Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 15b (13.06.2018)

Graphenalgorithmen IV

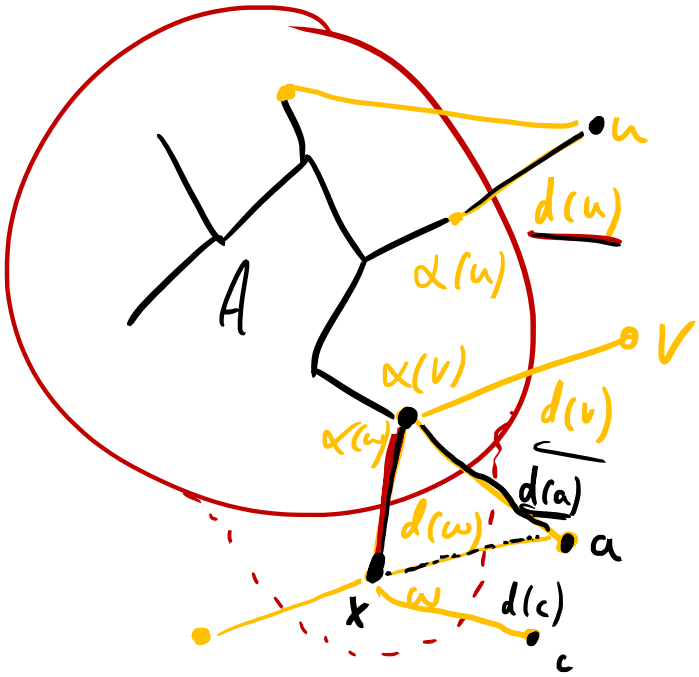
UNI FREIBURG

Fabian Kuhn

Algorithmen und Komplexität

Prims MST-Algorithmus

$A = \emptyset$
while A ist kein Spannbaum **do**
    $e = \{u, v\}$ ist Kante mit kleinstem Gewicht, so dass $u \in A$ und $v \notin A$
    $A = A \cup \{\{u, v\}\}$



z • $\alpha(z) = \text{Null}$
$d(z) = \infty$

Implementierung von Prims Algorithmus

- Knoten, welche im Teilbaum A sind, heißen markiert
- Knoten u:
  - $\alpha(u)$ ist der nächste Nachbar von u im durch A bestimmten Teilbaum
  - $d(u) = \text{dist}\big(u, \alpha(u)\big)$ (oder ∞ falls $\alpha(u) = \text{NULL}$)

for all $u \in V \setminus \{s\}$ **do**
    u.marked = false; $d(u) = \infty$; $\alpha(u) = \text{NULL}$
$d(s) = 0$; $A = \emptyset$ // Wir starten bei Knoten s
while there are unmarked nodes **do**
    u = unmarked node with minimal $d(u)$
    for all unmarked neighbors v of u **do**
        if $w(\{u,v\}) < d(v)$ then
            $\alpha(v) = u$; $d(v) = w(\{u,v\})$
    u.marked = true
    if $u \neq s$ **then** $A = A \cup \big\{\{u, \alpha(u)\}\big\}$

Implementierung von Prims Algorithmus

UNI FREIBURG

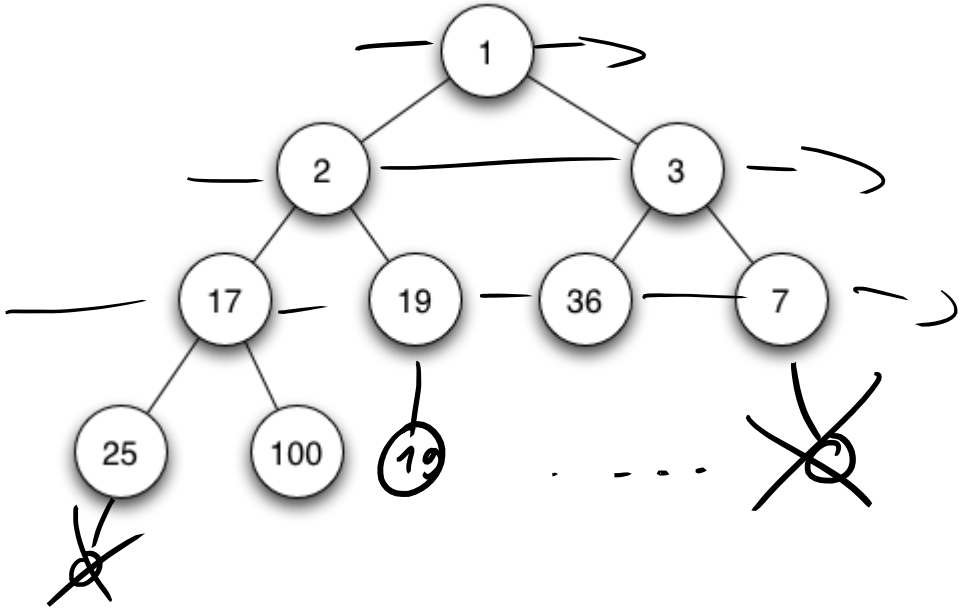
```
H = new priority queue; A = ∅
for all u ∈ V \ {s} do
    H.insert(u, ∞); α(u) = NULL
H.insert(s, 0)

while H is not empty do
    u = H.deleteMin()
    for all unmarked neighbors v of u do
        if w({u,v}) < d(v) then
            H.decreaseKey(v, w({u,v}))
            α(v) = u
    if u ≠ s then A = A ∪ {{u, α(u)}}
```

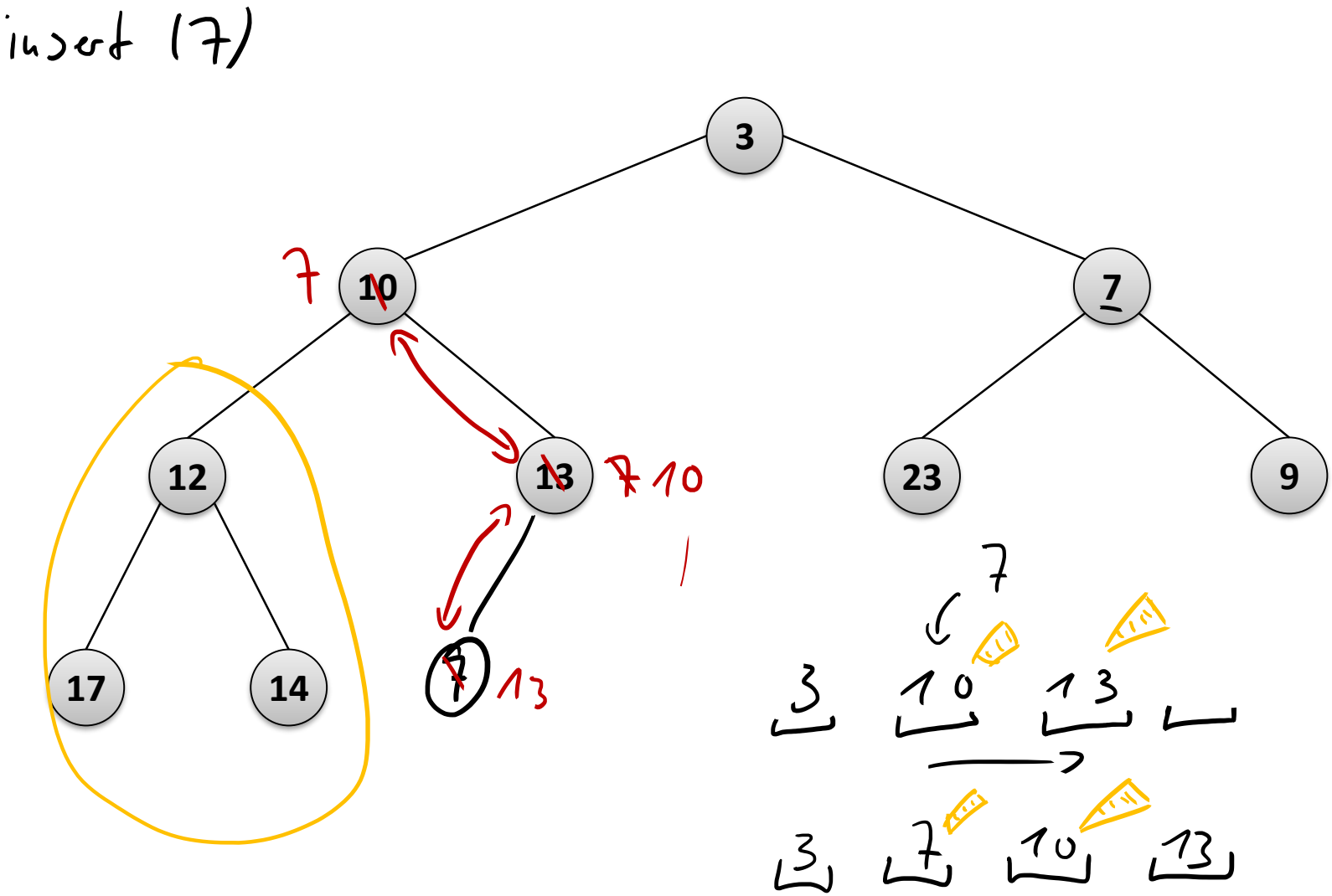
Prioritätswarteschlangen

Implementierung als Binärbaum mit Min-Heap Eigenschaft

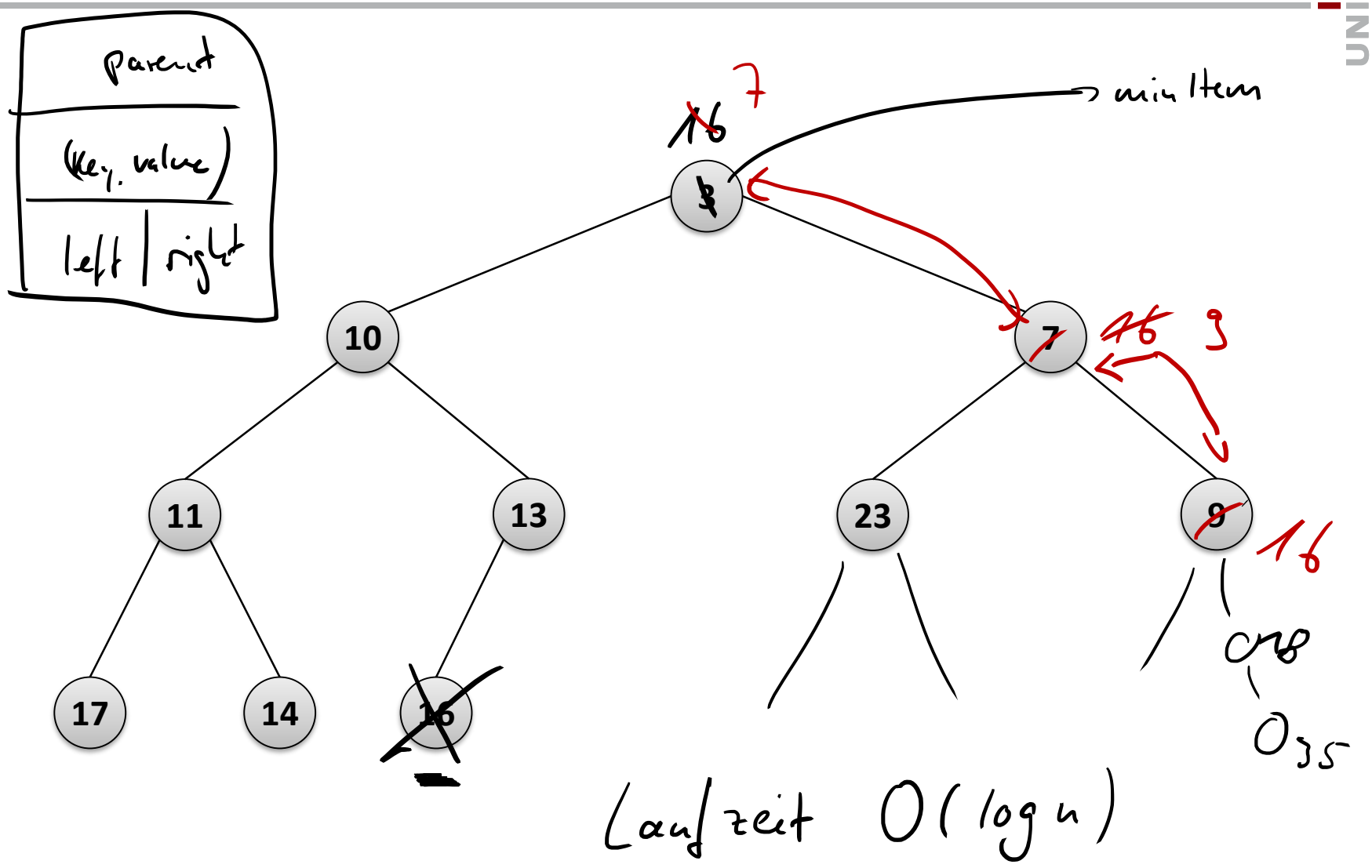
- Die Datenstruktur heißt deshalb oft auch Heap
- Ein Baum hat die Min-Heap Eigenschaft, falls in jedem Teilbaum, die Wurzel den kleinsten Schlüssel hat
- getMin-Operation: trivial!
- Baum wird immer so balanciert wie möglich gehalten



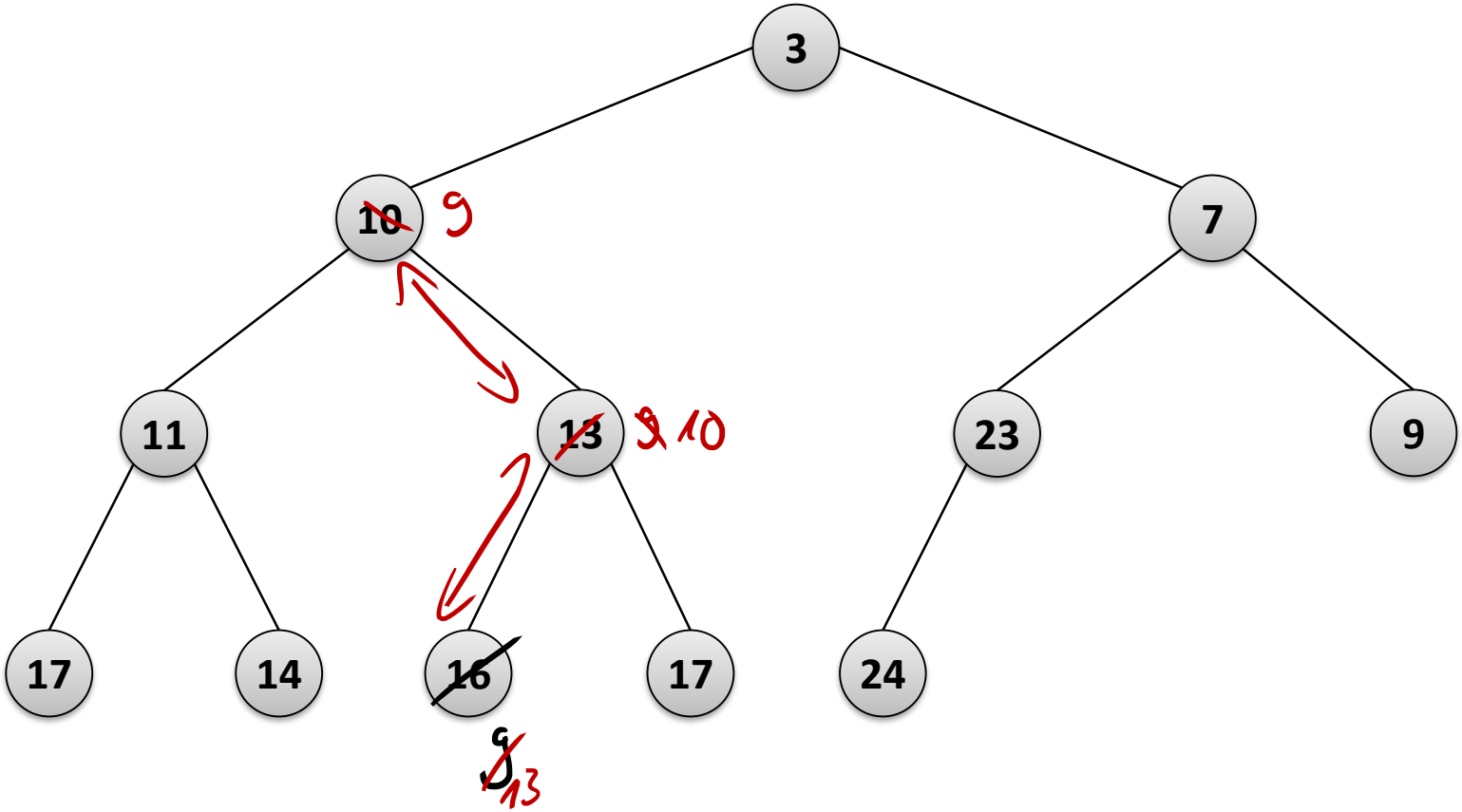
Prioritätswarteschlangen: Einfügen



Prioritätswarteschlangen: Delete-Min

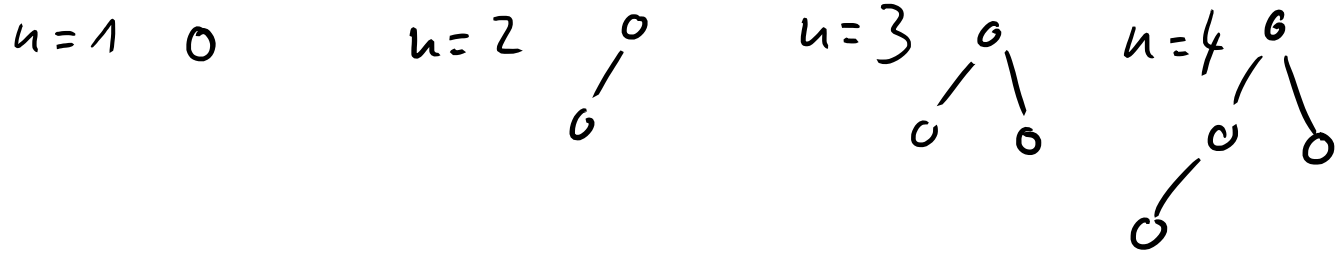


Prioritätswarteschlangen: Decrease-Key



Prioritätswarteschlangen: Analyse

- Die besprochene Variante heißt auch **binärer Heap**
  - durch einen Binärbaum mit Min-Heap-Eigenschaft implementiert
- **Tiefe des Baums** ist immer genau $\lfloor \log_2 n \rfloor$

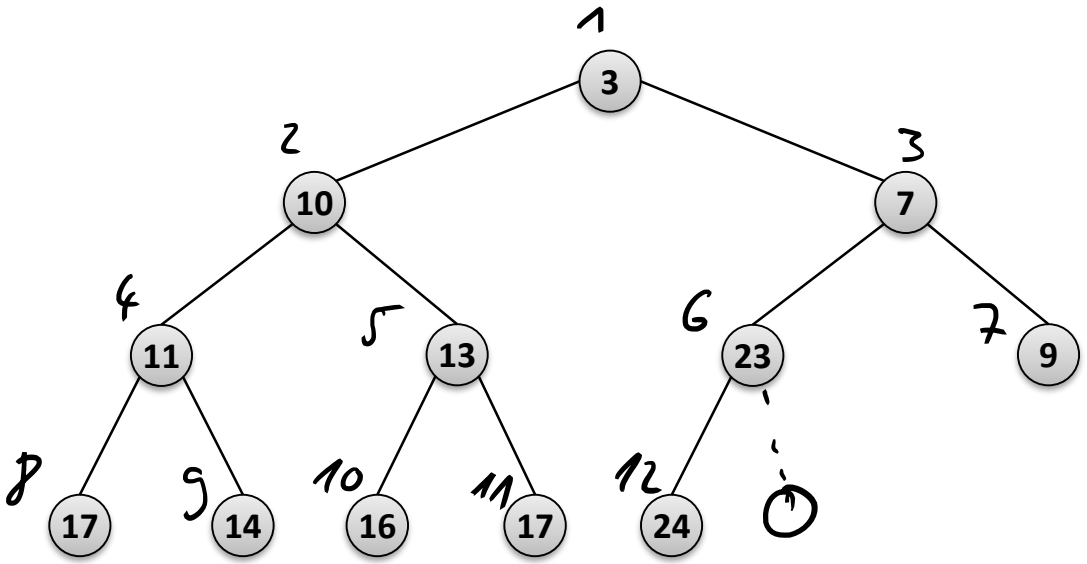


- **Laufzeit aller Operationen: $O(\log n)$**
  - wenn man den Binärbaum irgendwie vernünftig implementiert
  - man muss immer höchstens einmal den Binärbaum hoch (bei insert, decreaseKey) oder runter (bei deleteMin)
  - Wir werden gleich eine elegante Art sehen, den binären Heap zu impl.

Binäre Heaps, Array-Implementierung

Idee: Speichere **alles in ein Array**

- Das geht, weil der Baum perfekt balanciert ist



Nummer = Position im Array

0	1	2	3	4	5	6	7	8	9	10	11	12
⊥	3	10	7	11	13	23	9	17	14	16	17	24

linkes Kind von i (= Knoten an Pos. i) ist an Pos. $2i$

rechtes Kind von i ist an Pos. $2i+1$

Parent von j ist an Pos. $\lfloor \frac{j}{2} \rfloor$

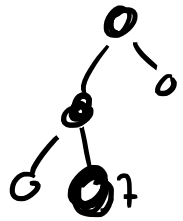
Binäre Heaps, Array-Implementierung

Positionen im Array (bei n Elementen):

- Wurzel: 1, letzter Eintrag: n, Parent von i: $\lfloor i/2 \rfloor$
- linkes Kind von i: $2i$, rechtes Kind von i: $2i + 1$

Pseudocodes:

Elemente sind in Array A an Positionen $1, \dots, n$ gespeichert

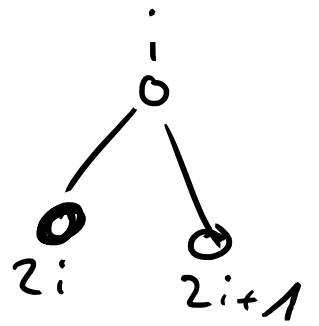


```
insert(x):
   n = n + 1
   A[n] = x
   i = n
   while (i > 1) and (A[i] < A[i/2]) do
        swap(A[i], A[i/2])
        i = i/2
```

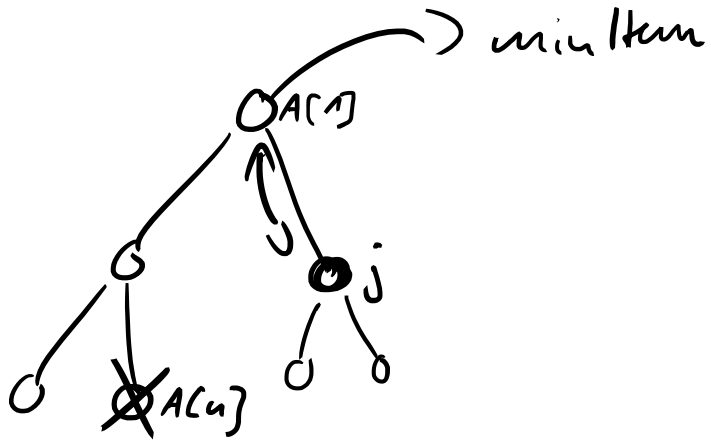
Ganzzahldivision

Binäre Heaps: Pseudocode Delete-Min

```
smallest(i):   // returns index of smallest key among i and children
  j = i
  if (2*i <= n) and (A[2*i] < A[i]) then
    j = 2*i
  if (2*i+1 <= n) and (A[2*i+1] < A[j]) then
    j = 2*i + 1
  return j
```




```
deleteMin():
  minItem = A[1]
  A[1] = A[n]; n = n – 1;
  i = 1; j = smallest(i)
  while j ≠ i do
      swap(A[i], A[j])
      i = j
      j = smallest(i)
  return minItem
```



Binäre Heaps: Pseudocode Decrease-Key

```
decreaseKey(i, x):
  if A[i]>x then
    A[i]=x
    while (i > 1) and (A[i] < A[i/2]) do
      swap(A[i], A[i/2])
      i = i/2
```

Bei MST: decreaseKey(v, x)

d(v)

Heapsort

- Die Array-Implementierung von Heaps (Prioritätswarteschlangen) gibt auch einen weiteren effizienten Sortieralgorithmus

Heapsort (H ist ein binärer Heap, sortiere Array A)

```
H = new BinaryHeap()
for i = 0 to n - 1 do
    H.insert(A[i])          O(log n)
for i = 0 to n - 1 do
    A[i] = H.deleteMin()    O(log n)
```

- **Laufzeit:** $O(n \log n)$

Vergleichsbasierte Sortieralgorithmen: untere Schranke $\Omega(n \log n)$

Prims Algorithmus: Bessere Laufzeit

Laufzeit mit binären Heaps: $O(m \cdot \log n)$ $= O(m + m \log n)$

- $n \leq m + 1$ insert-Operationen und deleteMin-Operationen
- $\leq m$ decreaseKey-Operationen

$G = (V, E)$, $n = |V|$, $m = |E|$, $m \in O(n^2)$

$m \geq n - 1$

Kruskal $O(m \log n)$ nicht effizient sortieren, $O(m \alpha(m, n))$ wenn man in lin. Zeit sortieren kann

Beste Implementierung von Prioritätswarteschlangen:

- Fibonacci Heaps (siehe Vorlesung Algorithmentheorie)
- Laufzeit der Operationen (deleteMin, decreaseKey amortisiert)

insert: $O(1)$, **deleteMin:** $O(\log n)$, **decreaseKey:** $O(1)$

n_i mal n_k mal n_d mal: $O(n_i + n_k + n_d \log(n))$

Laufzeit mit Fibonacci Heaps: $O(m + n \cdot \log n)$

- $n \leq m + 1$ insert-Operationen und deleteMin-Operationen
- $\leq m$ decreaseKey-Operationen

$m \in \Omega(n)$, $m \in O(n^2)$

$m \in O(n)$: gleiche Laufzeit

Lower bound für MST: $\Omega(m)$