Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 16 (17.6.2016)

Graphenalgorithmen IV
(MST, Prioritätswarteschlangen)

UNI FREIBURG

Fabian Kuhn

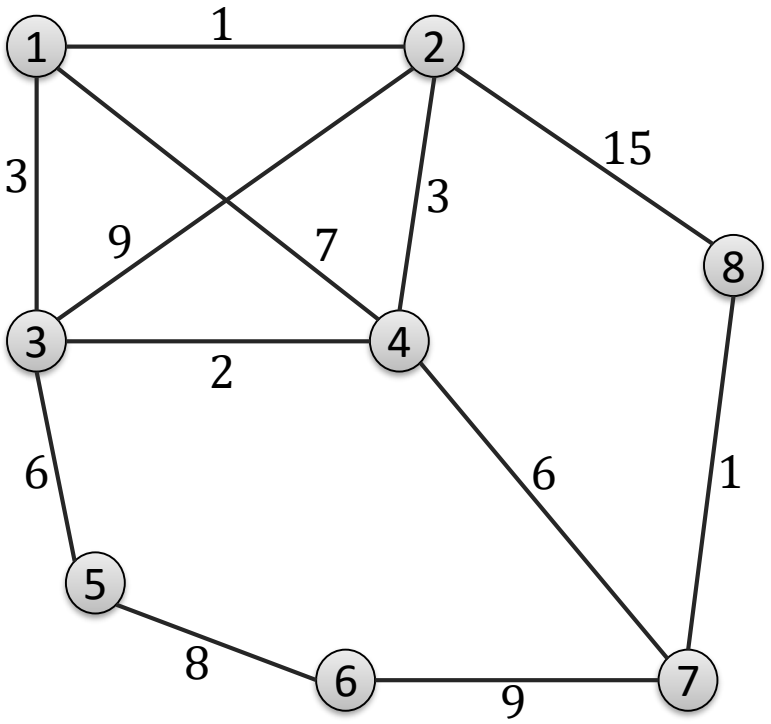
Algorithmen und Komplexität

Minimaler Spannbaum

Gegeben: Zus.-hängender, ungerichteter Graph $G = (V, E, w)$ mit Kantengewichten $w : E \to \mathbb{R}$

Minimaler Spannbaum $T = (V, E_T)$:

- Spannbaum mit kleinstem Gesamtgewicht



Basis-MST-Algorithmus

Invariante:
Algorithmus hat zu jeder Zeit eine Kantenmenge A, so dass A Teilmenge eines minimalen Spannbaums ist.

Basis-MST-Algorithmus:

$A = \emptyset$
while A ist kein Spannbaum **do**
    Finde sichere Kante $\{u, v\}$ für A
    $A = A \cup \{\{u, v\}\}$
return A

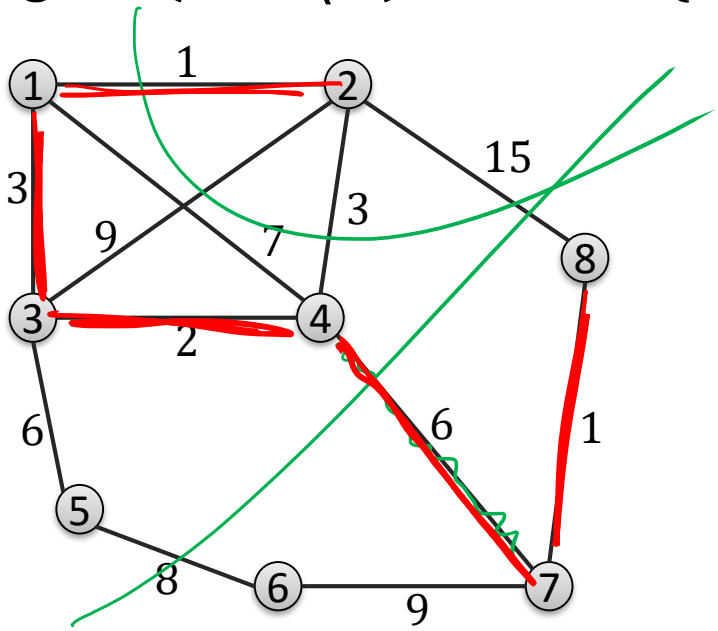
- Invariante ist eine gültige Schleifeninvariante
- Invariante + Abbruchbedingung $\Longrightarrow$ A ist ein MST!

Sichere Kanten

Annahmen:

- $G = (V, E, w)$ ist zus.-h., unger. Graph mit Kantengewichten $w(e)$
- A ist Teilmenge (Teilgraph) eines MST

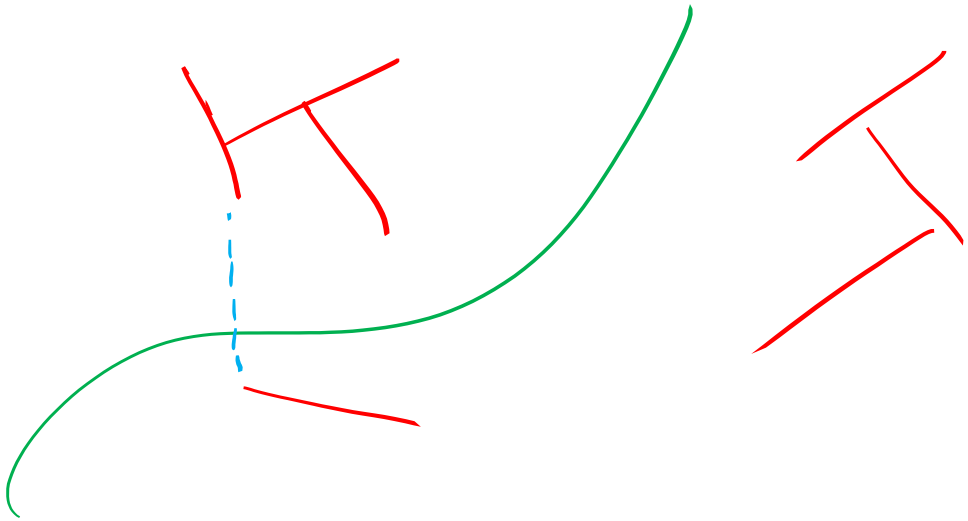
Theorem: Sei $(S, V \setminus S)$ ein Schnitt, so dass A keine Schnittkanten enthält und sei $\{u, v\}$, $u \in S$, $v \in V \setminus S$ eine leichte Schnittkante bezüglich $(S, V \setminus S)$. Dann ist $\{u, v\}$ eine sichere Kante für A.



Kruskal's MST-Algorithmus

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

- Wir müssen zeigen, dass e eine sichere Kante für A ist



Union-Find / Disjoint Sets Datentyp

Verwaltet eine Partition von Elementen

Operationen:

- *create* : erzeugt eine leere Union-Find-DS
- *U.makeSet(x)* : fügt Menge $\{x\}$ zur Partition hinzu
- *U.find(x)* : gibt Menge mit Element x zurück
- *U.union(S1, S2)* : vereinigt die Mengen $S1$ und $S2$

- Details dazu werden in der Algorithmentheorie besprochen…

Kruskal mit Union-Find

Kruskals Algorithmus

1. $A = \emptyset$
2. U = create new
3. **for all** $u \in V$ **do**
4. U.makeSet(u)
5. Sortiere Kanten aufsteigend nach Kantengewicht
6. **for all** $e = \{u, v\} \in E$ (in sorted order) **do**
7. S_u = U.find(u); S_v = U.find(v)
8. **if** $S_u \neq S_v$ **then**
9. $A = A \cup \{e\}$
10. U.union(S_u, S_v)

Kruskal: Laufzeitanalyse

UNI FREIBURG

Beste Union-Find Datenstruktur

- Laufzeit für m Union-Find-Operationen auf n Elementen (n makeSet-Operationen):

$$\boldsymbol{O\big(m \cdot \alpha(m,n)\big)}$$

- $\alpha(m,n)$ ist die Inverse der Ackermannfunktion und wächst extrem langsam (für alle halbwegs vernünftigen m, n, $\alpha(m,n) \leq 5$)

Laufzeit Kruskal

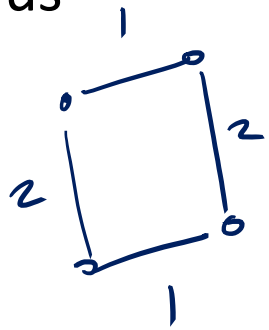
- Kanten sortieren: $O(m \cdot \log n)$
- Union-Find-Operationen: $O\big(m \cdot \alpha(m,n)\big)$
- Insgesamt: $O(m \cdot \log n)$
  - besser, falls Kantengewichte schneller sortiert werden können

Prims MST Algorithmus

- Sollte man wohl eigentlich Jarvíks Algorithmus nennen
  - wurde 1957 von Prim und bereits 1930 von Jarvík publiziert

- Eine zweite Implementierung des Basis-Algorithmus

$A = \emptyset$
while A ist kein Spannbaum **do**
    Finde sichere Kante $\{u, v\}$ für A
    $A = A \cup \{\{u, v\}\}$
return A



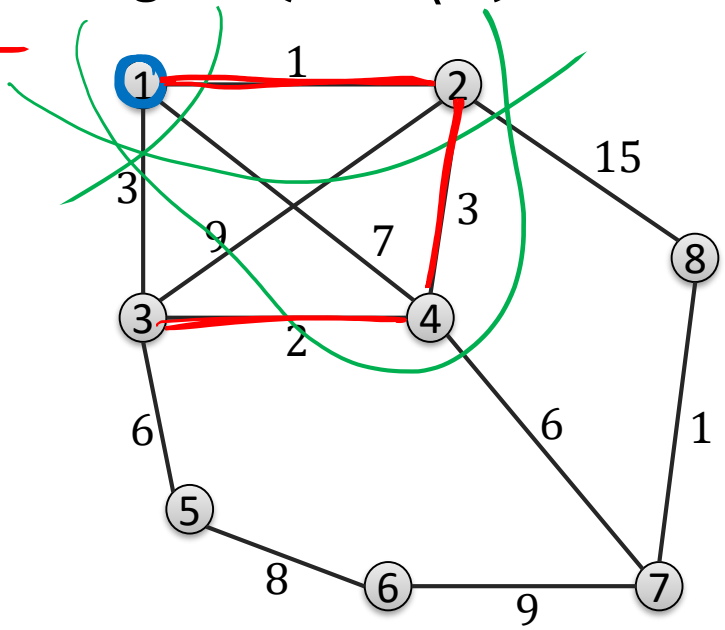
- **Idee:** A ist immer ein zusammenhängender Teilbaum
  - Starte bei einem beliebigen Knoten $s \in V$
  - Baum wächst von s aus, indem immer eine leichte Schnittkante des durch A induzierten Schnitts hinzugefügt wird.

Sichere Kanten

Annahmen:

- $G = (V, E, w)$ ist zus.-h., unger. Graph mit Kantengewichten $w(e)$
- A ist Teilmenge (Teilgraph) eines MST

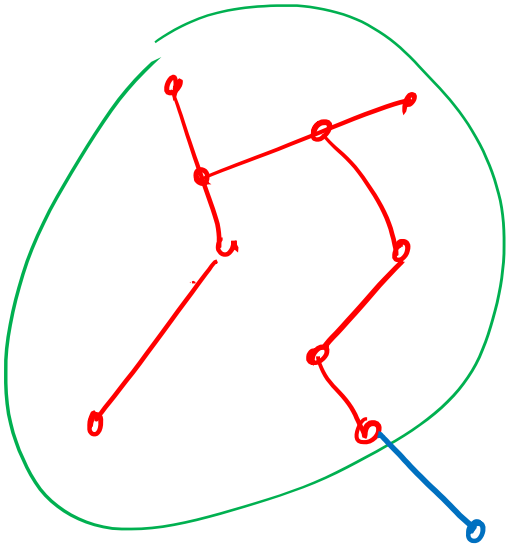
Theorem: Sei $(S, V \setminus S)$ ein Schnitt, so dass A keine Schnittkanten enthält und sei $\{u, v\}$, $u \in S$, $v \in V \setminus S$ eine leichte Schnittkante bezüglich $(S, V \setminus S)$. Dann ist $\{u, v\}$ eine sichere Kante für A.



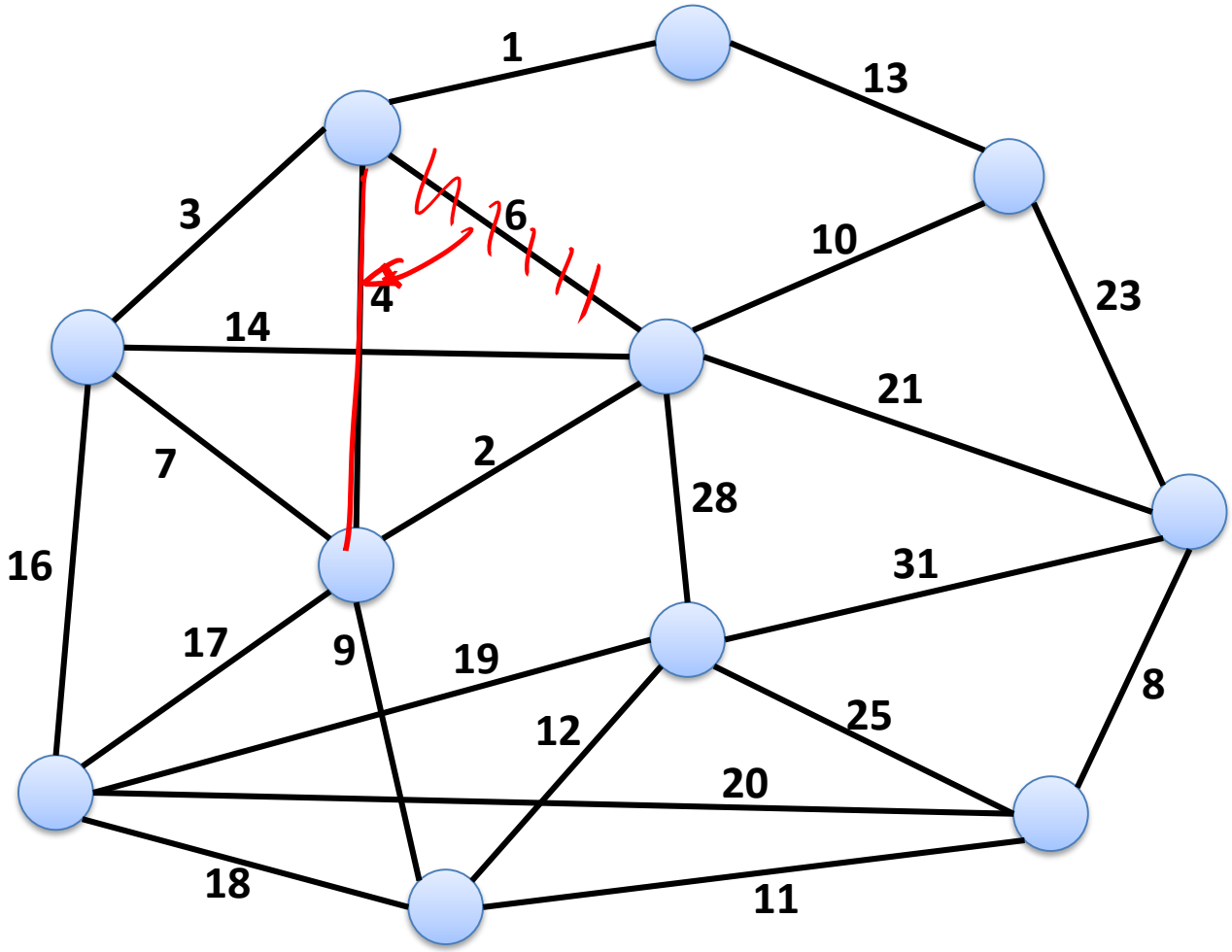
Prims MST-Algorithmus

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

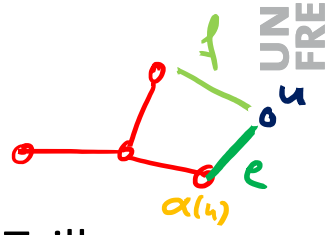
- Wir müssen zeigen, dass e eine sichere Kante für A ist



Prims MST-Algorithmus: Beispiel



Implementierung von Prims Algorithmus

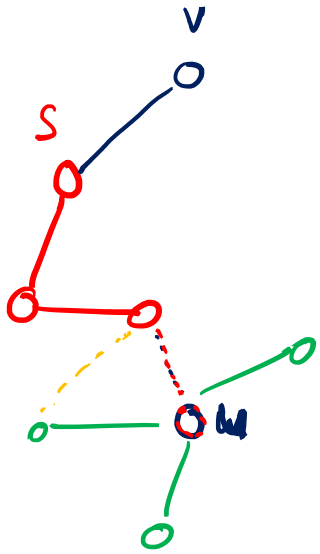


- Knoten, welche im Teilbaum A sind, heissen markiert
- Knoten u:
  - $\alpha(u)$ ist der nächste Nachbar von u im durch A bestimmten Teilbaum
  - $d(u) = \text{dist}(u, \alpha(u))$ (oder ∞ falls $\alpha(u) = \text{NULL}$)

s.marked = false

for all $u \in V \setminus \{s\}$ **do**
    u.marked = false; $d(u) = \infty$; $\alpha(u)$ = 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** $A = A \cup \{\{u, \alpha(u)\}\}$

$\alpha(v) = s$



Prioritätswarteschlange

UNI FREIBURG

Heap / Priority Queue:

- Verwaltet eine Menge von *(key,value)*-Paaren

Operationen:

- *create* : erzeugt einen leeren Heap
- *H.insert(x, key)* : fügt Element *x* mit Schlüssel *key* ein
- *H.getMin()* : gibt Element mit kleinstem Schlüssel zurück
- *H.deleteMin()* : löscht Element mit kleinstem Schlüssel (gibt Element mit kleinstem Schlüssel zurück)
- *H.decreaseKey(x, newkey)* : Falls *newkey* kleiner als der aktuelle Schlüssel von *x* ist, wird der Schlüssel von *x* auf *newkey* gesetzt

Implementierung von Prims Algorithmus

s.marked = false

benutze priority queue, um die unmarkierten Knoten zu verwalten

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

H = leere pr. queue

füge alle Knoten mit Schlüssel dem ein

while there are unmarked nodes **do**

H ist nicht leer

    u = unmarked node with minimal $d(u)$

    u = H.deleteMin()

    for all unmarked neighbors v of u **do**
        if $w(\{u,v\}) < d(v)$ then
            $\alpha(v) = u$; $d(v) = w(\{u,v\})$

            decrease key

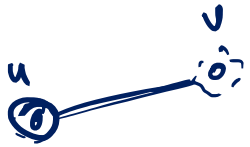
    u.marked = true
    if $u \neq s$ **then** $A = A \cup \{\{u, \alpha(u)\}\}$

Implementierung von Prims Algorithmus

H = new priority queue; $A = \emptyset$
for all $u \in V \setminus \{s\}$ **do**
    H.insert(u, ∞); $\alpha(u)$ = NULL
H.insert(s, 0)

#insert : n

G : n Knoten
m Kanten



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, $w(\{u,v\})$)
            $\alpha(v) = u$
    if $u \neq s$ **then** $A = A \cup \{\{u, \alpha(u)\}\}$

#deleteMin : n

#decreaseKey : m
(worst case: n^2)

Analyse von Prims MST-Algorithmus

UNI FREIBURG

Anzahl Priority Queue Operationen

- **create** 1
- **insert** n
- **getMin / deleteMin** n
- **decreaseKey** m

Gesamt-Laufzeit

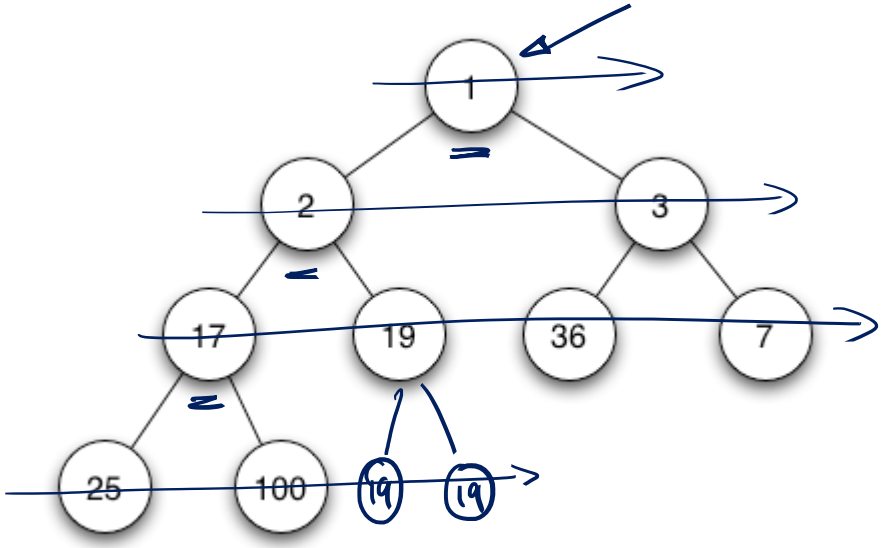
$$O(n \underbrace{(\text{Zeit f. insert/deleteMin})}_{O(\log n)} + m \underbrace{(\text{Zeit f. decr.key})}_{O(\log n)})$$

$$\in O(m \log n)$$

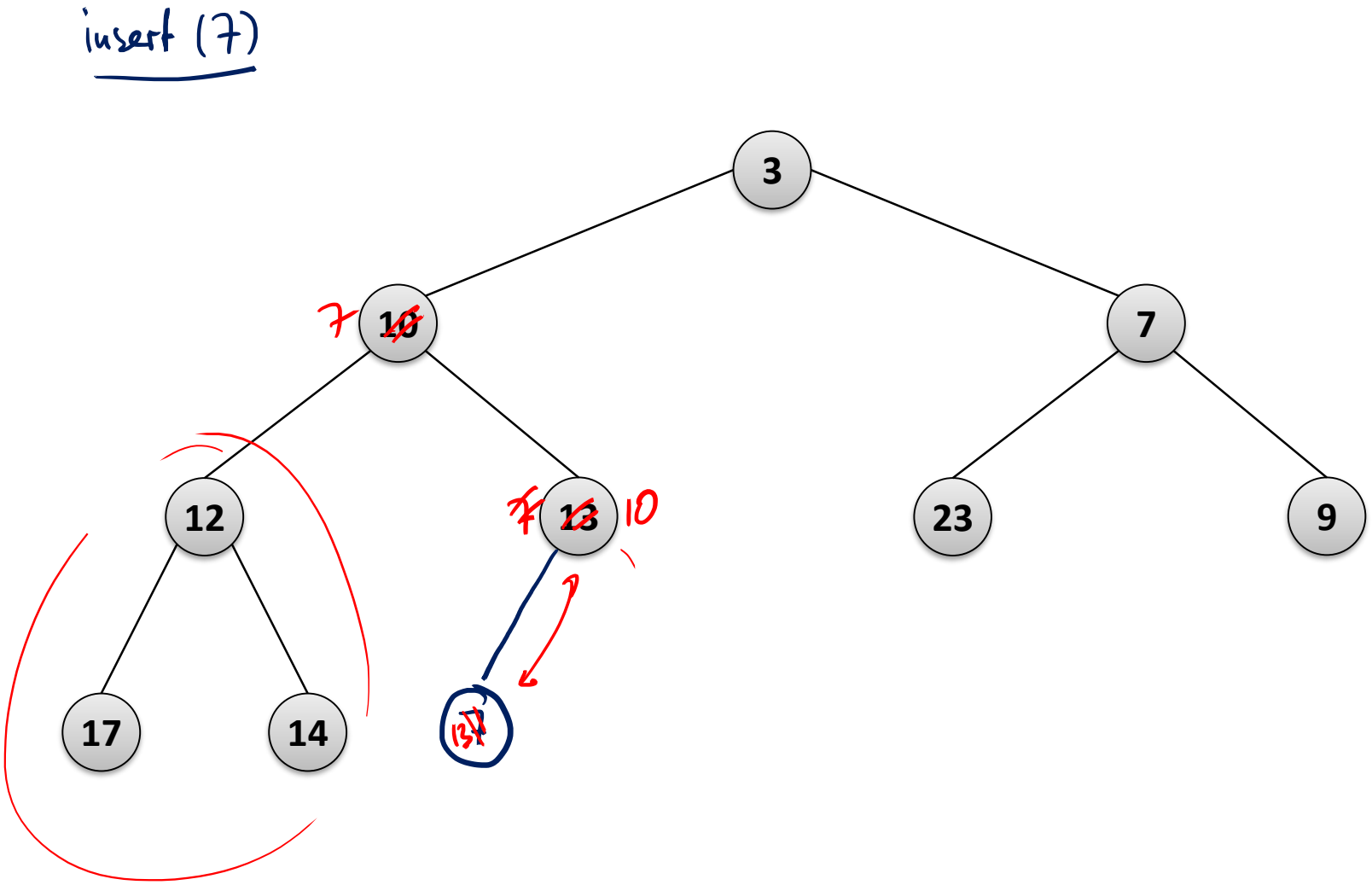
Prioritätswarteschlangen

Implementierung als Binärbaum mit Min-Heap Eigenschaft

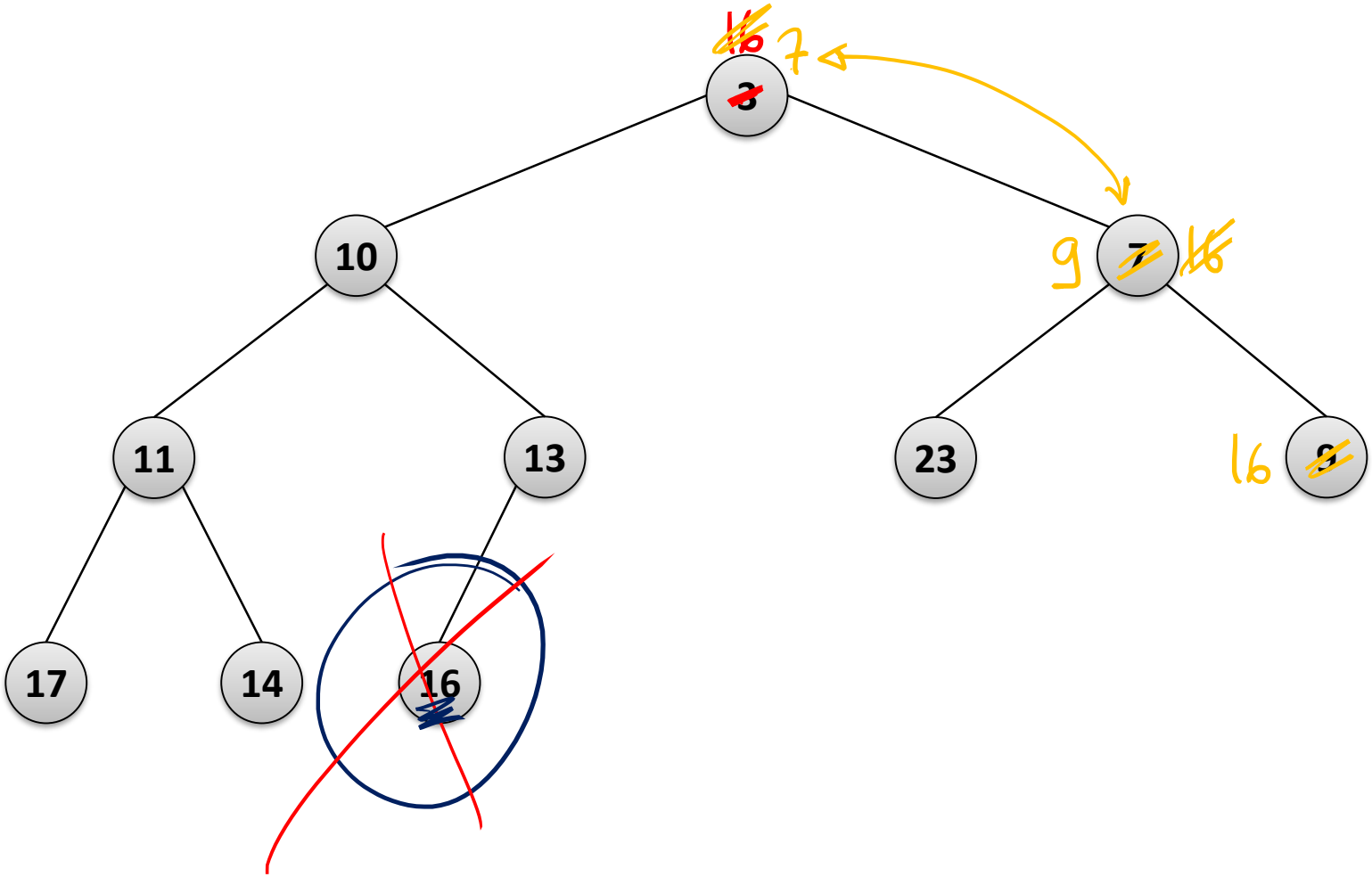
- Die Datenstruktur heisst deshalb oft auch Heap
- Ein Baum hat die Min-Heap Eigenschaft, falls in jedem Teilbaum, die Wurzel den kleinsten Schlüssel hat
- getMin-Operation: trivial!
- Baum wird immer so balanciert, wie möglich gehalten



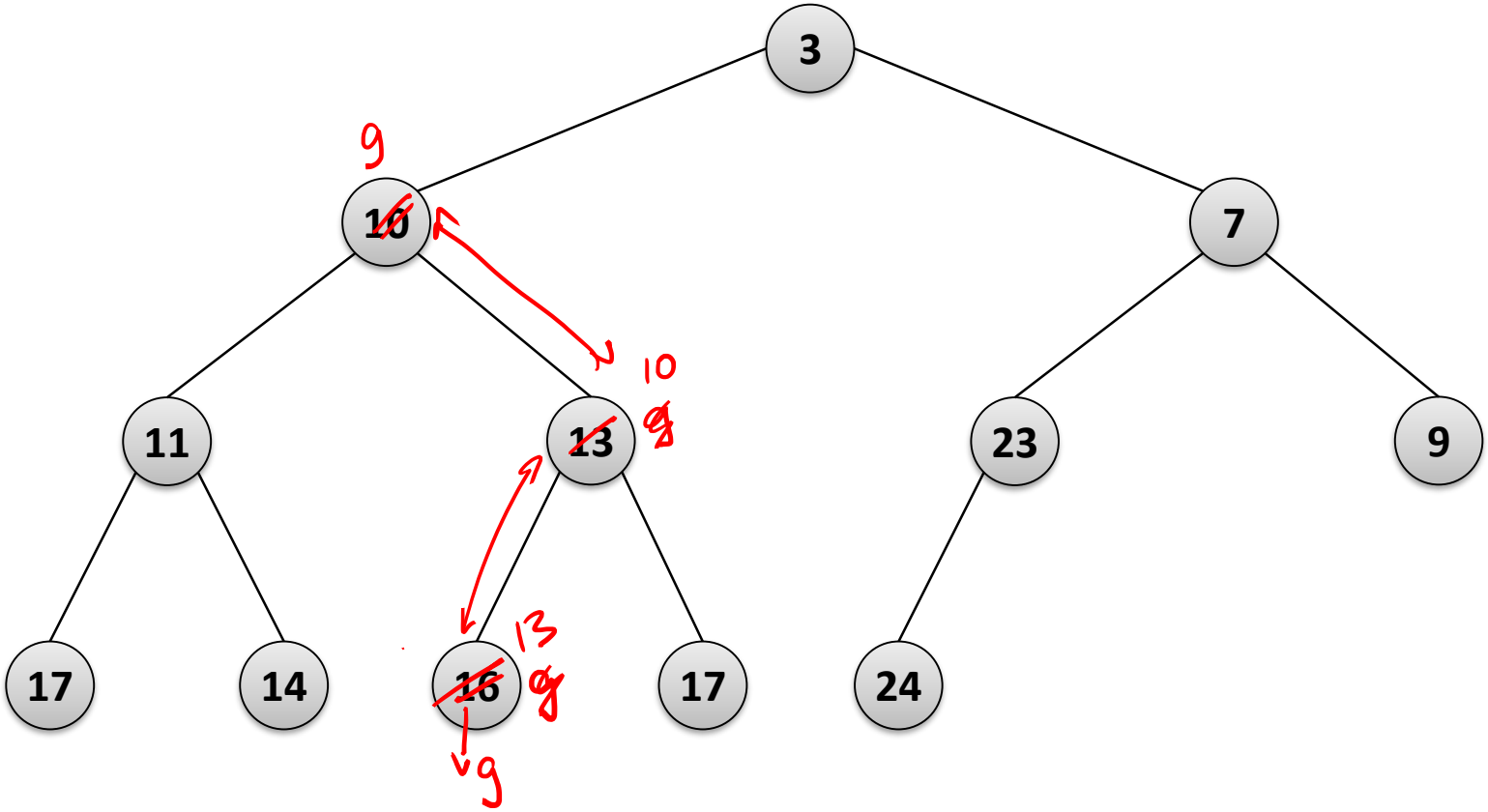
Prioritätswarteschlangen: Einfügen



Prioritätswarteschlangen: Delete-Min

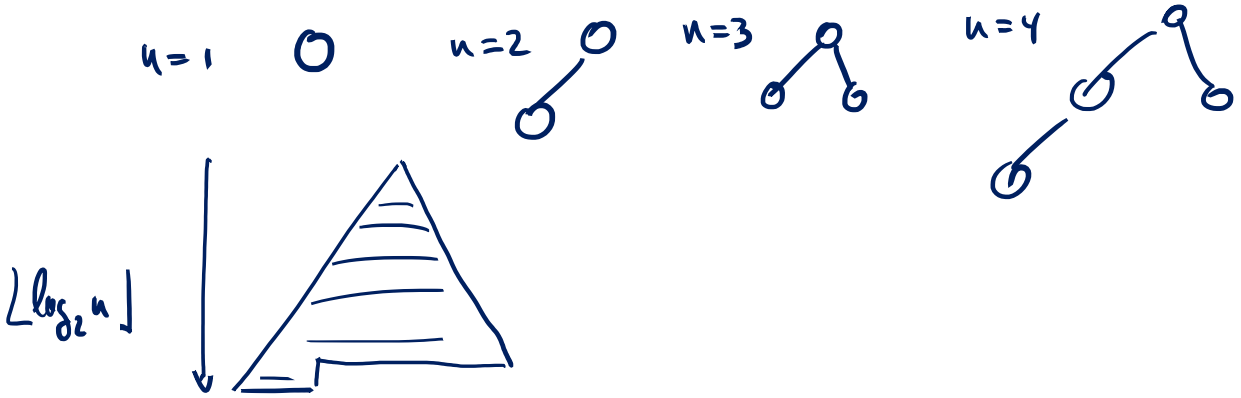


Prioritätswarteschlangen: Decrease-Key



Prioritätswarteschlangen: Analyse

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

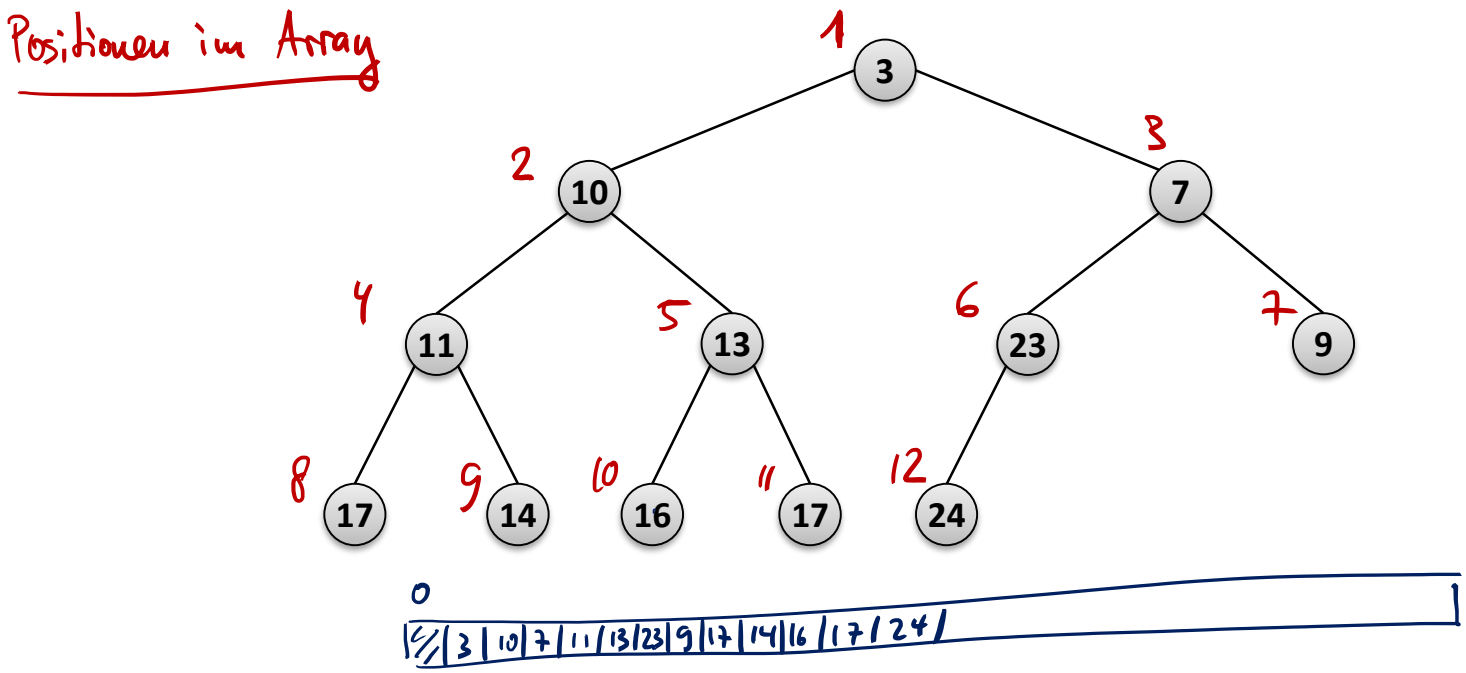


- **Laufzeit aller Operationen: $O(\log n)$**
  - wenn man den Binärbaum irgendwie vernünftig implementiert
  - man muss immer höchsten einmal den Binärbaum hoch (bei insert, decreaseKey) oder runter (bei deleteMin)
  - Wir werden gleich eine elegante Art sehen, den binären Heap zu impl.

Binäre Heaps, Array-Implementierung

Idee: Speichere **alles in ein Array**

- Das geht, weil der Baum perfekt balanciert ist



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: $2i + 1$

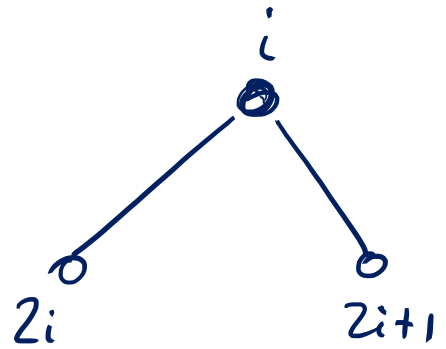
Pseudocode am Beispiel von Insert

- Elemente sind in Array A and 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 <= 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):
  while (i > 1) and (A[i] < A[i/2]) do
     swap(A[i], A[i/2])
     i = i/2
```

* if A[i] > x then
  A[i] = x

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

$O(\log n)$

$O(\log n)$

Laufzeit $O(n \log n)$

- **Laufzeit:** $O(n \log n)$ - Sortieren $\Omega(n \log n)$ untere Schranke

Prims Algorithmus mit binären Heaps

H = new BinaryHeap(); $A = \emptyset$
for all $u \in V \setminus \{s\}$ **do**
    H.insert(u, ∞); $\alpha(u)$ = NULL
H.insert(s, 0)
while H is not empty **do**
    u = H.deleteMin()
    for all 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** $A = A \cup \{\{u, \alpha(u)\}\}$

Laufzeit: $O(m \cdot \log n)$

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

Prims Algorithmus: Bessere Laufzeit

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

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

$a(n_i + n_k + n_d \log n)$

Beste Implementierung von Prioritätswarteschlangen:

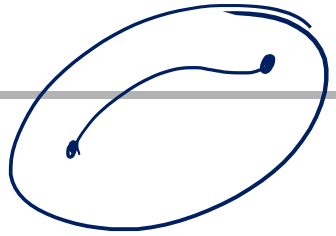
- Fibonacci Heaps (siehe Vorlesung Algorithmentheorie)
- Laufzeit der Operationen (deleteMin, decreaseKey amortisiert)

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

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

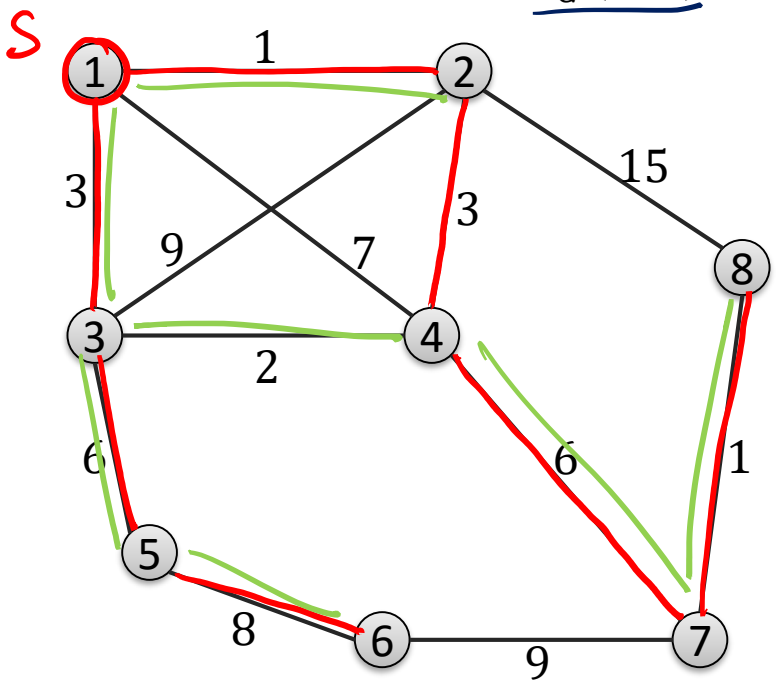
- $n \leq m + 1$ insert-Operationen und deleteMin-Operationen
- $\leq m$ decreaseKey-Operationen

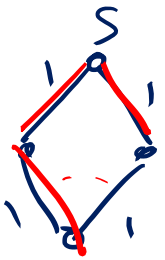
Kürzeste Wege



Problem

- Gegeben: gewichteter Graph $G = (V, E, w)$, Startknoten $s \in V$
  - Wir bezeichnen Gewicht einer Kante (u, v) als $w(u, v)$
  - Annahme: $\forall e \in E: w(e) \geq 0$
- Ziel: Finde kürzeste Pfade / Distanzen von s zu allen Knoten
  - Distanz von s zu v: $d_G(s, v)$ (Länge eines kürzesten Pfades)





shortest path tree

Wurzel s

$\forall v$: Pfad von s zu v im Baum ist ein kürzester Pfad von s zu v in G.