Informatik II - SS 2014

(Algorithmen & Datenstrukturen)

Vorlesung 17 (8.7.2014)

Minimale Spannbäume II
Union-Find, Prioritätswarteschlangen

UNI FREIBURG

Fabian Kuhn

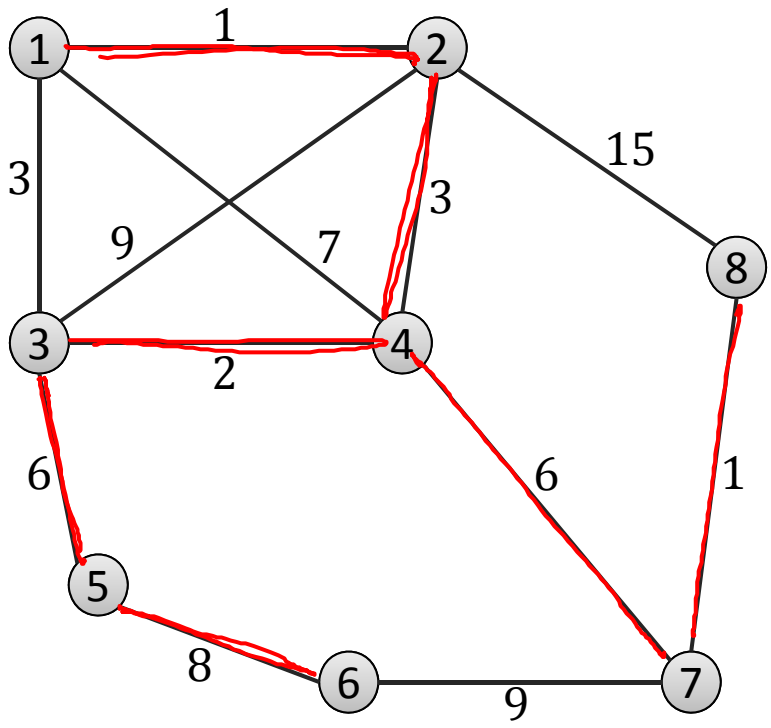
Algorithmen und Komplexität

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

UNI FREIBURG

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

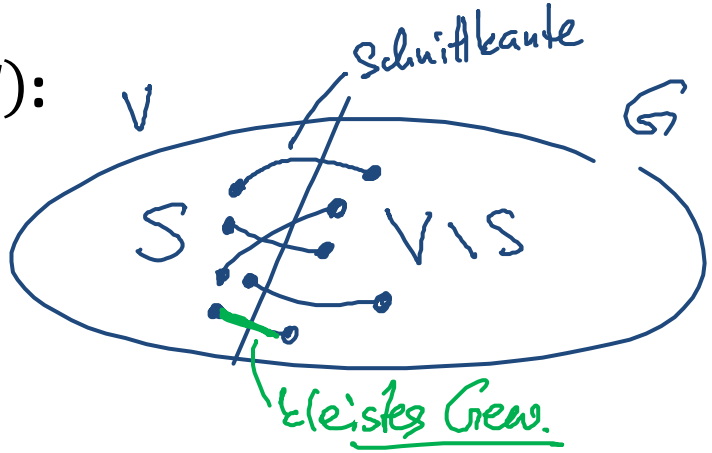
- Invariante ist eine gültige Schleifeninvariante
- Invariante + Abbruchbedingung $\Longrightarrow$ A ist ein MST!

Wie findet man sichere Kanten?

- Invariante → es gibt immer mindestens eine sicher Kante
  - A ist Teilmenge eines MST und kann daher zu einem MST erweitert werden

- Zuerst benötigen wir ein paar Begriffe…

Schnitt $(S, V \setminus S)$:



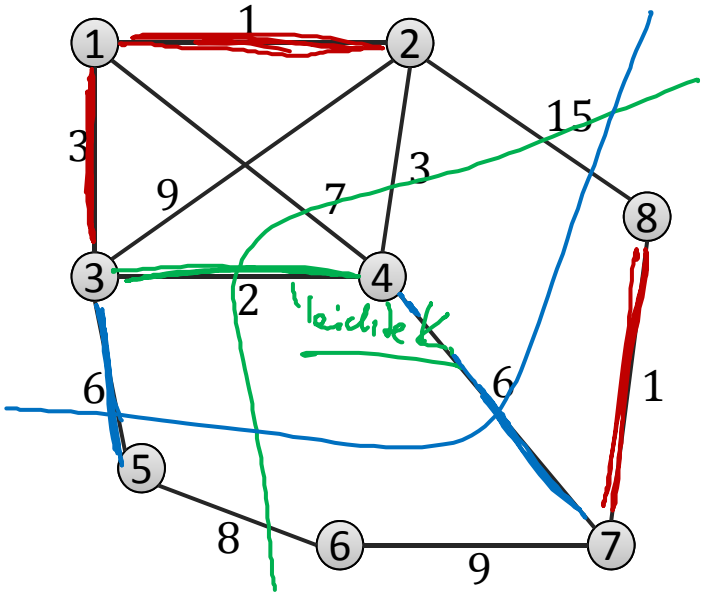
- Kante $\{u, v\} \in E$ ist eine Schnittkante bezüglich $(S, V \setminus S)$, falls ein Ende in S und ein Ende in $V \setminus S$ ist.
- Wir nennen Kante $\{u, v\}$ eine **leichte Schnittkante** bez. $(S, V \setminus S)$, falls sie das kleinste Gewicht von allen Schnittkanten hat

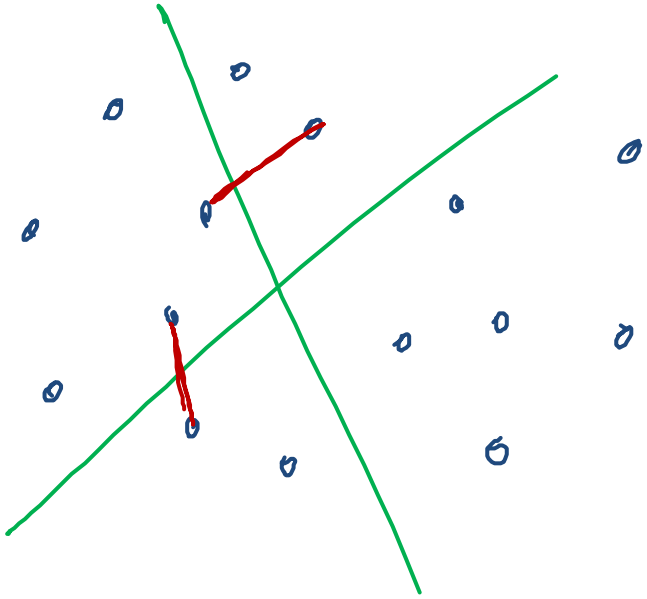
Sichere Kanten

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.



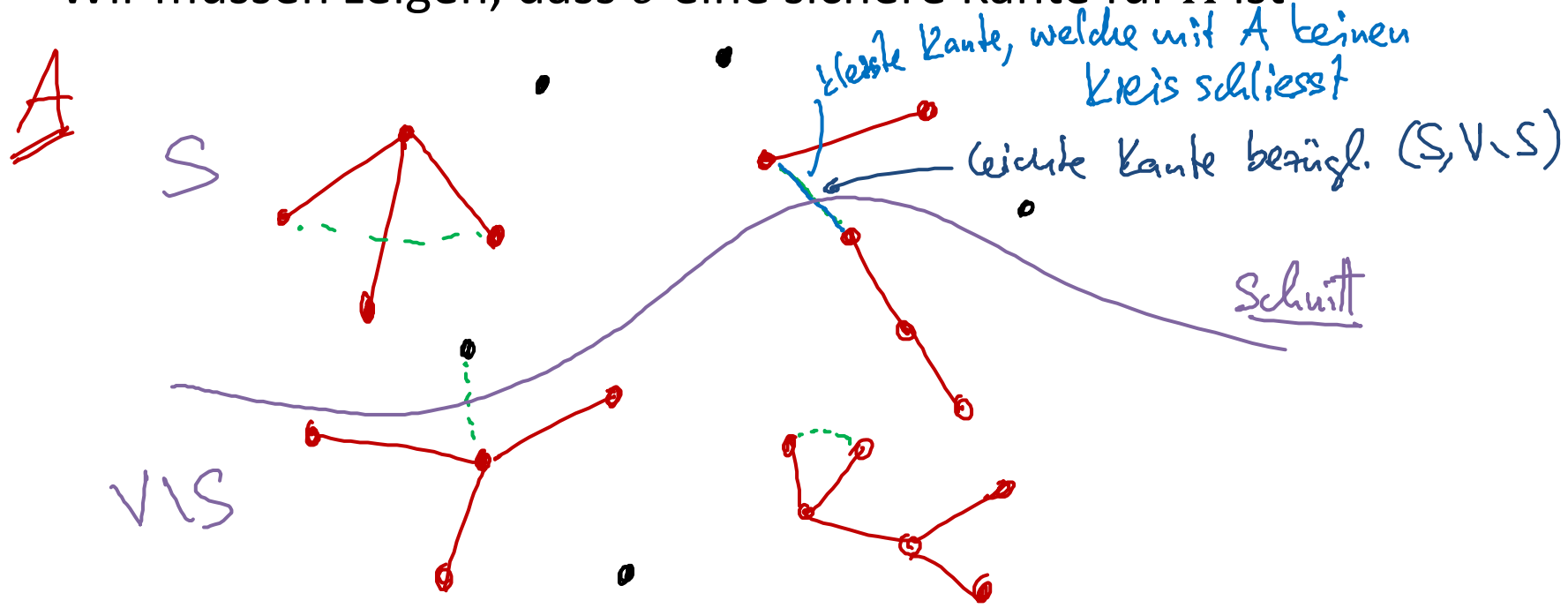


Kruskal's MST-Algorithmus

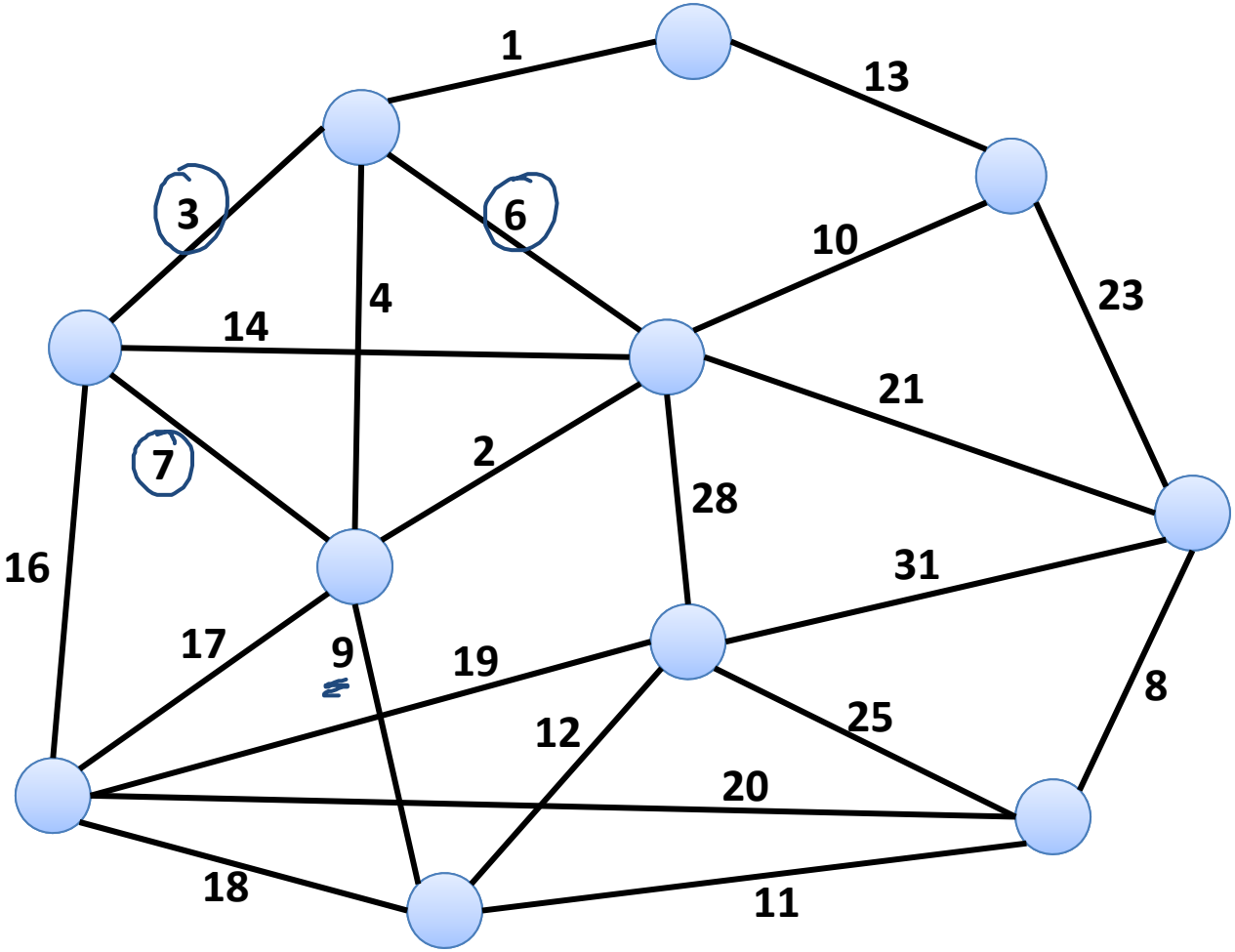
$A = \emptyset$
while A ist kein Spannbaum **do**
    $e = \{u, v\}$ ist Kante mit kleinstem Gewicht,
    so dass $A \cup \{\{u, v\}\}$ keinen Zyklus enthält
    $A = A \cup \{\{u, v\}\}$

= Kreis

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



Kruskal's MST-Algorithmus: Beispiel



Kruskal's MST-Algorithmus: Bemerkungen

UNI FREIBURG

Kruskal's Algorithmus berechnet minimalen Spannbaum

- Die hinzugefügte Kante ist in jedem Schritt sicher
- Wir haben gesehen, dass der Basis-Algorithmus korrekt ist

Kruskal's Algorithmus ist ein typisches Beispiel eines sogenannten

Greedy Algorithmus

- Wir beginnen mit einer leeren Kantenmenge
- In jedem Schritt wird die im Moment beste Kante hinzugenommen
- Eine gewählte Kante wird nie wieder verworfen

- Wir werden noch kurz besprechen, wie man den Algorithmus effizient implementieren kann...

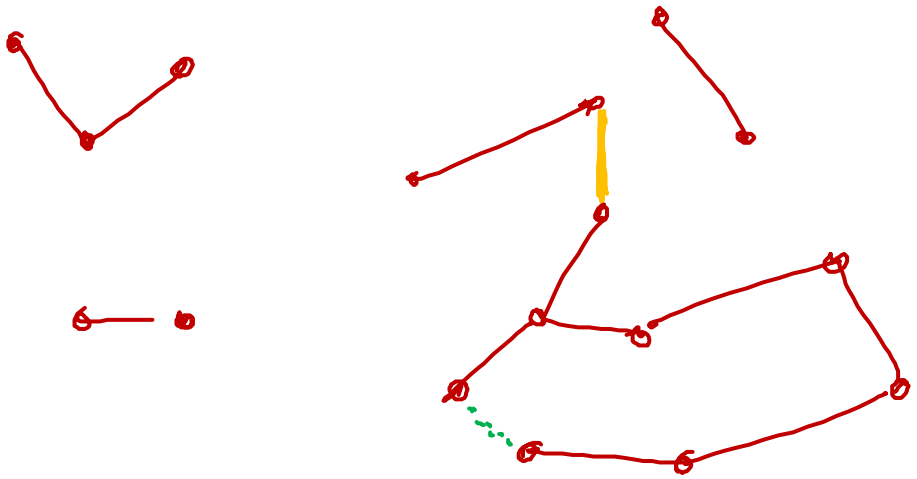
Implementierung Kruskal's Algorithmus

UNI FREIBURG

Kruskal's Algorithmus (Pseudo-Code)

1. $A = \emptyset$
2. Sortiere Kanten aufsteigend nach Kantengewicht
3. **for** $e = \{u, v\} \in E$ (in sorted order) **do**
4. **if** u and v are in different components **then**
5. $A = A \cup \{e\}$

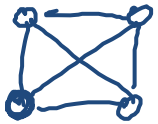
des durch A best. Teilgraphen



Implementierung Kruskal's Algorithmus

Kruskal's Algorithmus (Pseudo-Code)

1. $A = \emptyset$
2. Sortiere Kanten aufsteigend nach Kantengewicht
3. **for** $e = \{u, v\} \in E$ (in sorted order) **do**
4. **if** u and v are in different components **then**
5. $A = A \cup \{e\}$



$\binom{n}{2} = \frac{n(n-1)}{2}$

- Müssen Komponenten des durch A bestimmten Graphen effizient verwalten können

$n-1 \leq m \leq \frac{n^2}{2}$ $\quad \log(n-1) \leq \log m \leq 2 \cdot \log(n)$

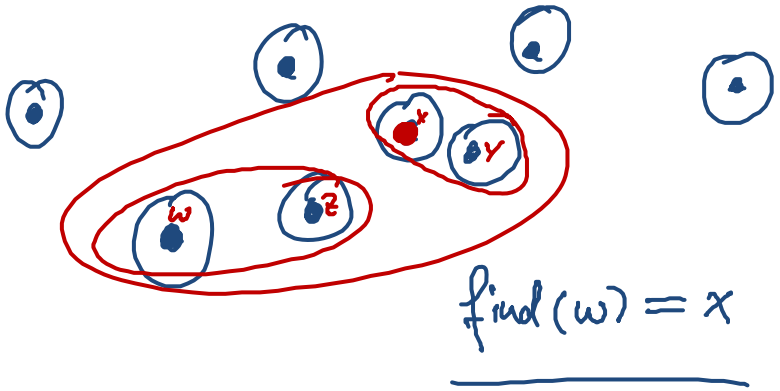
- **Laufzeit:** $O(m \log n)$ für's Sortieren, sowie die Gesamtzeit, um die Komponenten zu verwalten…

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

UNI FREIBURG

Kruskal's Algorithmus

1. $A = \emptyset$
2. U = create new Union Find DS
3. **for all** $u \in V$ **do**
4. U.makeSet(u) (n x makeSet)
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) 2m find Op.
8. **if** $S_u \neq S_v$ **then**
9. $A = A \cup \{e\}$
10. U.union(S_u, S_v) n-1 union Op.

Kruskal: Laufzeitanalyse

UNI FREIBURG

Beste Union-Find Datenstruktur

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

in der Praxis ist $\alpha(m,n)$ eine Konstante

$$O(m \cdot \alpha(m,n))$$

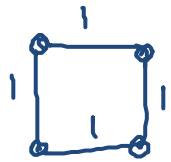
- $\alpha(m,n)$ ist die Inverse der Ackermannfunktion und wächst extrem langsam (für alle halbwegs vernünftigen m, n, $\alpha(m,n) \leq 5$)

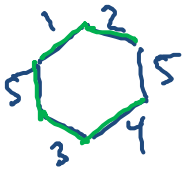
Laufzeit Kruskal

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

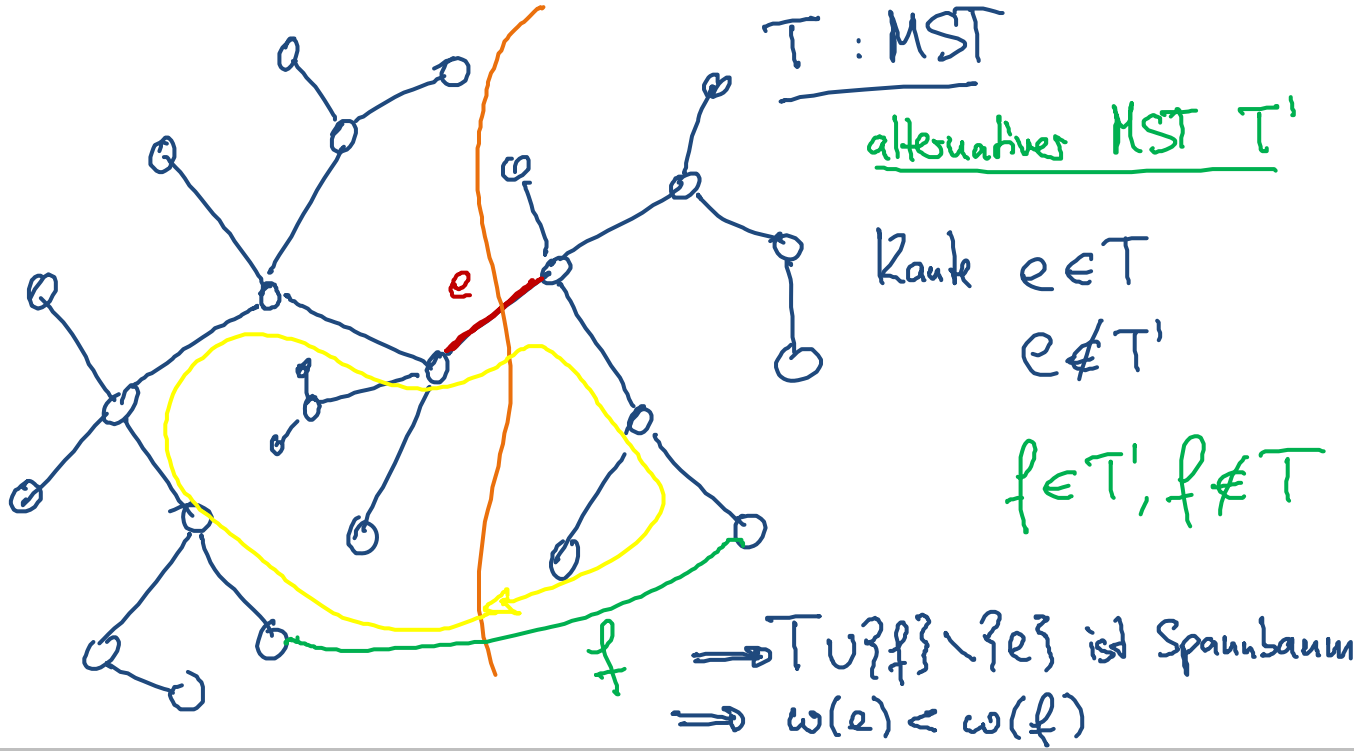
MST Eindeutigkeit

- Im Allgemeinen ist der MST nicht eindeutig





Satz: Bei paarweise verschiedenen Kantengewichten ist der MST eindeutig.



Prim's MST Algorithmus

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

- Eine zweite Implementierung des Basis-Algorithmus

```
A = ∅
while A ist kein Spannbaum do
    Finde sichere Kante {u,v} für A
    A = A ∪ {{u,v}}
return A
```

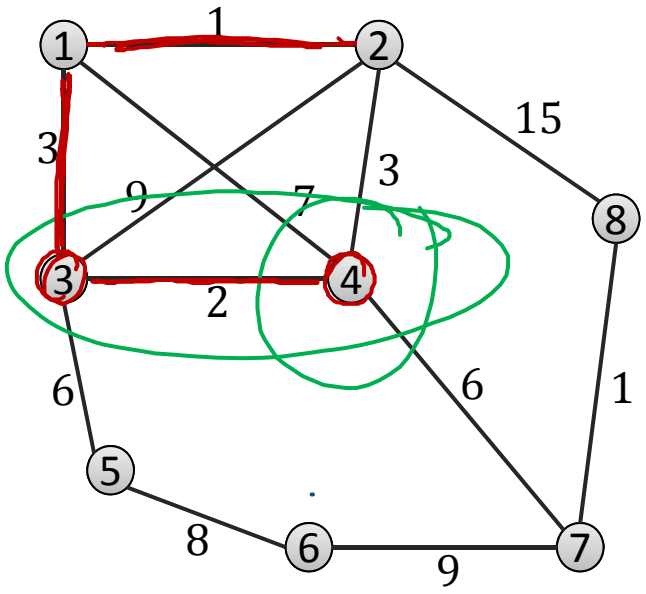
- **Idee:** A ist immer ein zusammenhängender Teilbaum
  - Starte bei einem beliebigen Knoten $s \in V$
  - Baum wächst von s aus, indem immer eine leichte Schnittkante des durch A induzierten Schnitts hinzugefügt wird.

Sichere Kanten

Annahmen:

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

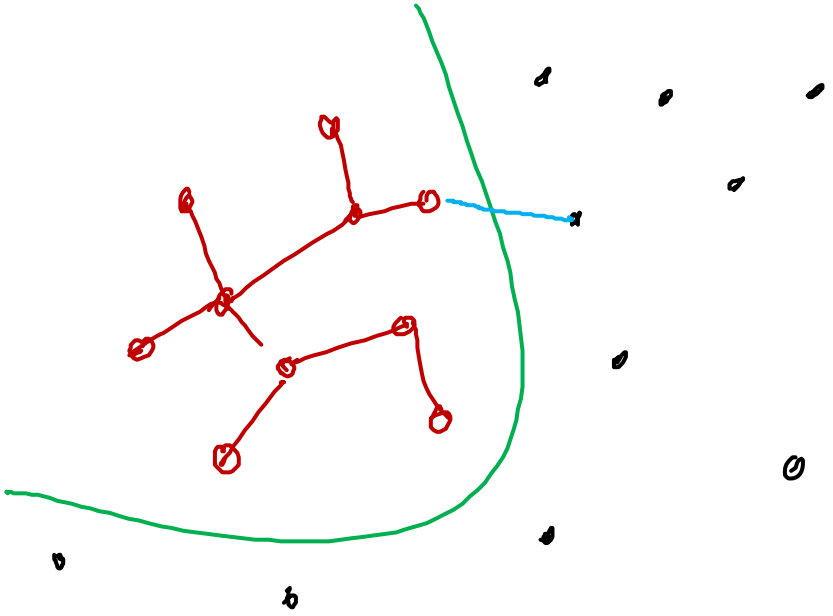
Theorem: Sei $(S, V \setminus S)$ ein Schnitt, so dass A keine Schnittkanten enthält und sei $\{u, v\}$, $u \in S$, $v \in V \setminus S$ eine leichte Schnittkante bezüglich $(S, V \setminus S)$. Dann ist $\{u, v\}$ eine sichere Kante für A.



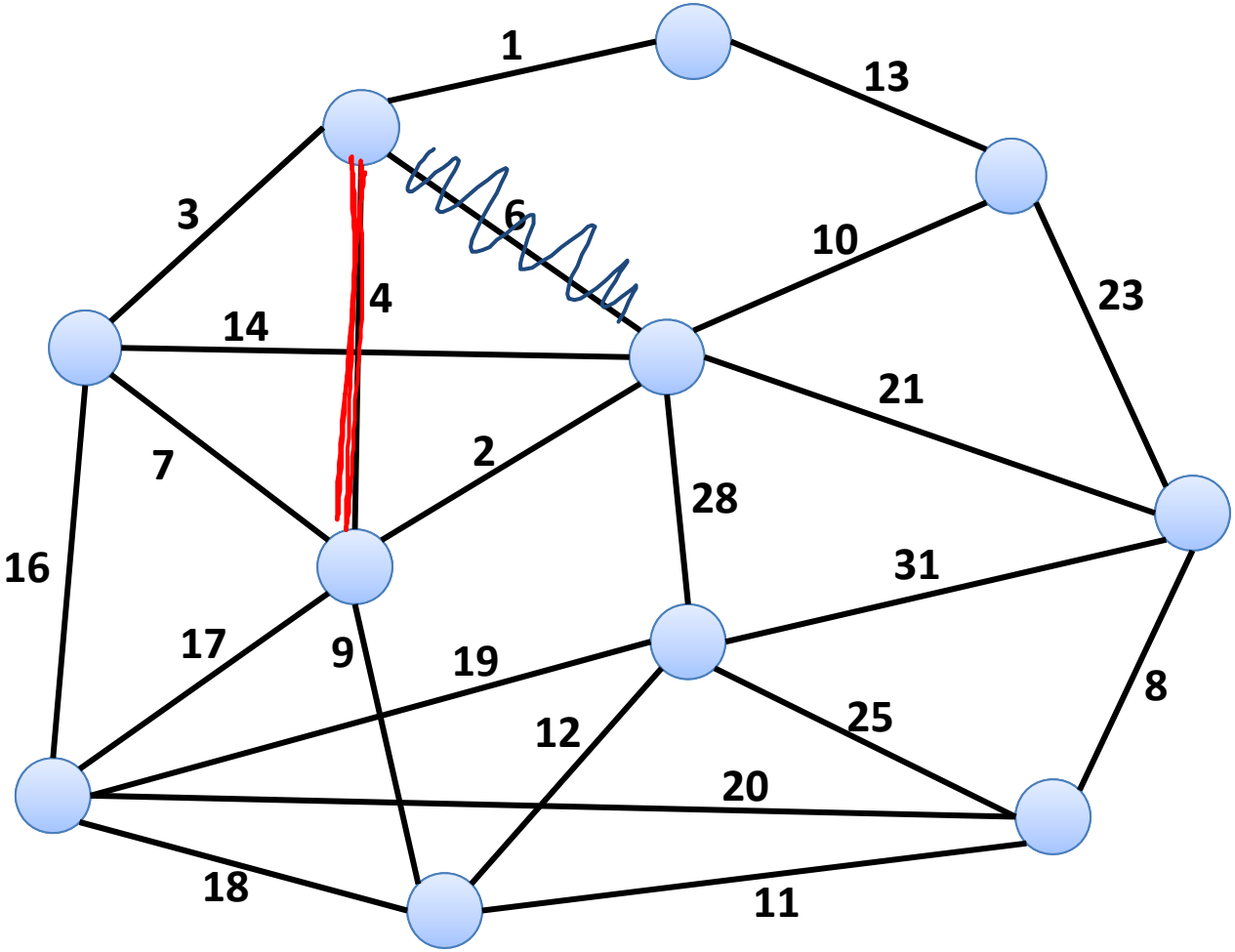
Prim's MST-Algorithmus

$A = \emptyset$
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

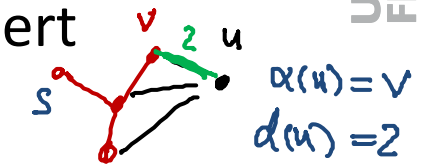


Prim's MST-Algorithmus: Beispiel

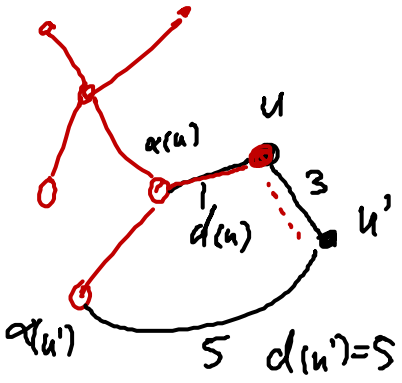


Implementierung von Prim's Algorithmus

- 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 $\alpha(u) = \text{NULL}$)




```
for all u ∈ V \ {s} do
    u.marked = false; d(u) = ∞; α(u) = NULL
d(s) = 0; A = ∅                          // 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
            α(v) = u; d(v) = w({u,v})
    u.marked = true
    if u ≠ s then A = A ∪ {{u, α(u)}}
```



Prioritätswarteschlange

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 Prim's Algorithmus

benutze priority queue, um die unmarkierten Knoten zu verwalten

```
for all u ∈ V \ {s} do
    u.marked = false; d(u) = ∞; α(u) = NULL
d(s) = 0; A = ∅                    // 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
            α(v) = u; d(v) = w({u,v})
    u.marked = true
    if u ≠ s then A = A ∪ {{u, α(u)}}
```

H = leere pr. queue
füge all Knoten ein mit Schlüssel d(u)

H ist nicht leer

deleteMin

decreaseKey

Implementierung von Prim's Algorithmus

UNI FREIBURG

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

$n \times$ H.insert

while H is not empty **do**
    u = H.deleteMin()

$n \times$ deleteMin

    for all neighbors v of u **do**

// only unmarked neighbors

        if $w(\{u,v\}) < d(v)$ **then**
            H.decreaseKey(v, $w(\{u,v\})$)
            $\alpha(v) = u$

decreaseKey's $\leq m$

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

Analyse von Prim's MST-Algorithmus

UNI FREIBURG

Anzahl Priority Queue Operationen

- **create** 1
- **insert** n
- **getMin / deleteMin** n
- **decreaseKey** $\leq m$

Gesamt-Laufzeit

$$O\Big(\underbrace{n \cdot (\text{Zeit f. insert/delMin})}_{O(\log n)} + \underbrace{m \cdot (\text{Zeit f. decreaseKey})}_{O(\log n)}\Big)$$

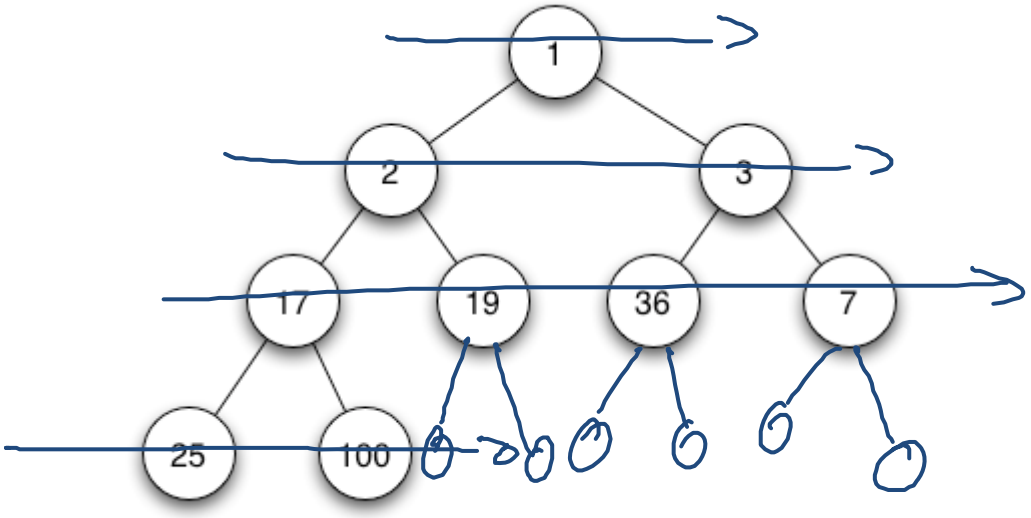
$$\in O(m \cdot \log n)$$

Prioritätswarteschlangen

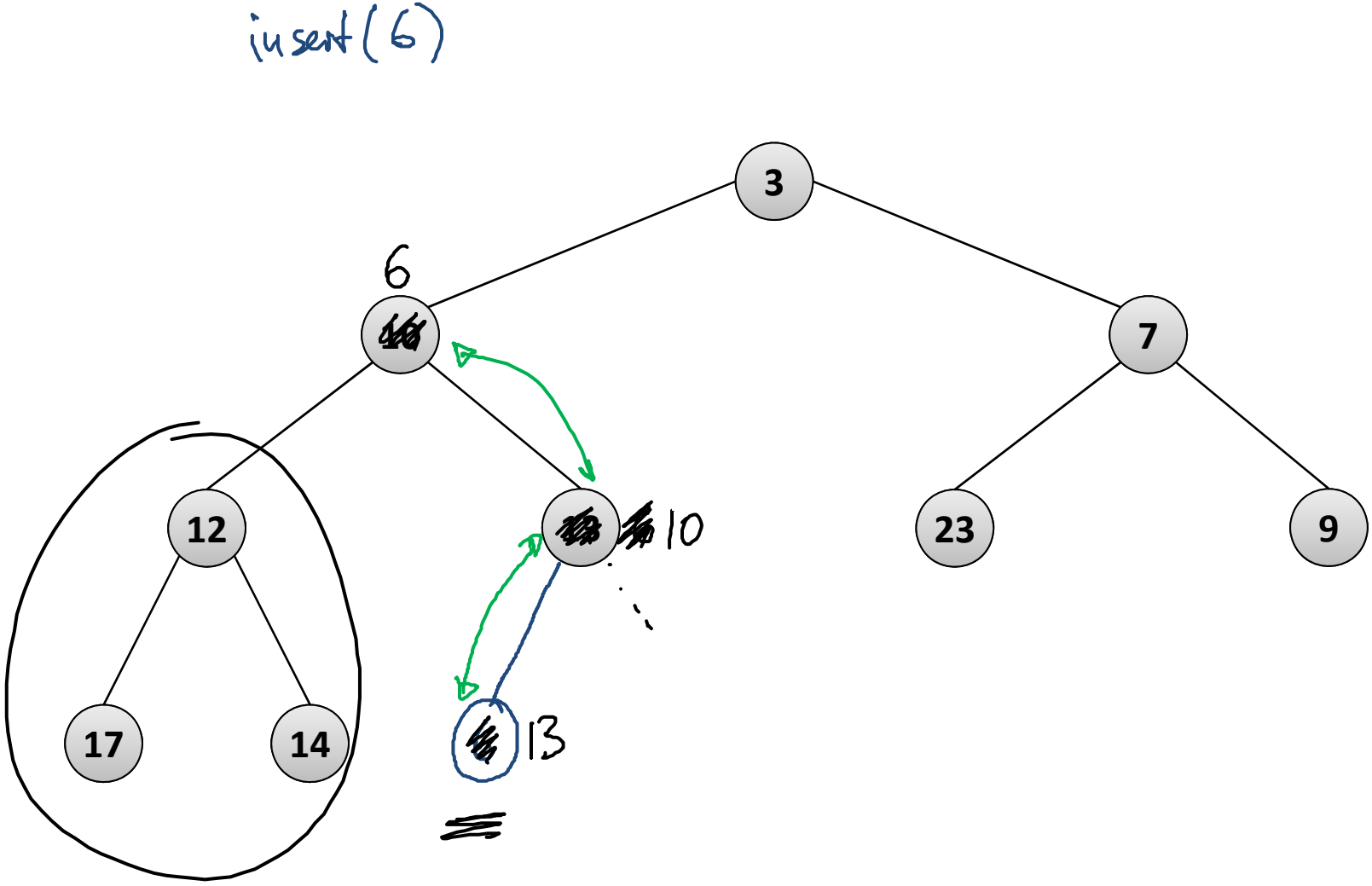
UNI FREIBURG

Implementierung als Binärbaum mit Min-Heap Eigenschaft

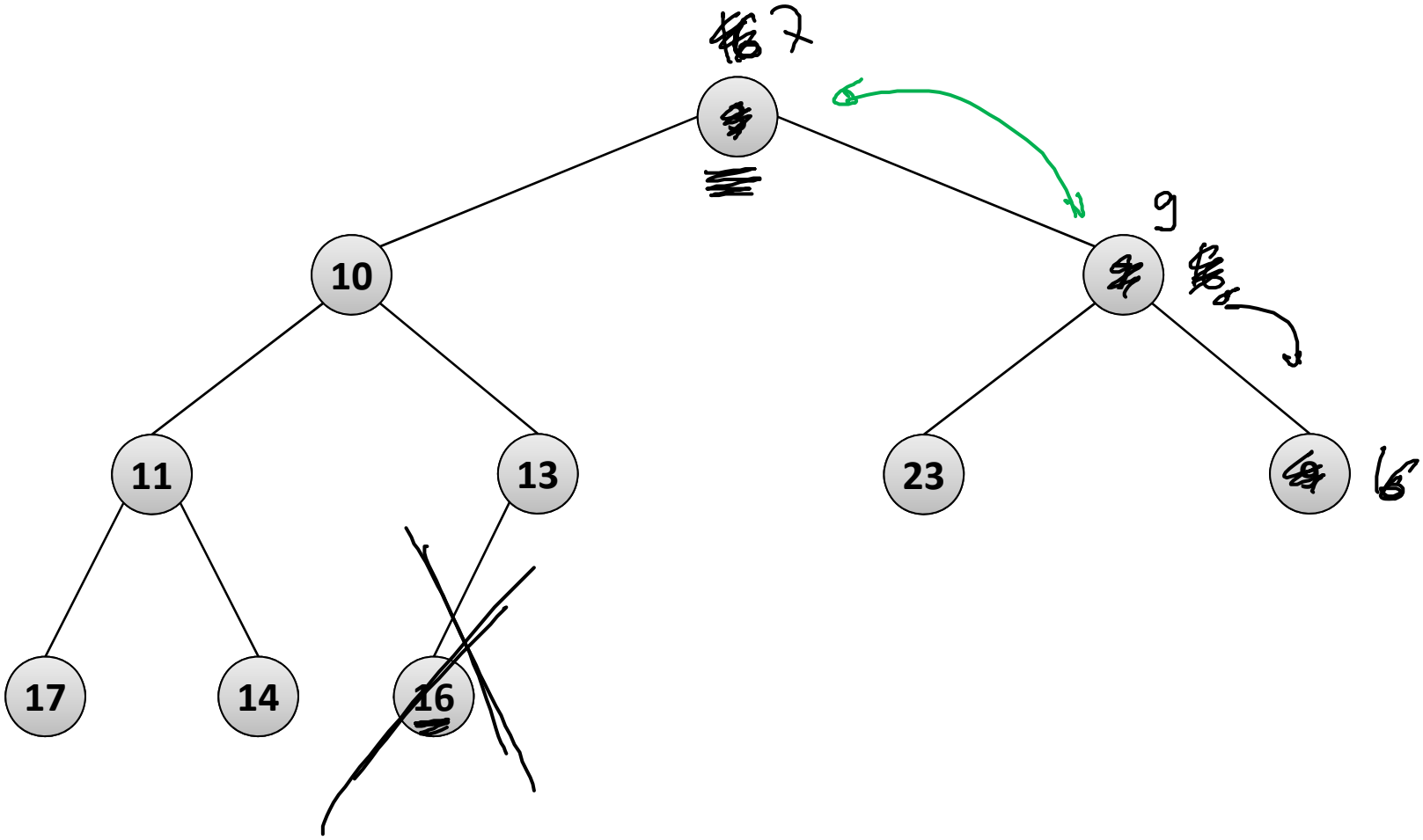
- Die Datenstruktur heisst deshalb oft auch Heap
- Ein Baum hat die Min-Heap Eigenschaft, falls in jedem Teilbaum, die Wurzel den kleinsten Schlüssel hat
- getMin-Operation: trivial!
- Baum wird immer so balanciert, wie möglich gehalten



Prioritätswarteschlangen: Einfügen



Prioritätswarteschlangen: Delete-Min



Prioritätswarteschlangen: Decrease-Key