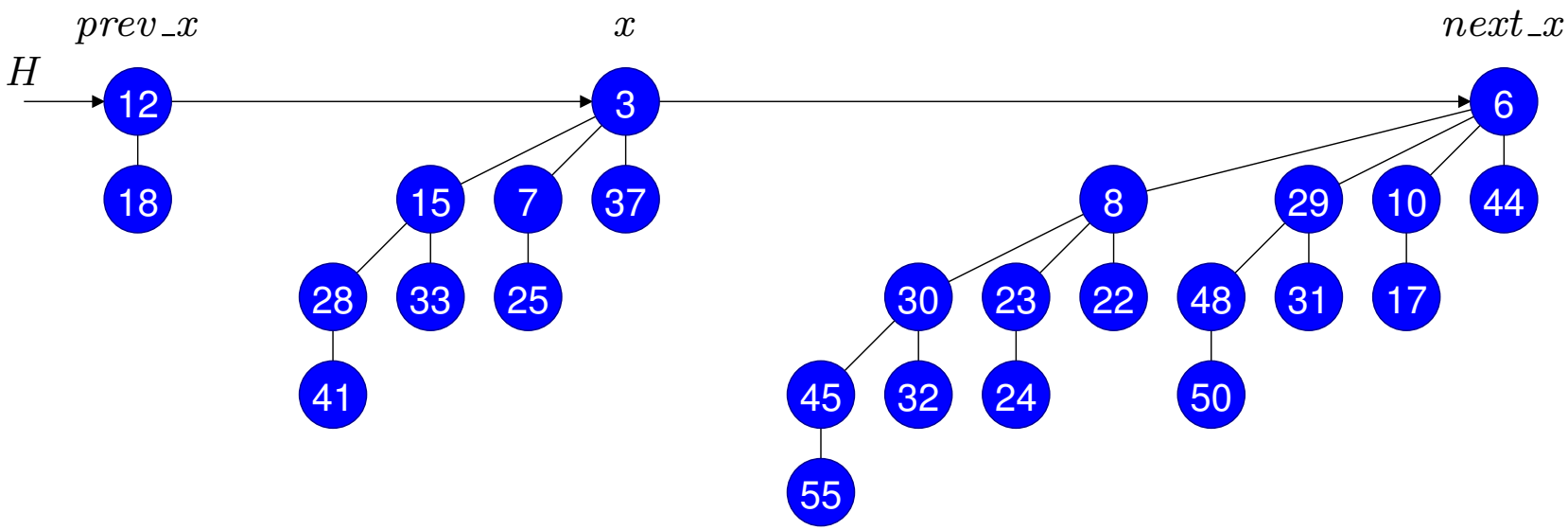
Situation: $degree(x) = degree(next\_x)$ und $key(x) > key(next\_x)$ $\rightsquigarrow$ Fall 4

Beispiel zur Heap Vereinigung (Forts.)



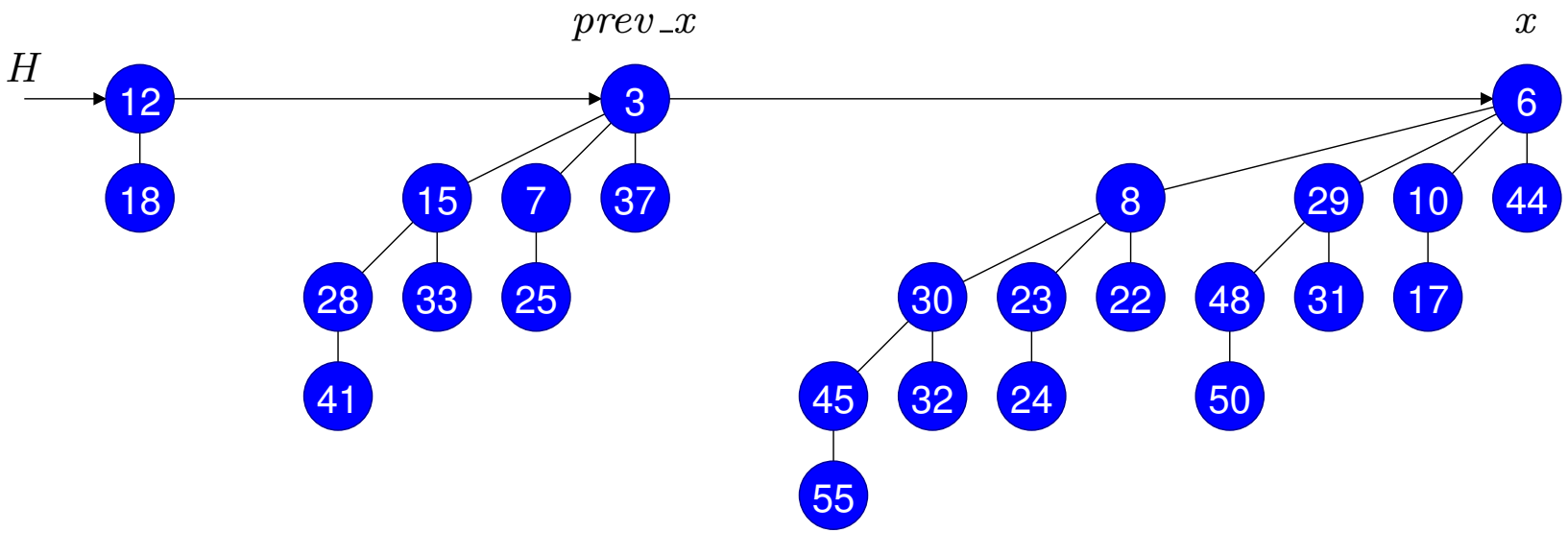
Situation: $degree(x) = degree(next\_x)$ und $key(x) \leq key(next\_x) \rightsquigarrow$ Fall 3

Beispiel zur Heap Vereinigung (Forts.)



Situation: $degree(x) \neq degree(next\_x) \rightsquigarrow$ Fall 1

Beispiel zur Heap Vereinigung (Forts.)



Situation: $next\_x = \texttt{NIL} \rightsquigarrow$ while Schleife wird verlassen

Einfügen eines Knotens

Idee: Führe das Einfügen auf die Vereinigung zurück

BinomialHeapInsert(H, x)
Input: Binomial Heap H, Knoten x

1 $H' :=$ MakeBinomialHeap()
2 *Allokiere Knoten* x
3 $parent(x) :=$ NIL
4 $child(x) :=$ NIL
5 $sibling(x) :=$ NIL
6 $degree(x) := 0$
7 $head(H') := x$
8 $H =$ BinomialHeapUnion(H, H')

Laufzeit: $O(\log_2 n)$ bei einem Heap H mit n Elementen

Löschen eines Knotens mit minimalem Schlüssel

BinomialHeapExtractMin(H)

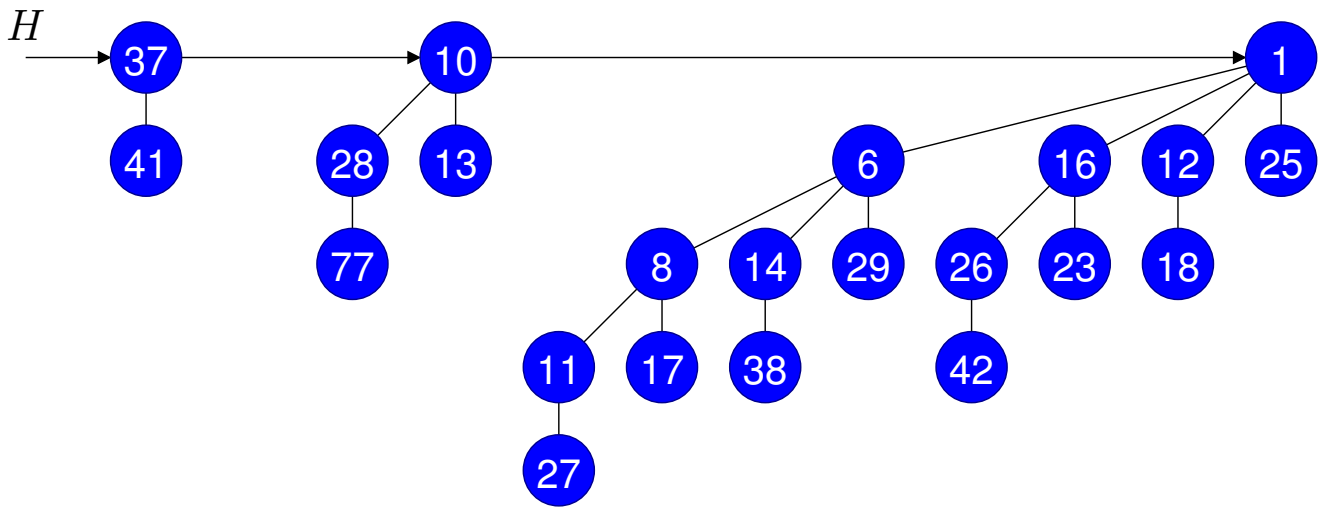
Input: Binomial Heap H

Output: Element x mit minimalem Schlüssel

1. *Finde in der Wurzelliste von H einen Knoten x mit minimalem Schlüssel*
2. *Lösche x aus der Wurzelliste von H*
3. $H' :=$ MakeBinomialHeap()
4. *Invertiere die Reihenfolge der Kinder von x und setze $head(H')$ auf den Anfang dieser Liste*
5. $H =$ BinomialHeapUnion(H, H')
6. **return** x

Beispiel Entfernen eines Minimums

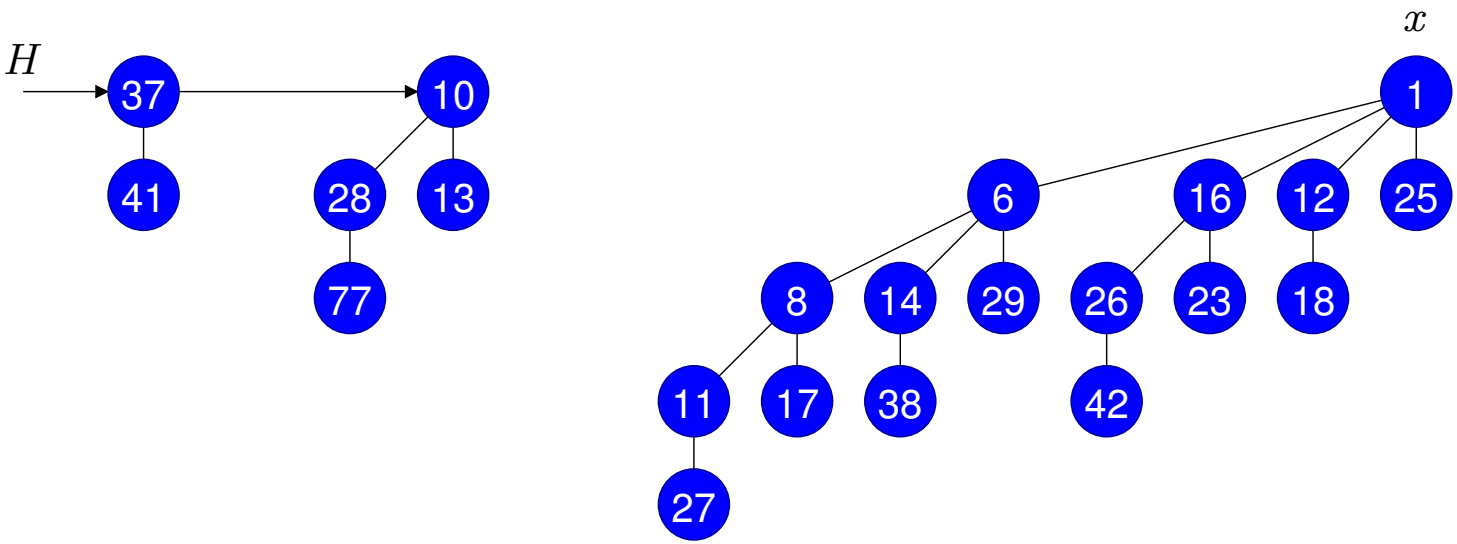
Ausgangssituation:



Nächster Schritt: Suche des Minimums

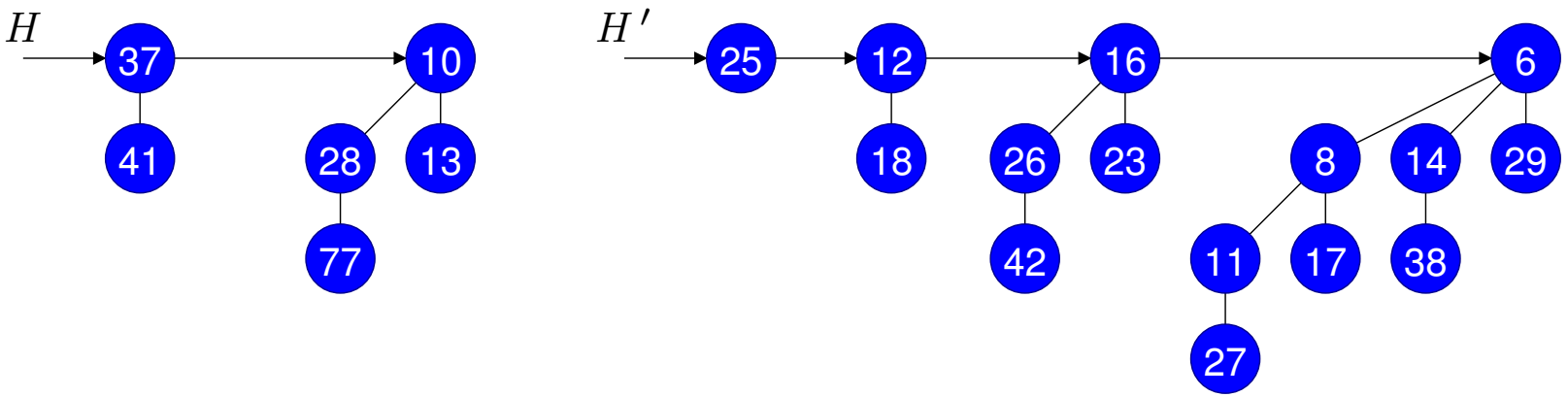
Beispiel Entfernen eines Minimums (Forts.)

Minimum: Knoten x



Nächster Schritt: Entfernen von x aus dem Heap und Invertieren der Liste seiner Kinder

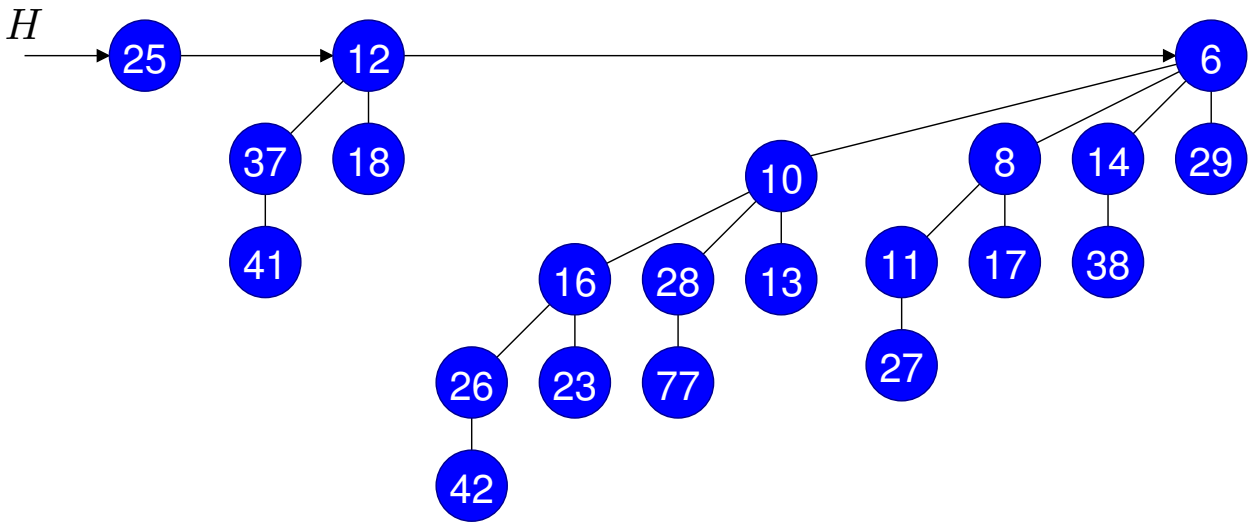
Beispiel Entfernen eines Minimums (Forts.)



Nächster Schritt: Vereinigen der beiden Heaps H und H'

Beispiel Entfernen eines Minimums (Forts.)

Ergebnis:



Verkleinern des Schlüssels eines Knotens

Idee: Schiebe den Inhalt des Knotens solange in Richtung der Wurzel des Binomialbaums bis zu einem Elternknoten mit kleinerem Schlüssel

Beachte: Es werden bei dieser Operation keine Zeiger verändert, sondern die Inhalte des Schlüsselfelds sowie der Satellitendaten vertauscht

Laufzeit:

- Schlimmster Fall: Der Schlüssels eines Blatts im größten Binomialbaum des Heaps wird so klein, dass der Datensatz bis zur Wurzel hoch wandert
- Laufzeit: $O(\log_2 n)$

Algorithmus zum Verkleinern des Schlüssels

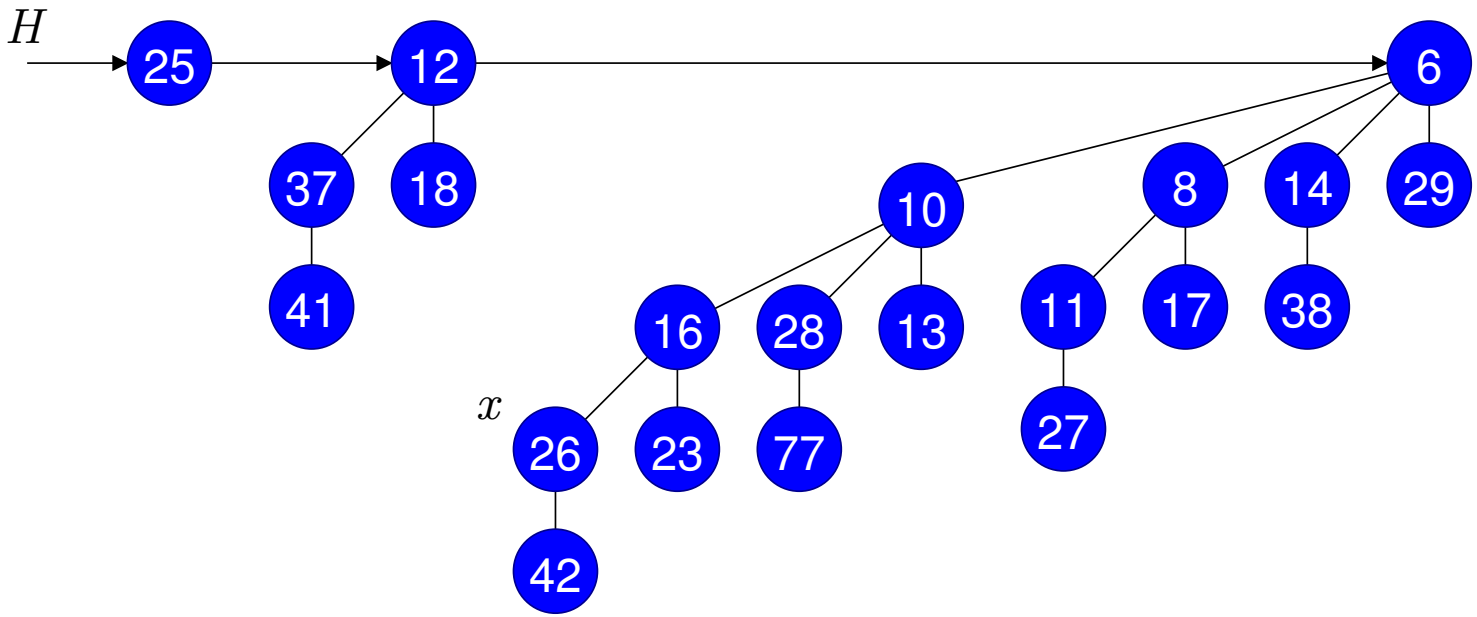
BinomialHeapDecreaseKey(H, x, k)

Input: Binomial Heap H, Knoten x, neuer Schlüssel k

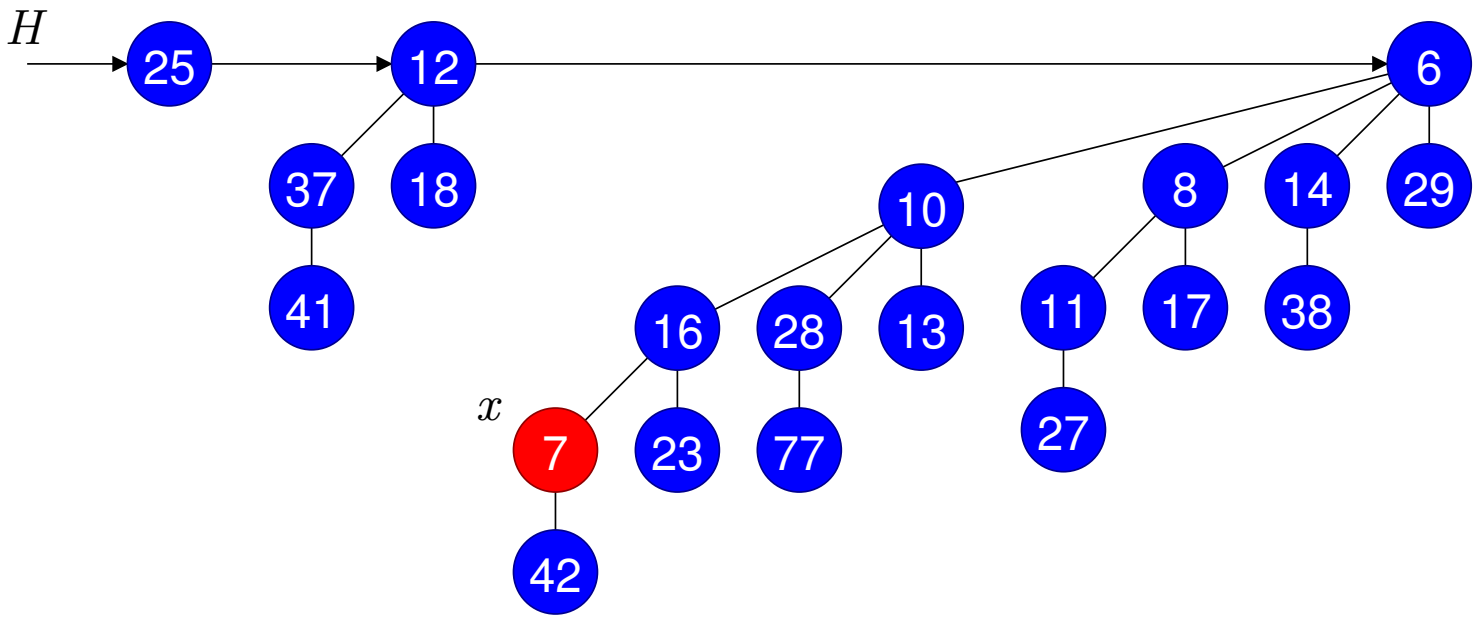
```
1  if k > key(x) then
2     error "neuer Schlüssel ist größer als der alte"
3  key(x) := k
4  y := x
5  z := parent(y)
6  while z ≠ NIL and key(y) < key(z) do
7     Vertausche key(y) und key(z)
8     Vertausche die Satellitendaten von y und z
9     y := z
10    z := parent(y)
```

Beispiel Verkleinern des Schlüssels

Ausgangssituation: Verkleinern des Schlüssels von x auf 7

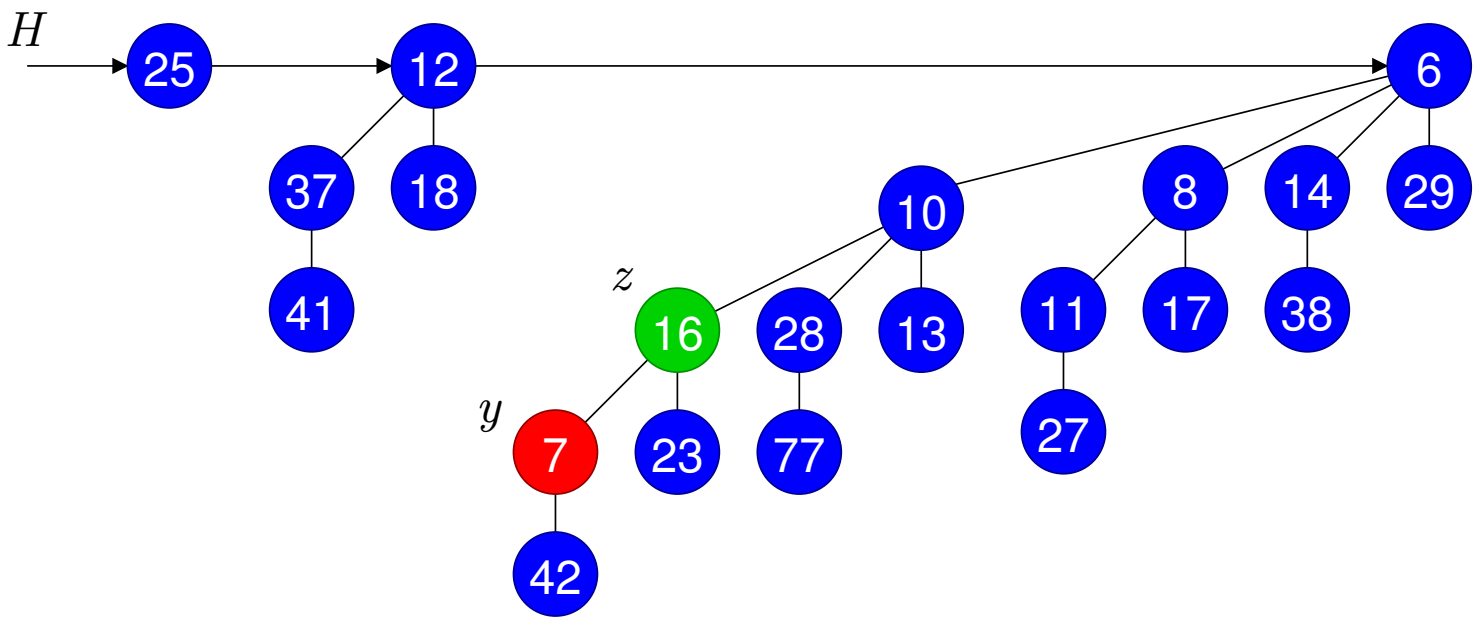


Beispiel Verkleinern des Schlüssels (Forts.)



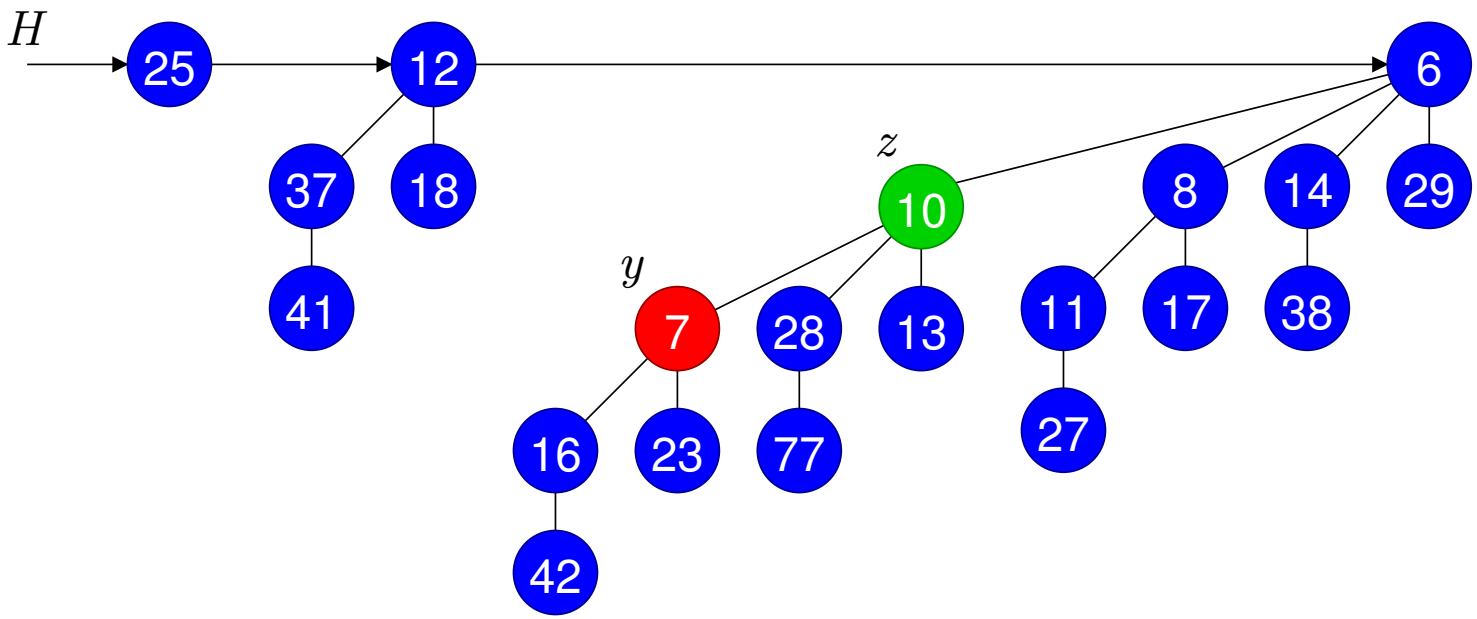
Beispiel Verkleinern des Schlüssels (Forts.)

Zuweisen der Zeiger y und z:



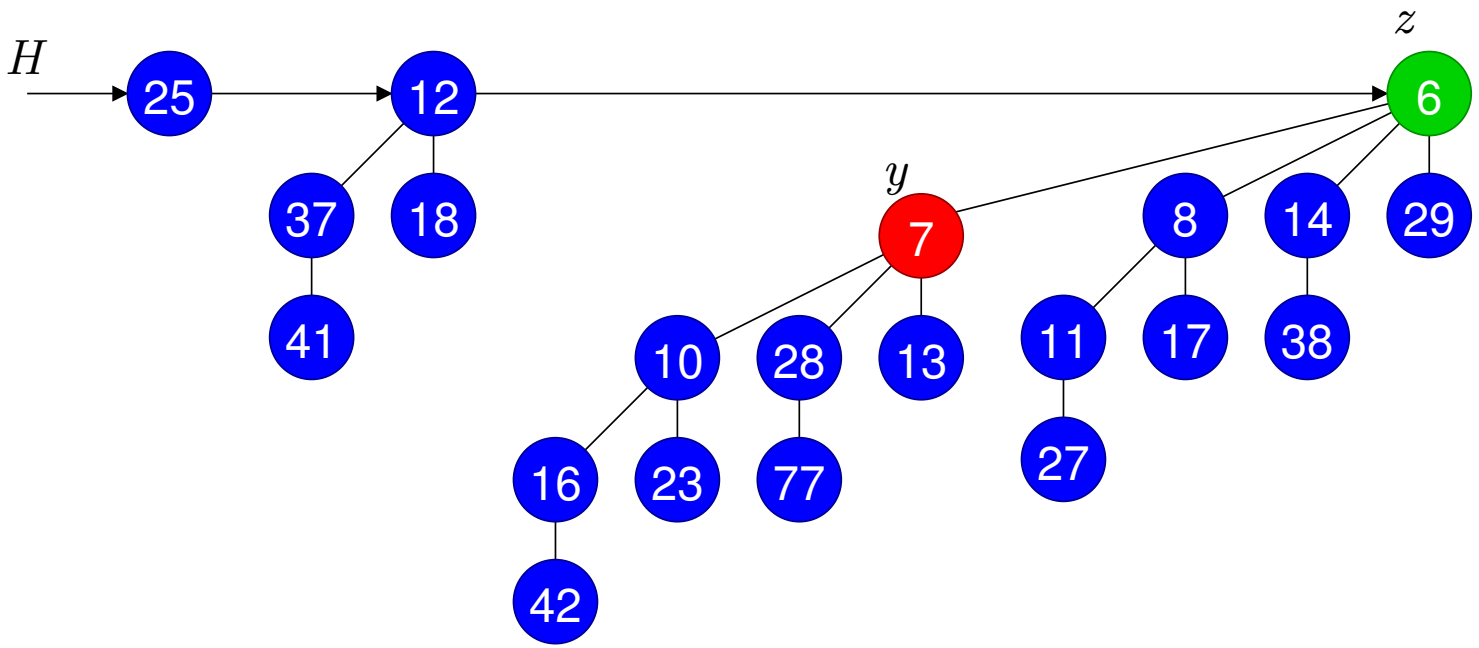
Situation: Schlüssel von y ist kleiner als der von $z \rightsquigarrow$ Daten vertauschen

Beispiel Verkleinern des Schlüssels (Forts.)



Situation: Schlüssel von y ist kleiner als der von $z \rightsquigarrow$ Daten vertauschen

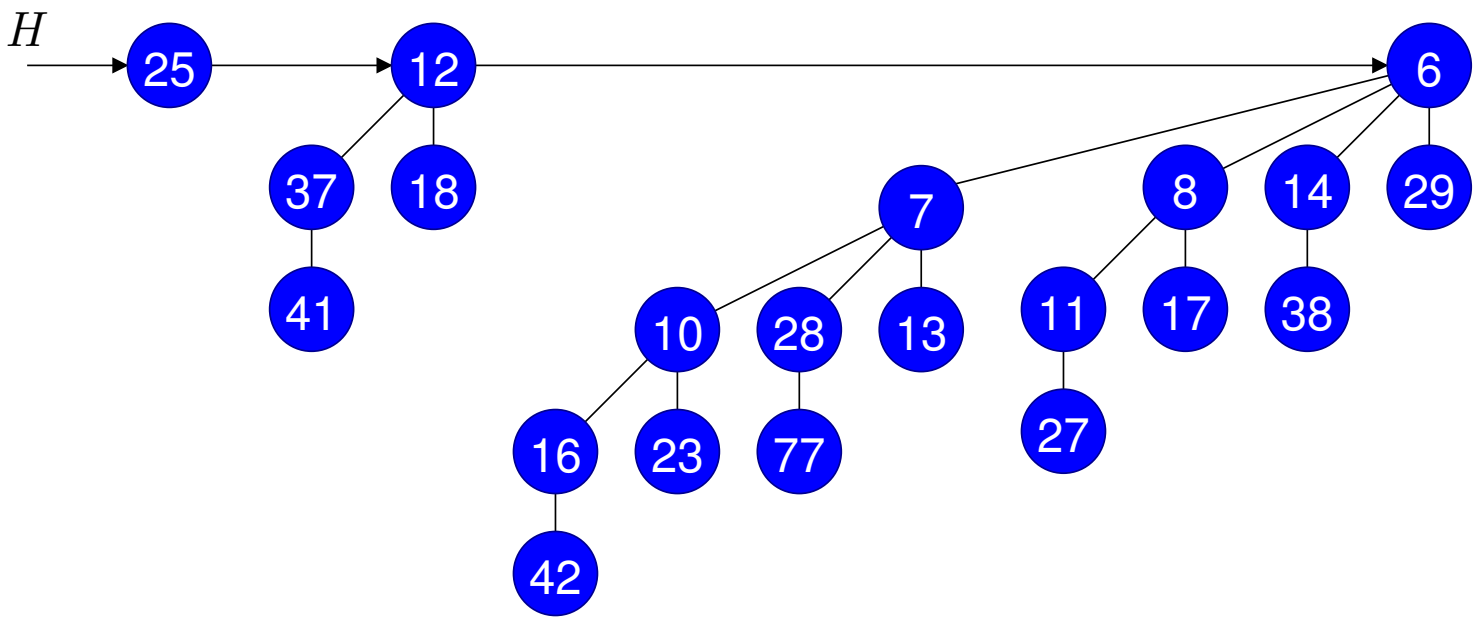
Beispiel Verkleinern des Schlüssels (Forts.)



Situation: Schlüssel von y ist nicht kleiner als der von $z \rightsquigarrow$ Fertig

Beispiel Verkleinern des Schlüssels (Forts.)

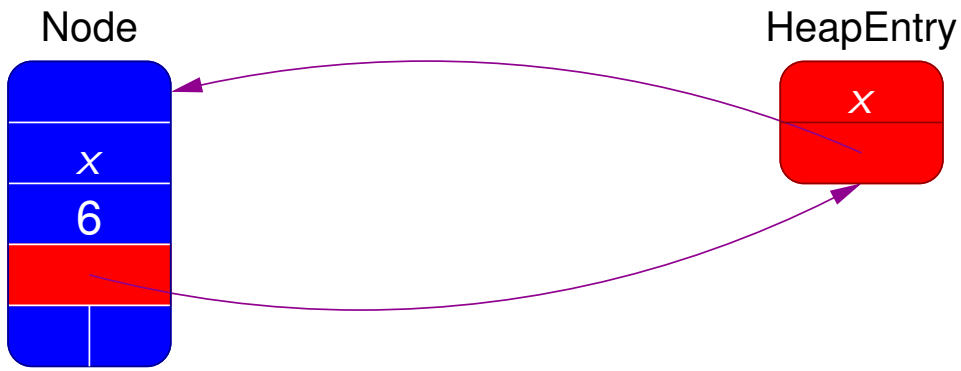
Endergebnis:



Implementierungsaspekte

Feststellung: Um den Schlüssel verkleinern zu können, muss das Element im Binomial Heap gefunden werden

Ansatz: Verknüpfe Element mit dem entsprechenden Knoten im Binomialbaum



Beachte: Der Zeiger vom Knoten zum Element ist notwendig, um den anderen Zeiger bei Verschiebung des Elements im Binomialbaum aktualisieren zu können

Löschen eines Knotens anhand des Schlüssels

Aufgabe: Löschen des Knotens x aus dem Heap

Idee:

1. Mache x zum (einzigen) Minimum des Heaps
2. Lösche das Minimum aus dem Heap

Algorithmus

BinomialHeapDelete(H, x)
Input: Binomial Heap H, zu löschender Knoten x

```
1 BinomialHeapDecreaseKey(H, x, −∞)
2 BinomialHeapExtractMin(H)
```

Laufzeit: $O(\log_2 n)$

Zusammenfassung

- Priority Queues dienen zur Speicherung von Daten mit Prioritäten
- Die Priorität eines Datensatzes wird anhand des Schlüssels ermittelt
- Heap Sort liefert eine Priority Queue auf Basis eines Arrays
- Eine Alternative ist der Binomial Heap, der eine Priority Queue auf Basis von Binomialbäumen bereitstellt
- Beide Ansätze sind optimal, da alle Zugriffe auf die Queue in Laufzeit $O(\log_2 n)$ durchführbar sind