

# 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  $v \notin A$

$A = A \cup \{u, v\}$

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

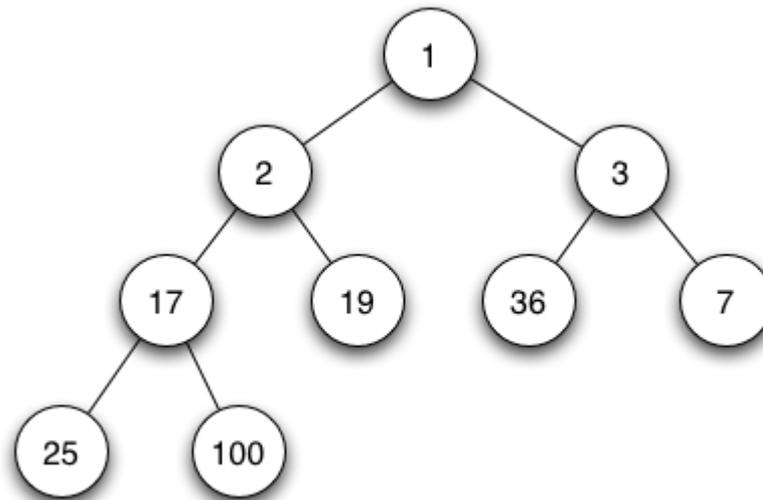
```
for all  $u \in V \setminus \{s\}$  do  
     $u.\text{marked} = \text{false}; d(u) = \infty; \alpha(u) = \text{NULL}$   
 $d(s) = 0; A = \emptyset$  // Wir starten bei Knoten  $s$   
while there are unmarked nodes do  
     $u =$  unmarked node with minimal  $d(u)$   
    for all unmarked neighbors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

# Implementierung von Prim's Algorithmus

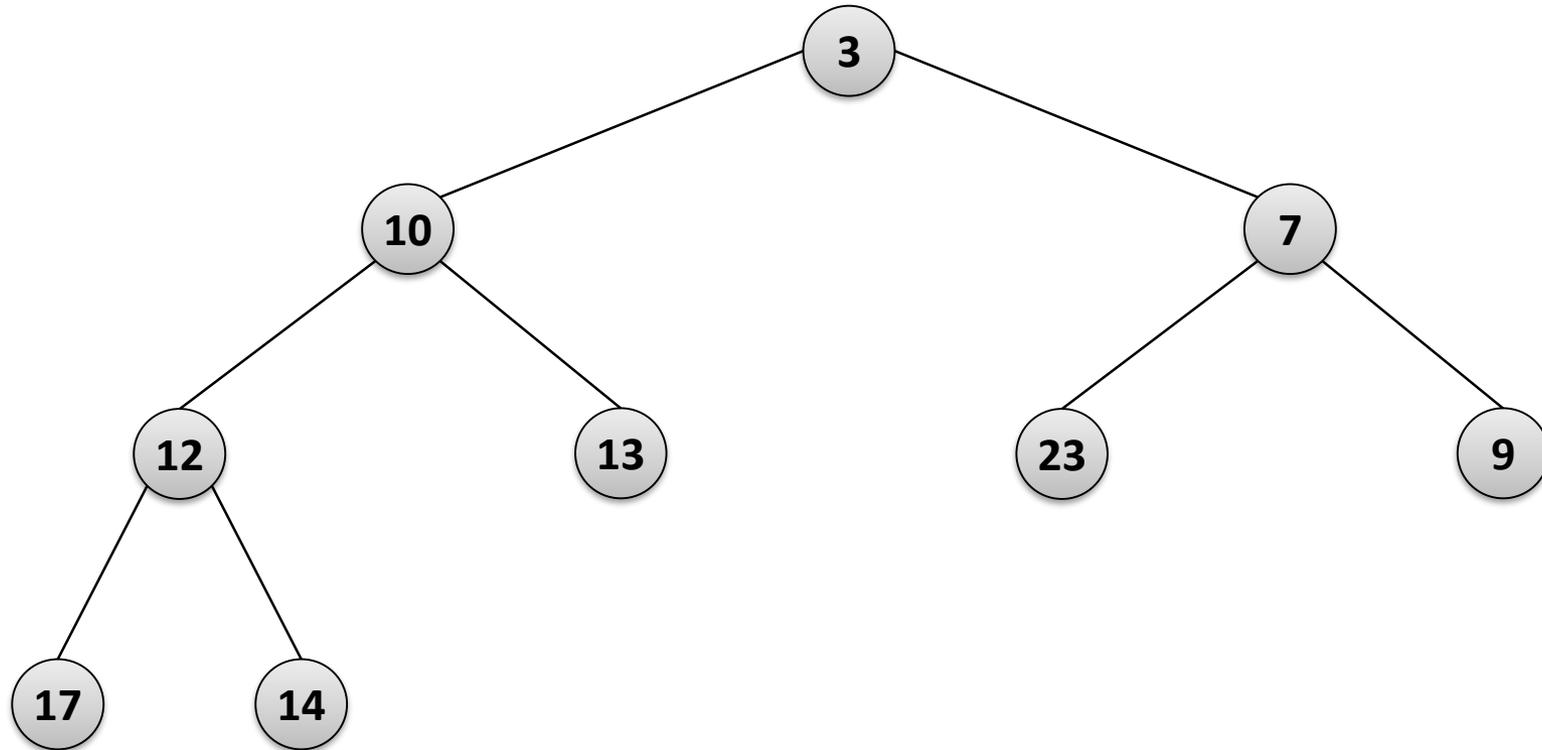
```
H = new priority queue;  $A = \emptyset$   
for all  $u \in V \setminus \{s\}$  do  
    H.insert( $u, \infty$ );  $\alpha(u) = \text{NULL}$   
H.insert( $s, 0$ )  
  
while H is not empty do  
     $u = H.\text{deleteMin}()$   
    for all unmarked neighbors  $v$  of  $u$  do  
        if  $w(\{u, v\}) < d(v)$  then  
            H.decreaseKey( $v, w(\{u, v\})$ )  
             $\alpha(v) = u$   
    if  $u \neq s$  then  $A = A \cup \{u, \alpha(u)\}$ 
```

## Implementierung als Binärbaum mit Min-Heap Eigenschaft

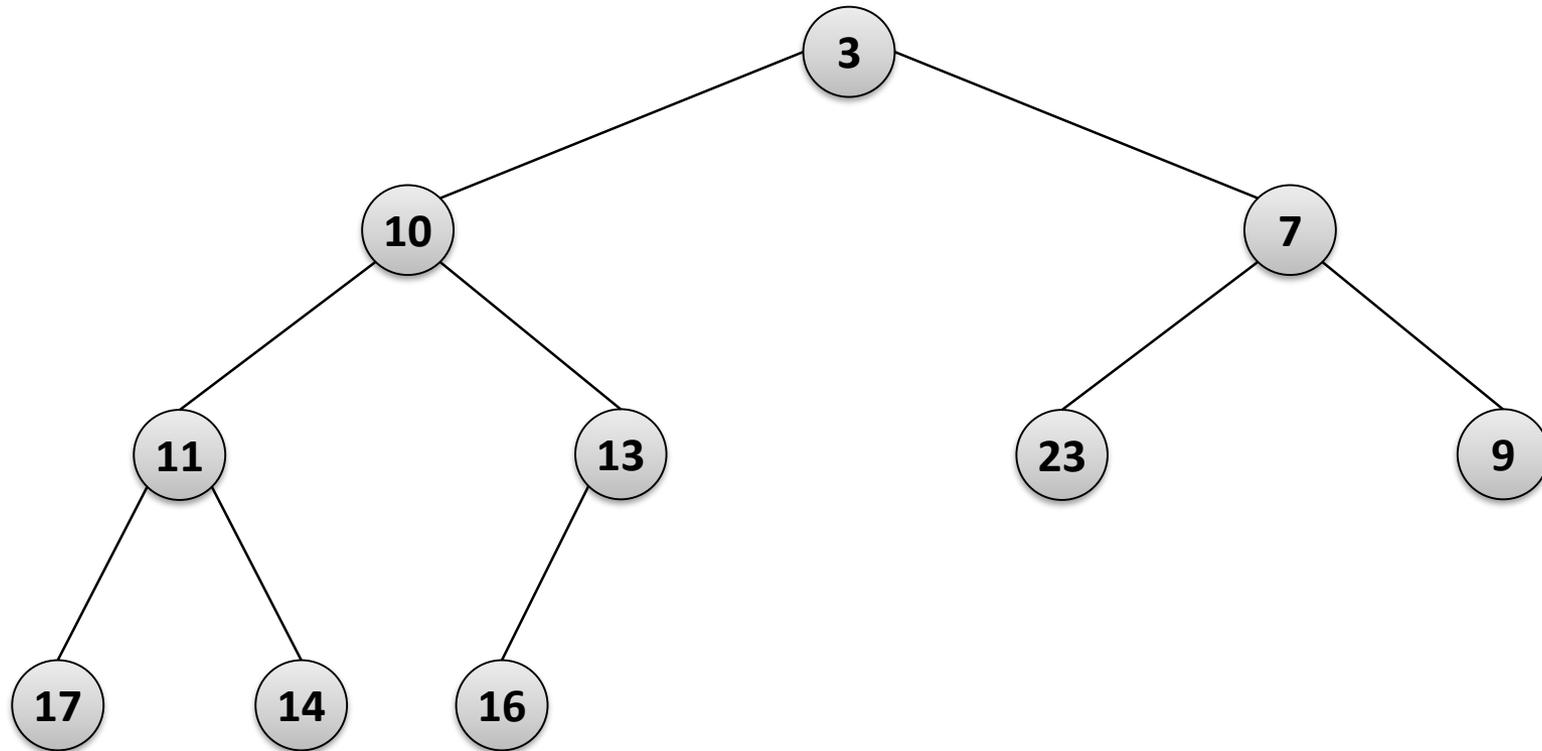
- Die Datenstruktur heißt deshalb oft auch Heap
- Ein Baum hat die Min-Heap Eigenschaft, falls **in jedem Teilbaum**, die **Wurzel** den **kleinsten Schlüssel** hat
- getMin-Operation: trivial!
- Baum wird immer so balanciert wie möglich gehalten



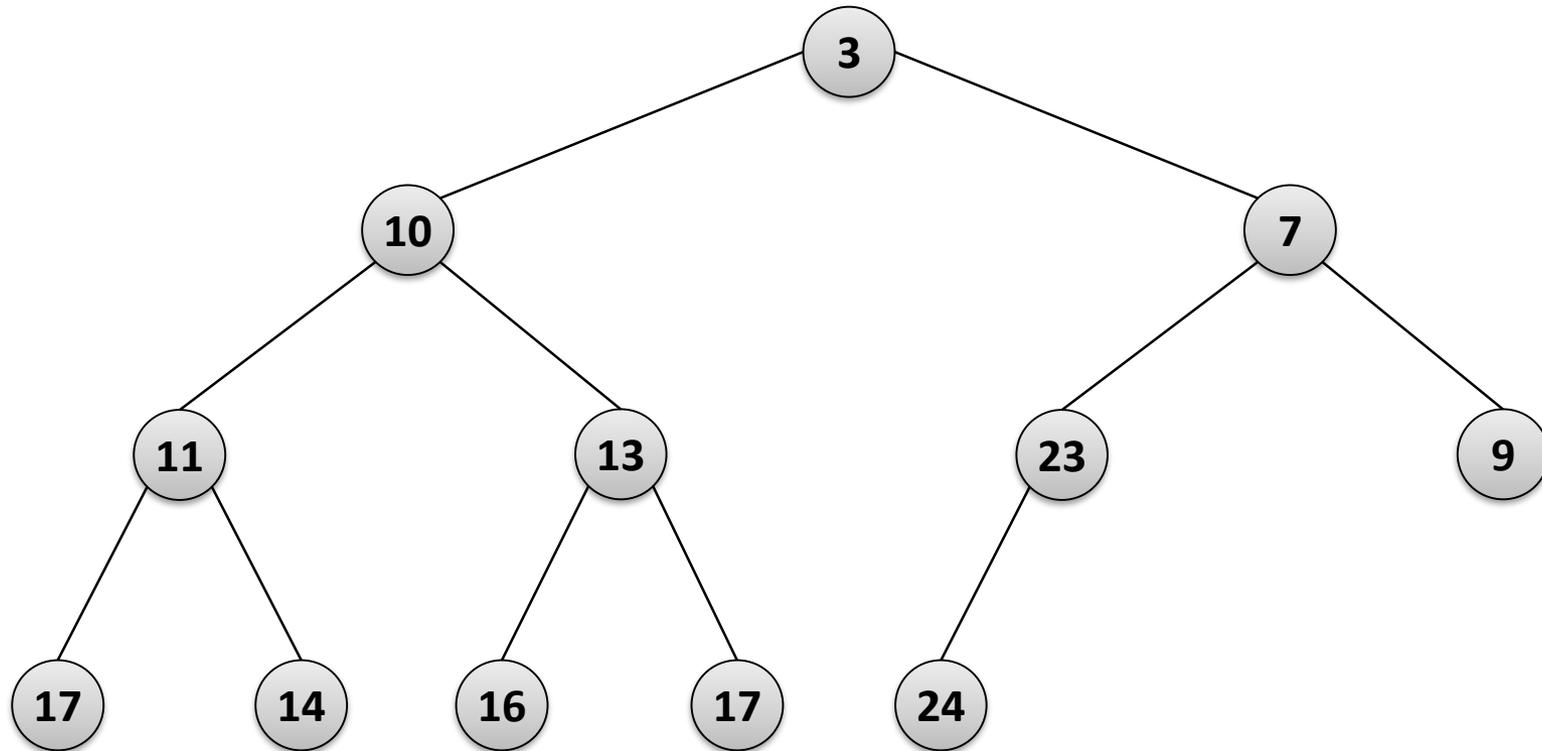
# Prioritätswarteschlangen: Einfügen



# Prioritätswarteschlangen: Delete-Min



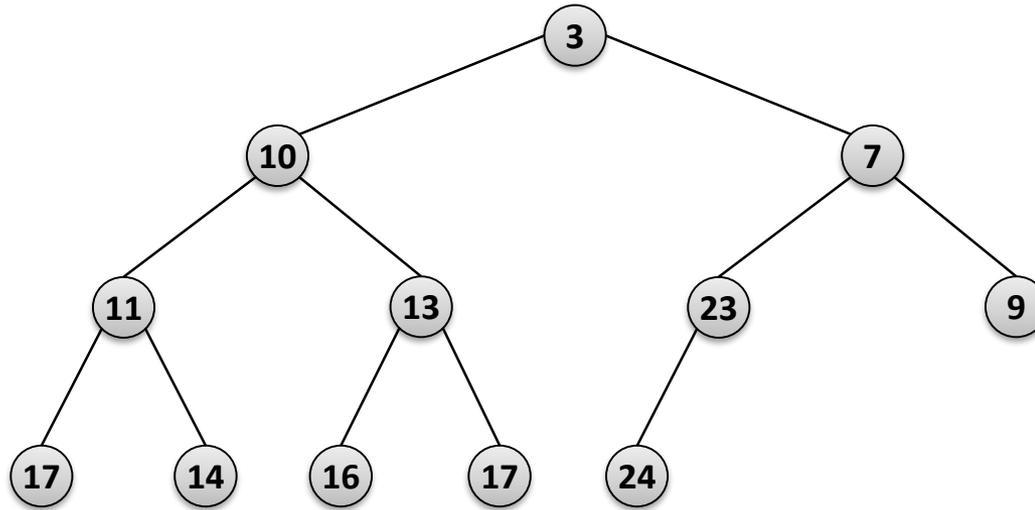
# Prioritätswarteschlangen: Decrease-Key



- 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.

## Idee: Speichere alles in ein Array

- Das geht, weil der Baum perfekt balanciert ist



## 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
```

# Binäre Heaps: Pseudocode Delete-Min

**smallest(i):** // returns index of smallest key among  $i$  and children

$j = i$

if  $(2*i \leq n)$  and  $(A[2*i] < A[i])$  then

$j = 2*i$

if  $(2*i+1 \leq n)$  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  $j \neq i$  do

$swap(A[i], A[j])$

$i = j$

$j = smallest(i)$

return  $minItem$

**decreaseKey(i, x):**

if  $A[i] > x$  then

$A[i] = x$

    while  $(i > 1)$  and  $(A[i] < A[i/2])$  do

        swap( $A[i]$ ,  $A[i/2]$ )

$i = i/2$

- 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])
for i = 0 to n - 1 do
    A[i] = H.deleteMin()
```

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

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

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

## Beste Implementierung von Prioritätswarteschlangen:

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

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

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

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