

# Algorithmen und Datenstrukturen

## Vorlesung 9

### Graphenalgorithmen II: Minimale Spannbäume



**UNI  
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

# Graphen

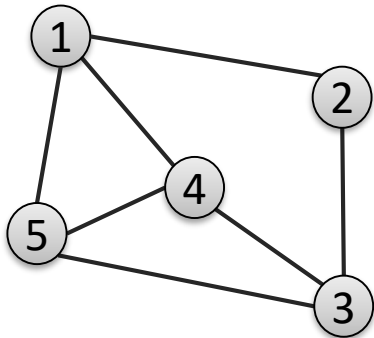
**Knotenmenge  $V$** , typischerweise  $n := |V|$  (English: **vertex** or **node**)

**Kantenmenge  $E$** , typischerweise  $m := |E|$  (English: **edge**)

- ungerichteter Graph:  $E \subseteq \{\{u, v\} : u, v \in V\}$

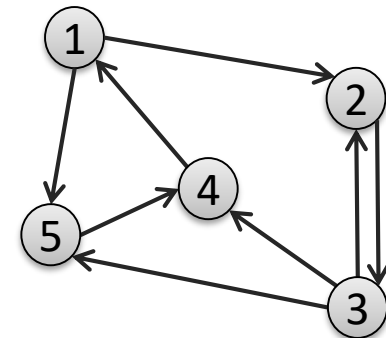
In dieser Vorlesung: Nur ungerichtete Graphen

## Beispiele:



$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1,2), (1,5), (2,3), (3,4), (3,4), (3,5), (4,1), (5,4)\}$$



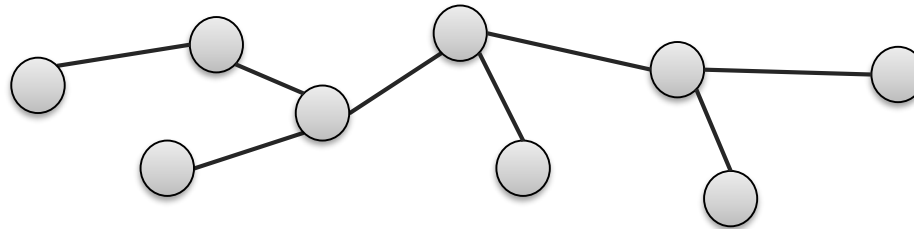
$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{\{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{3,4\}, \{3,5\}, \{4,5\}\}$$

- Als ungerichtete Graphen (mit  $n$  Knoten) betrachtet...

## Baum:

- Zusammenhängender ungerichteter Graph, ohne Zyklen
  - Ein nicht zusammenhängender zyklensfreier (unger.) Graph heisst Wald
  - Anzahl Kanten:  $n - 1$  (jede Kante reduziert die #Komponenten um 1)



## Äquivalente Definitionen:

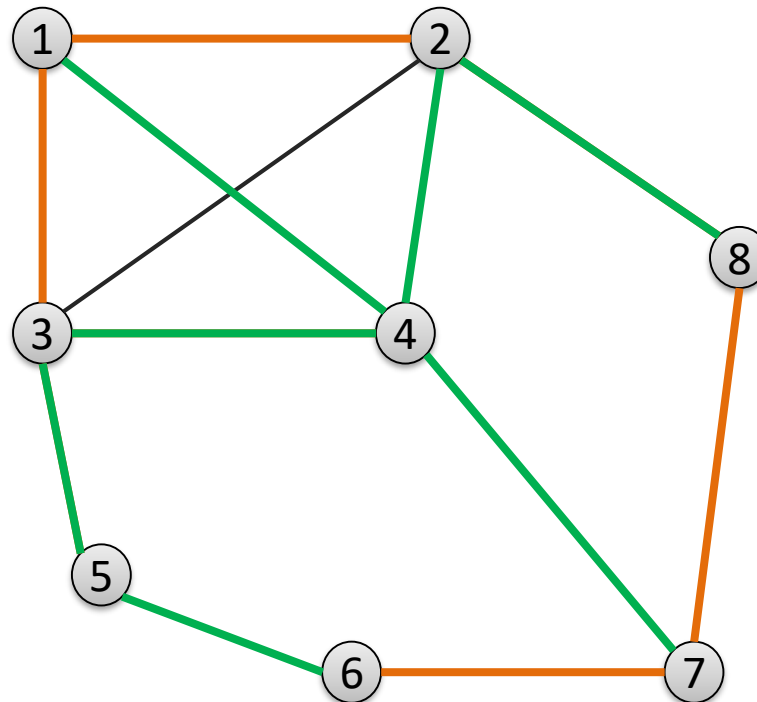
- Minimaler zusammenhängender Graph
- Maximaler zyklensfreier Graph
- Eindeutiger Pfad zwischen jedem Knotenpaar
- Zusammenhängender Graph mit  $n - 1$  Kanten

# Spannbaum

**Gegeben:** Zusammenhängender, ungerichteter Graph  $G = (V, E)$

**Spannbaum  $T = (V, E_T)$ :** Teilgraph ( $E_T \subseteq E$ )

- $T$  ist ein Baum, welcher alle Knoten von  $G$  enthält
- Alternativ:  $T$  ist ein Baum mit  $n - 1$  Kanten aus  $E$



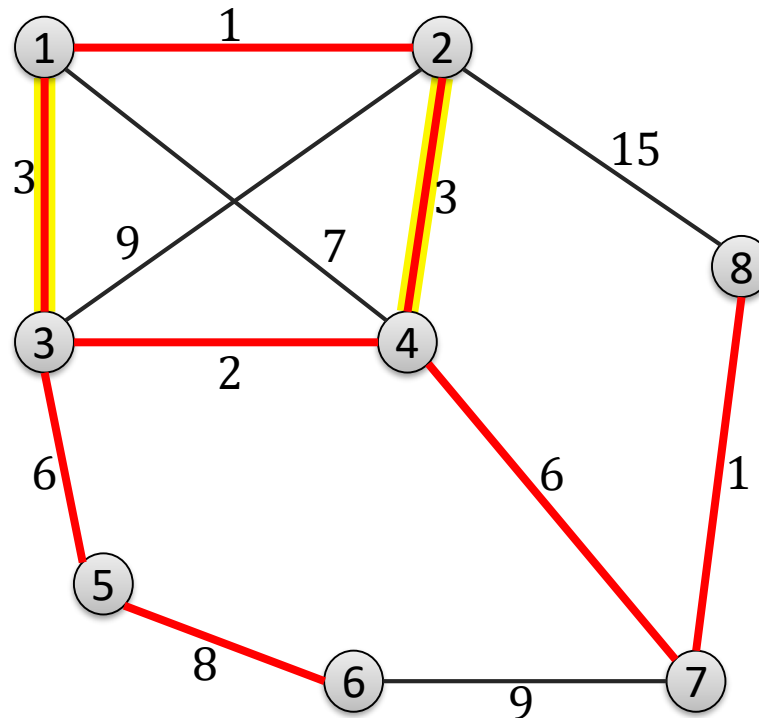
# Minimaler Spannbaum

**Gegeben:** Zusammenhängender, ungerichteter

Graph  $G = (V, E, w)$  mit Kantengewichten  $w : E \rightarrow \mathbb{R}$

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

- Spannbaum mit kleinstem Gesamtgewicht



# Minimale Spann­b­ume

**Ziel:** Gegeben ein gewichteter, ungerichteter Graph  $G$ , finde einen Spannbaum mit minimalem Gesamtgewicht.

- **Minimaler Spannbaum = Minimum Spanning Tree = MST**
- Ein grundlegendes Optimierungsproblem auf Graphen
  - eines von sehr vielen Optimierungsproblemen auf Graphen
- kommt oft als Teilproblem vor
- ist aber auch interessant an sich
  - Zum Beispiel im Zusammenhang mit Netzwerken
  - Ein minimaler Spannbaum ist die kostengünstigste Möglichkeit alle Knoten in einem Netzwerk zu verbinden.

**Idee:** Starte mit leerer Kantenmenge und füge die Kanten schrittweise hinzu, bis es ein Spannbaum ist

## Invariante:

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

- Am Anfang ist  $A = \emptyset$
- Danach wird jeweils eine Kante hinzugefügt, ohne die Invariante zu verletzen
- Wir nennen eine Kante, für welche wir sicher sein können, dass wir sie zu  $A$  hinzufügen können eine **sichere Kante für  $A$**
- Wie man sichere Kanten findet, werden wir sehen...

## 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**  $A$

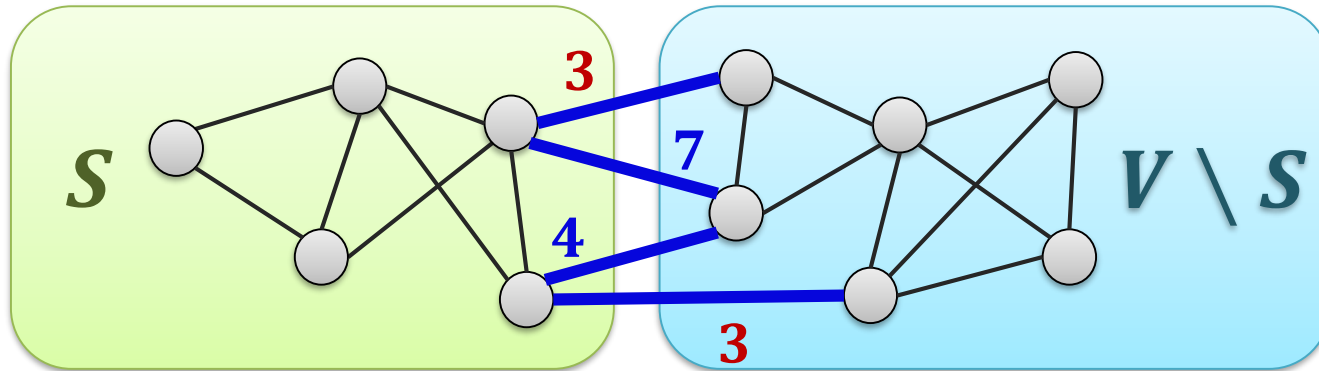
- Invariante ist eine gültige Schleifeninvariante
- **Invariante + Abbruchbedingung  $\Rightarrow A$  ist ein MST!**



# Wie findet man sichere Kanten?

- Invariante  $\rightarrow$  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  $(S, V \setminus S)$ ,  $S \neq \emptyset$ ,  $S \neq V$ :**

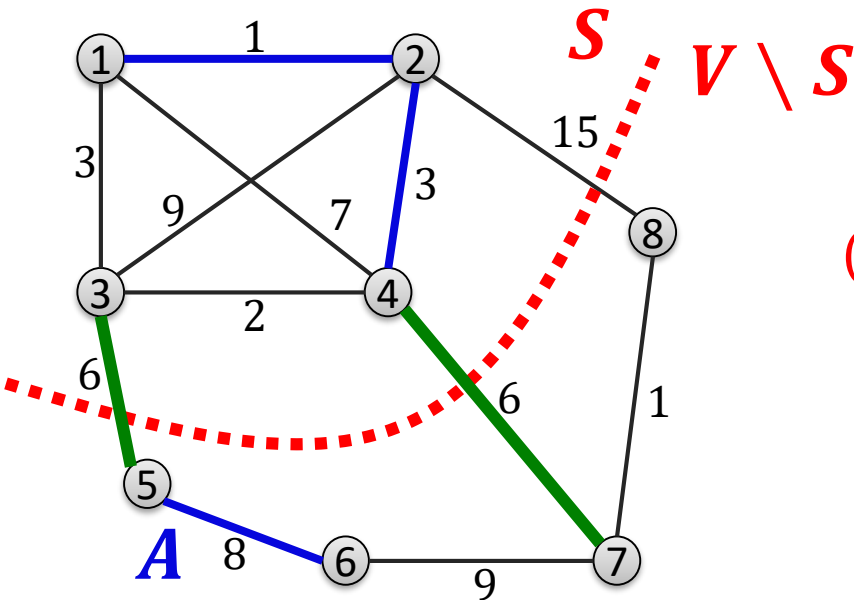


- Kante  $\{u, v\} \in E$  ist eine **Schnittkante** bezüglich  $(S, V \setminus S)$ , falls ein Knoten der Kante in  $S$  und ein Knoten der Kante in  $V \setminus S$  ist.
- Wir nennen Kante  $\{u, v\}$  eine **leichte Schnittkante** bez.  $(S, V \setminus S)$ , falls sie das **kleinste Gewicht** von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

## 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  $A$ .



**A:** Kantenmenge, die Teilmenge eines MST ist.

**$(S, V \setminus S)$ :** Schnitt, bei dem keine Kante in  $A$  Schnittkante ist.

Leichte Schnittkanten sind sichere Kanten und können zu  $A$  hinzugefügt werden.

# Kurzer Exkurs zu Bäumen

**Theorem:** Ein zusammenhängender (ungerichteter) Graph  $G = (V, E)$  mit  $n$  Knoten und  $n - 1$  Kanten ist ein Baum.

**Beweis:** Per Induktion über  $n$

• **Verankerung** ( $n = 1$ ): 

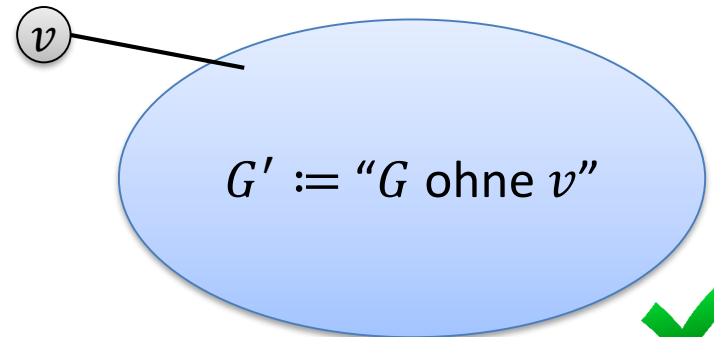


• **Induktionsschritt** ( $n - 1 \rightarrow n$ ):

– Ein Graph mit  $n$  Knoten und  $n - 1$  Kanten hat einen Knoten mit  $\text{Grad} \leq 1$

$$\text{avgdeg}(G) = \frac{1}{n} \cdot \sum_{v \in V} \text{deg}(v) = \frac{2|E|}{n} = \frac{2n - 2}{n} < 2$$

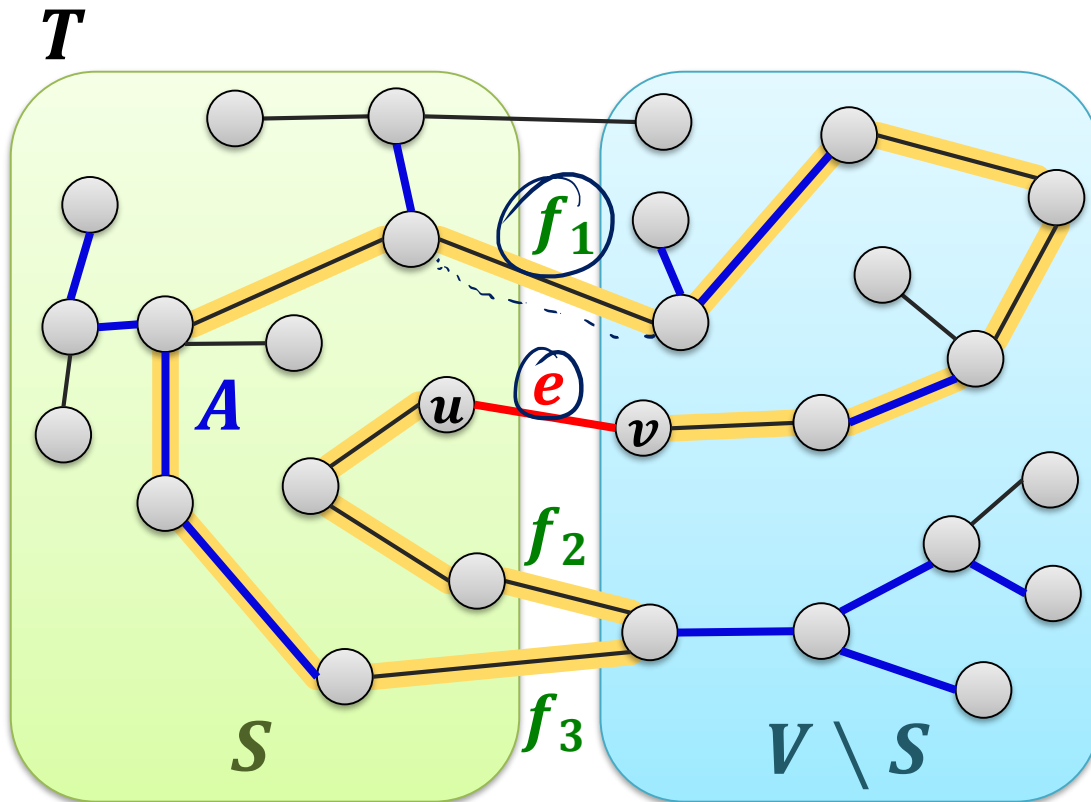
– Falls  $G$  zusammenhängend ist :  $\exists v \in V : \text{deg}(v) = 1$



# Sichere Kanten

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

**Beweis:** Betrachte einen MST  $T$ , der die Kanten in  $A$  enthält.



## Schnittkante $f_i$ auf $u$ - $v$ Pfad

- $T' := (V, E \setminus \{f_i\} \cup \{e\})$   
 $\Rightarrow T'$  ist zus.-hängend.
- $T'$  hat  $n - 1$  Kanten  
 $\Rightarrow T'$  ist ein Baum.
- $e$  leichte Kante  
 $\Rightarrow \underline{w(e) \leq w(f_i)}$
- $\underline{w(T')} \leq \underline{w(T)}$   
 $\Rightarrow T'$  ist ein MST, der  $A$  und  $e$  enthält.

# Prims MST Algorithmus

- Sollte man wohl eigentlich Jarníks Algorithmus nennen
  - wurde 1957 von Prim und bereits 1930 von Jarník publiziert

- Eine mögliche Implementierung des Basis-Algorithmus

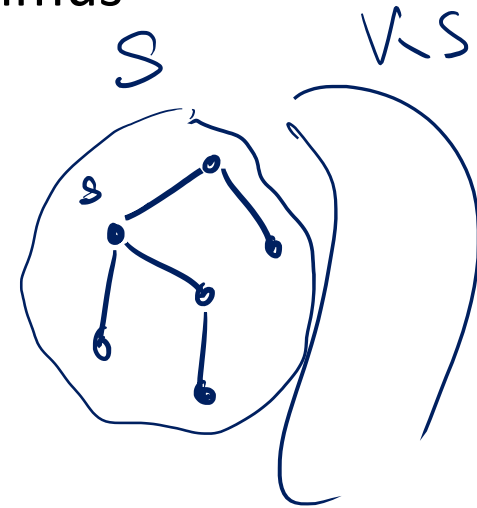
$A = \emptyset$

**while**  $A$  ist kein Spannbaum **do**

    Finde sichere Kante  $\{u, v\}$  für  $A$

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

**return**  $A$

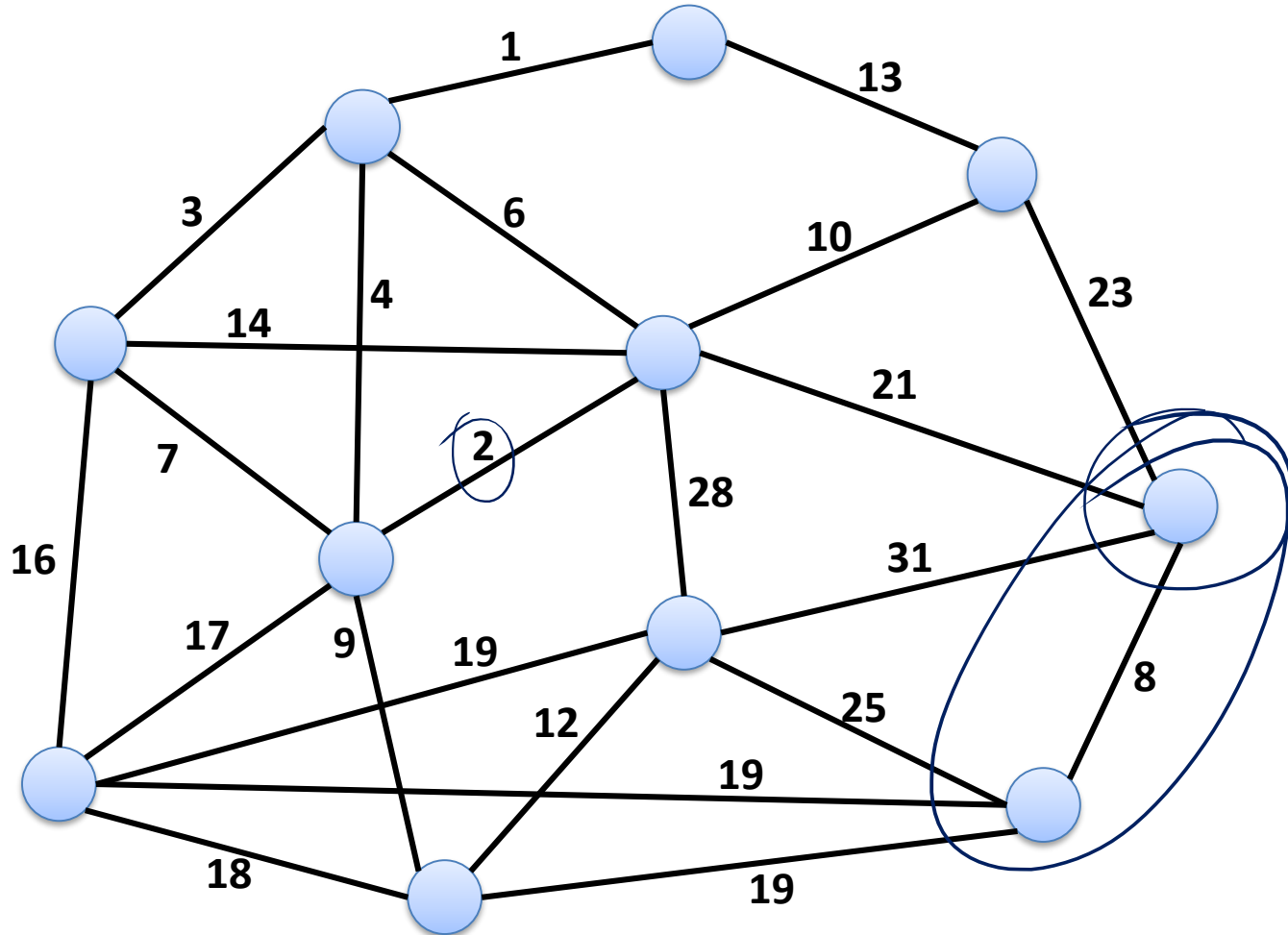


- **Idee:**  $A$  ist immer ein zusammenhängender Teilbaum

- 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.

# Prims MST-Algorithmus: Beispiel



$S := \{s\}; A := \emptyset$

**while**  $(S, A)$  ist kein Spannbaum **do**

$e = \{u, v\}$  ist Kante mit kleinstem Gewicht,  
so dass  $u \in S$  und  $v \notin S$

$S := S \cup \{v\}; A := A \cup \{\{u, v\}\}$

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

- Das folgt aber direkt, weil
  - $S$  immer genau die Knoten enthält, die in einer Kante in  $A$  enthalten sind.
  - Es hat deshalb keine Schnittkante von  $(S, V \setminus S)$  in  $A$
  - $e = \{u, v\}$  ist eine solche Kante mit kleinstem Gewicht
  - Das vorige Theorem impliziert deshalb, dass  $A$  eine sichere Kante für  $A$  ist.

# Implementierung von Prim's Algorithmus

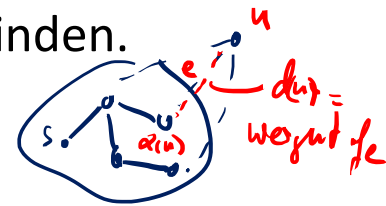
- **Knoten in  $S$  heissen markiert**
  - Das sind also genau die Knoten, die durch  $A$  bestimmten Teilbaum sind.

- **Ein Schritt des Algorithmus:**

- Man sucht eine Kante mit kleinstem Gewicht, die einen markierten Knoten (einen Knoten in  $S$ ) mit einem nicht-markierten Knoten verbindet.
- Diese Kante kann grundsätzlich beliebigen nicht markierten Knoten  $u \in V \setminus S$  mit einem beliebigen markierten Knoten in  $S$  verbinden.

- **Knoten  $u \in V \setminus S$ :**

- $\alpha(u)$  ist der nächste Nachbar von  $u$  im durch  $A$  bestimmten Teilbaum
- $d(u) = \text{dist}(u, \alpha(u))$ 
  - $d(u) = \infty$  falls  $u$  keinen Nachbar in  $S$  hat
- Wir suchen also immer einen Knoten  $u \in V \setminus S$  mit kleinstem  $d(u)$  und fügen dann die Kante  $\{u, \alpha(u)\}$  zu  $A$  hinzu.
- Dazu müssen wir die Werte  $d(u)$  nach jedem Schritt aktualisieren





# Implementierung von Prim's Algorithmus

- Knoten in  $S$  heißen markiert
- Knoten  $u$ :
  - $\alpha(u)$  ist der nächste Nachbar von  $u$  in  $V \setminus S$  (falls definiert)
  - $d(u) = \text{dist}(u, \alpha(u))$  (oder  $d(u) = \infty$  falls  $u \notin S$  oder  $\alpha(u) = \text{NULL}$ )

**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  $d(u)$

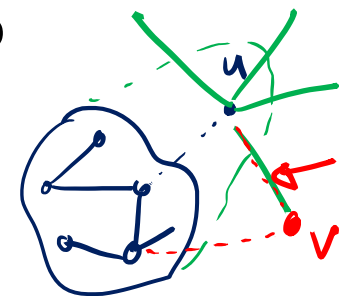
**for all** unmarked neighbors  $v$  of  $u$  **do**

**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**  $A = A \cup \{u, \alpha(u)\}$



## Heap / Priority Queue (Prioritätswarteschlange):

- 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
  - $H.getMin()$  und  $H.deleteMin()$  müssen konsistent sein
- $H.decreaseKey(x, newkey)$  : Falls  $newkey$  kleiner als der aktuelle Schlüssel von  $x$  ist, wird der Schlüssel von  $x$  auf  $newkey$  gesetzt

# Implementierung von Prim's Algorithmus

$H = \text{new priority queue}; A = \emptyset$

**for all**  $u \in V \setminus \{s\}$  **do**

$H.\text{insert}(u, \infty); \alpha(u) = \text{NULL}$

$H.\text{insert}(s, \underline{0})$

**while**  $H$  is not empty **do**

$u = H.\text{deleteMin}()$

**for all** unmarked neighbors  $v$  of  $u$  **do**

**if**  $\underline{w(\{u, v\})} < \underline{d(v)}$  **then**

$H.\text{decreaseKey}(v, w(\{u, v\}))$

$\alpha(v) = u$

$u.\text{marked} = \text{true}$

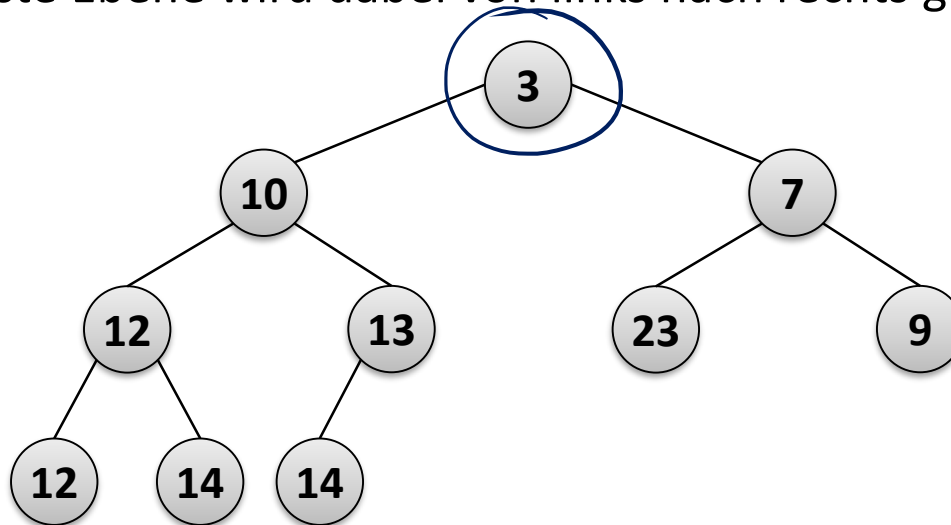
**if**  $u \neq s$  **then**  $A = A \cup \{u, \alpha(u)\}$

## Anzahl Priority Queue Operationen

- **create**  $1$
- **insert**  $\underline{O(n)}$  (jeder Knoten genau einmal)
- **getMin / deleteMin**  $\underline{O(n)}$  (jeder Knoten genau einmal)
- **decreaseKey**  $\underline{O(m)}$  (für jede Kante höchstens einmal, dann wenn der erste Knoten der der Kante zu  $S$  hinzugefügt wird)

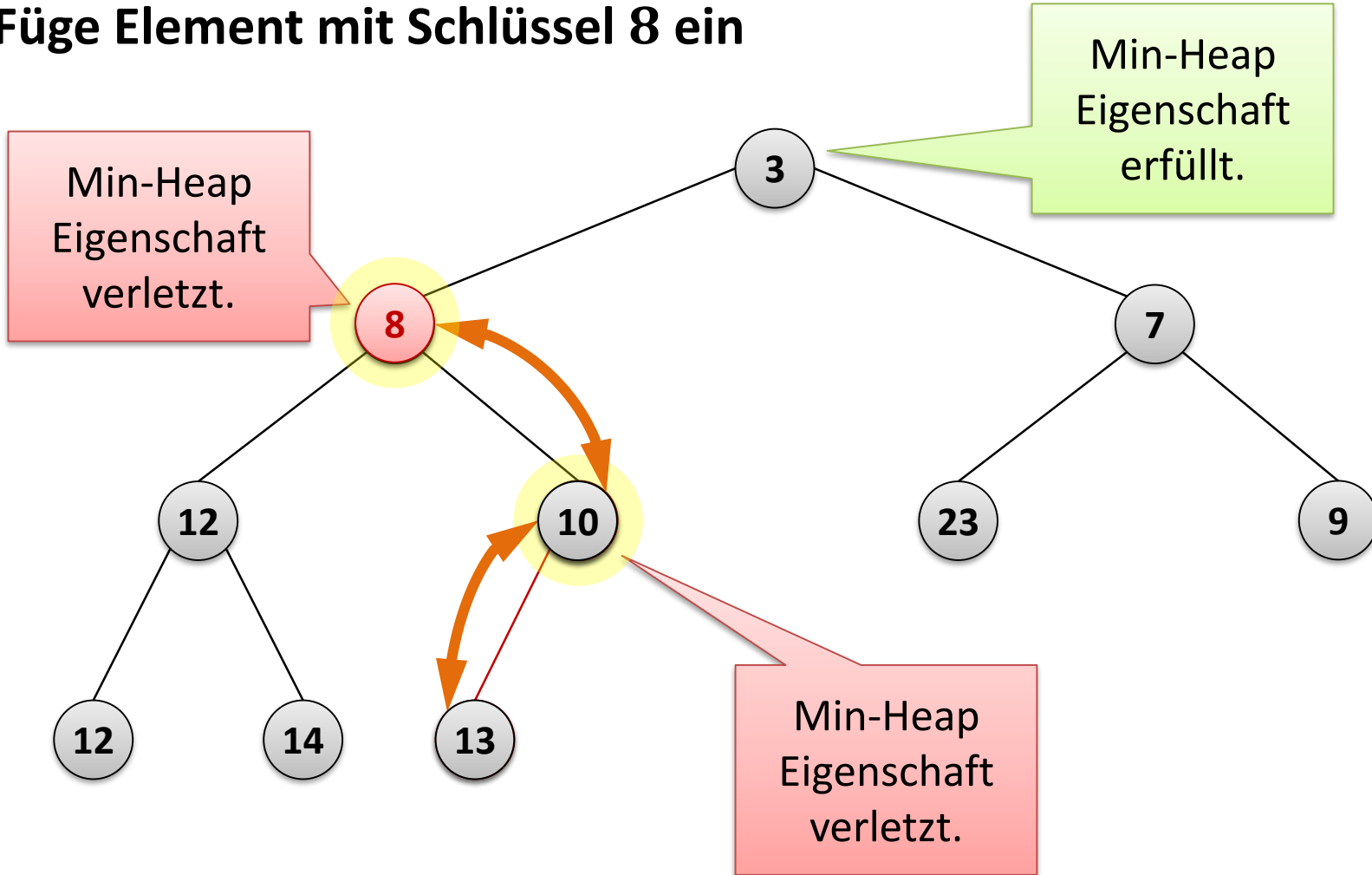
## Implementierung als Binärbaum mit Min-Heap Eigenschaft

- 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
  - Alle ausser der untersten Ebene sind voll.
  - Die unterste Ebene wird dabei von links nach rechts gefüllt.



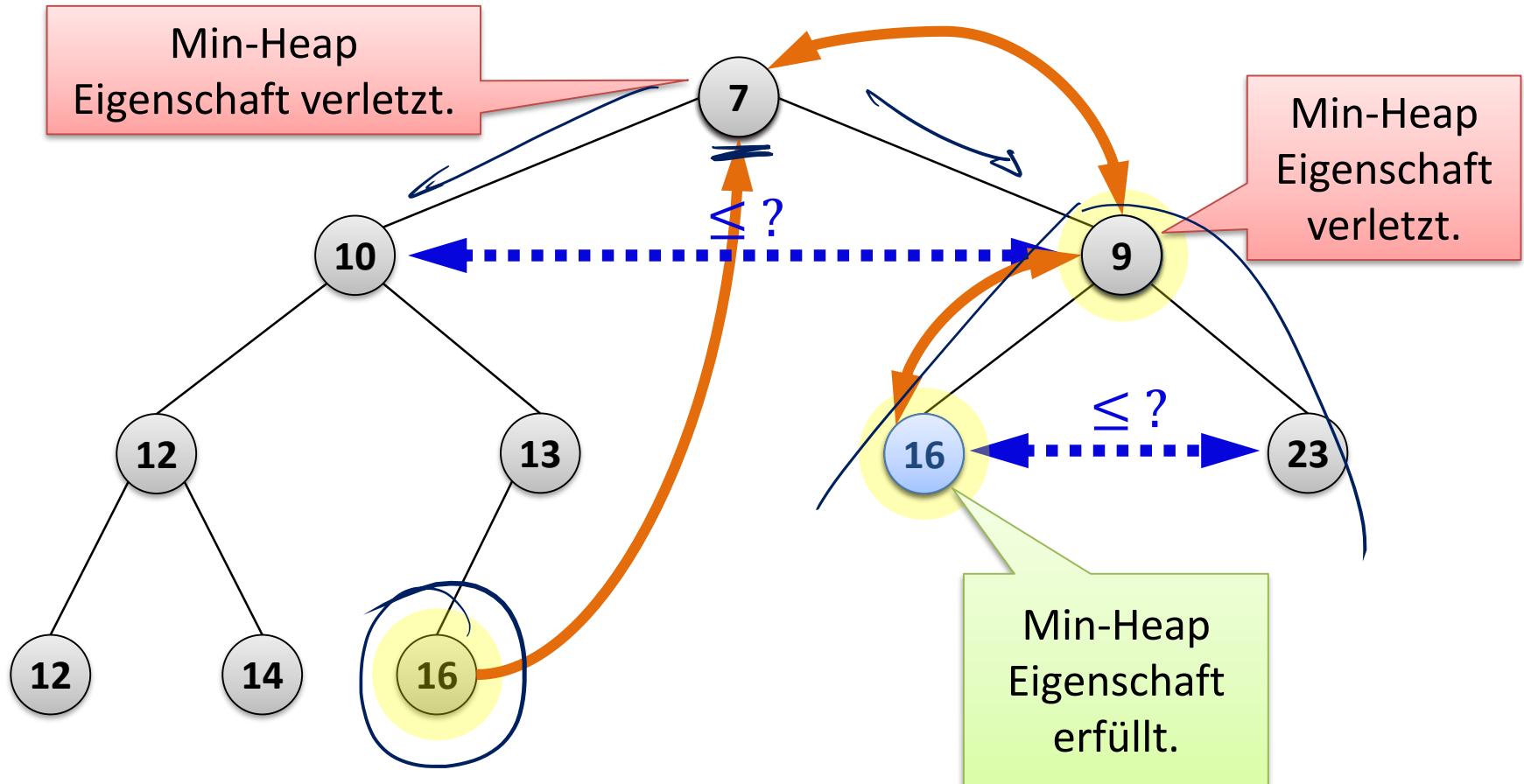
# Prioritätswarteschlangen: Einfügen

Füge Element mit Schlüssel 8 ein



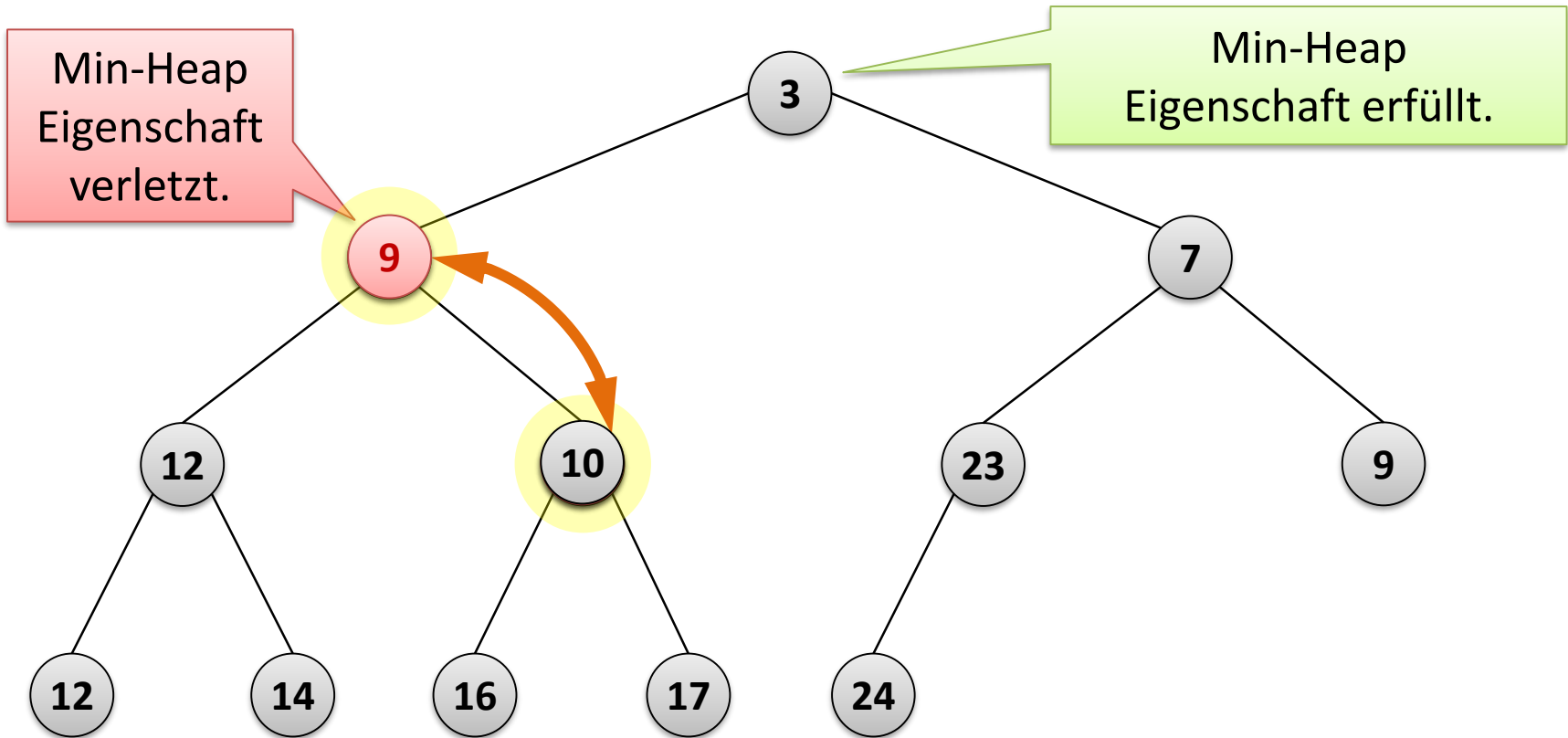
# Prioritätswarteschlangen: Delete-Min

Lösche das Element in der Wurzel (mit kleinstem Schlüssel)



# Prioritätswarteschlangen: Decrease-Key

Decrease Key: Knoten mit Schlüssel 13  $\Rightarrow$  neuer Schlüssel 9



Für die decrease-key Operation muss man eine Referenz auf den Knoten gegeben haben, dessen Schlüssel verkleinert werden soll.



- Die besprochene Variante heiße auch **binärer Heap**
  - durch einen Binärbaum mit Min-Heap-Eigenschaft implementiert
- **Höhe (oder Tiefe) des Baums** ist immer genau  $\lfloor \log_2 n \rfloor$ 
  - Anzahl Knoten in einem vollen Binärbaum der Höhe  $i$  ist  $2^{i+1} - 1$

Anzahl Knoten in Abstand  $j$  zur Wurzel ist  $2^j$ :

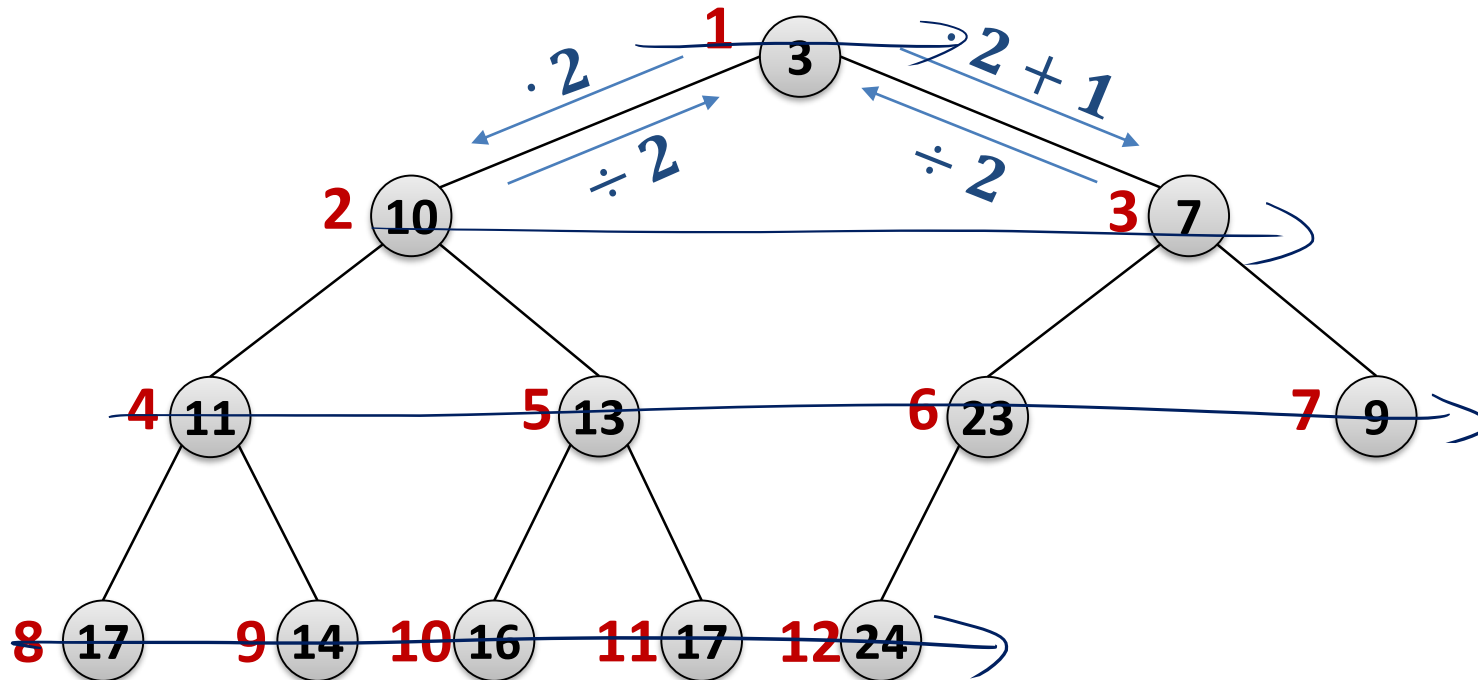
$$\#\text{Knoten} = \sum_{j=0}^i 2^j = 2^{i+1} - 1.$$

- **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 in den Positionen 1 bis  $n$

- Das geht, weil der Baum perfekt balanciert ist



- Von Knoten an Position  $i$ 
  - Linkes Kind an Position  $j = 2 \cdot i$ , rechtes Kind an Position  $j = 2 \cdot i + 1$
  - Parent an Position  $j = i/2$  (Ganzzahl-Division, d.h.,  $j = \lfloor i/2 \rfloor$ )

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

# Prims Algorithmus mit binären Heaps

```
 $H = \text{new BinaryHeap}(); A = \emptyset$   
for all  $u \in V \setminus \{s\}$  do  
     $H.\text{insert}(u, \infty); \alpha(u) = \text{NULL}$   
 $H.\text{insert}(s, 0)$   
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  
             $H.\text{decreaseKey}(v, w(\{u, v\})); \alpha(v) = u$   
     $u.\text{marked} = \text{true}$   
    if  $u \neq s$  then  $A = A \cup \{\{u, \alpha(u)\}\}$ 
```

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

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

# Prims Algorithmus ohne Decrease-Key

```
H = new BinaryHeap(); A = ∅  
for all u ∈ V \ {s} do  
    H.insert(u, ∞);  $\alpha(u) = \text{NULL}$   
H.insert(s, 0)  
while H is not empty do  
    u = H.deleteMin()  
    if not u.marked then  
        for all unmarked neighbors v of u do  
            if  $w(\{u, v\}) < d(v)$  then  
                H.insert(v,  $w(\{u, v\})$ );  $\alpha(v) = u$   
            u.marked = true  
        if  $u \neq s$  then A = A ∪ {u,  $\alpha(u)$ }
```

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

- $O(m)$  insert-Operationen und deleteMin-Operationen

## 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  
(in diesem Fall muss man es mit decrease-key implementieren)

# Kruskal's MST-Algorithmus

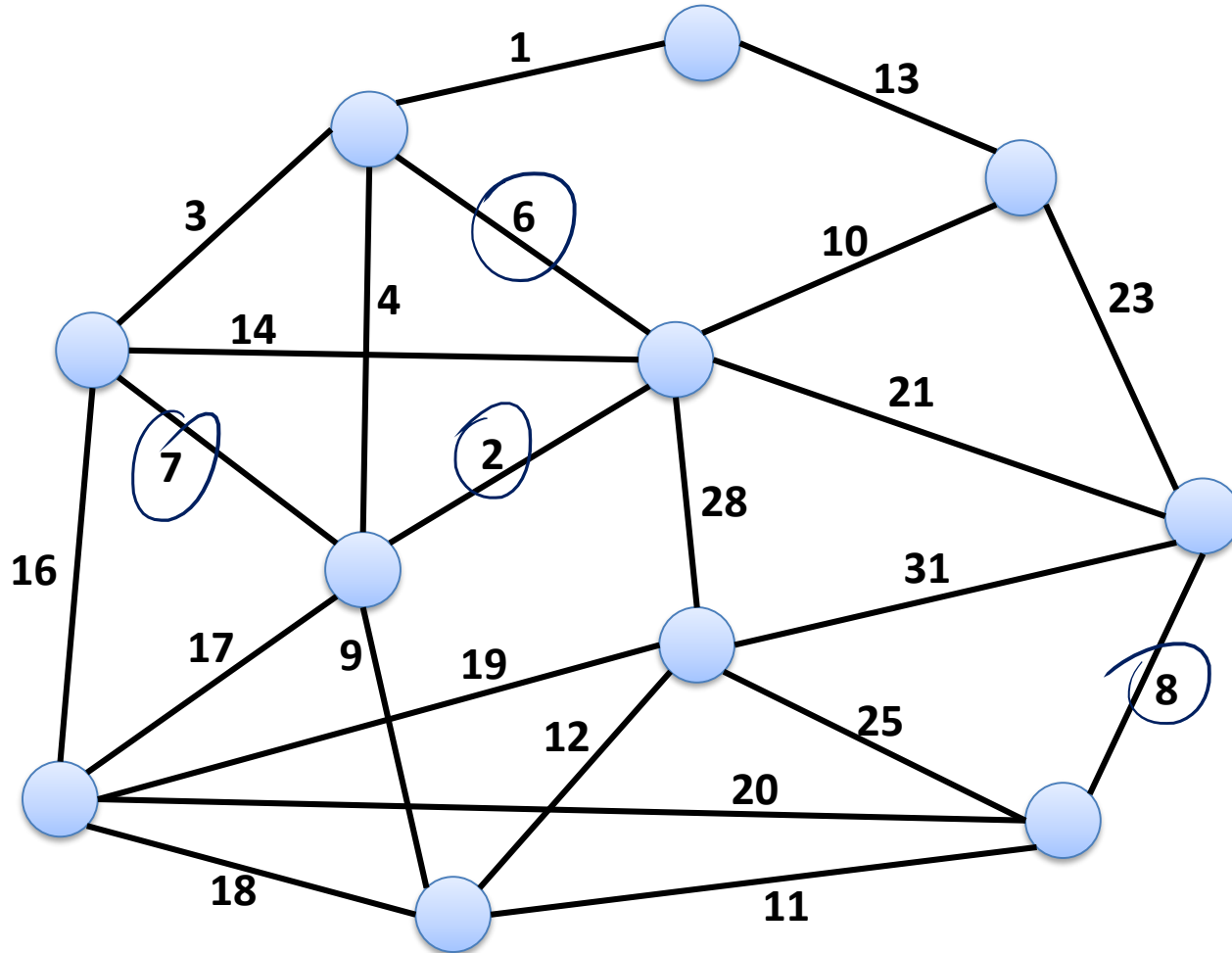
$A = \emptyset$

**while**  $A$  ist kein Spannbaum **do**

$e = \{u, v\}$  ist Kante mit kleinstem Gewicht,  
so dass  $A \cup \{\{u, v\}\}$  keinen Zyklus enthält

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

# Kruskal's MST-Algorithmus: Beispiel





# Kruskal's MST-Algorithmus

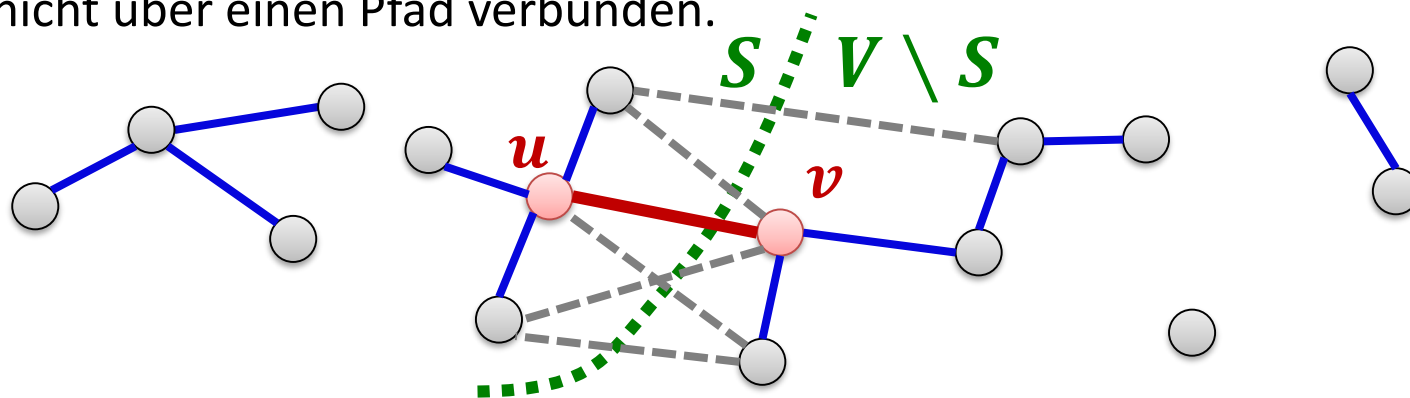
$A = \emptyset$

**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
  - Da  $A \cup \{\{u, v\}\}$  keinen Zyklus enthält, sind  $u$  und  $v$  durch die Kanten in  $A$  nicht über einen Pfad verbunden.



- Es gibt einen Schnitt  $(S, V \setminus S)$ , so dass  $A$  keine Schnittkanten enthält, so dass  $u \in S$  und  $v \in V \setminus S$  ist, und  $\{u, v\}$  eine leichteste Schnittkante ist.

## Kruskals Algorithmus (Pseudocode)

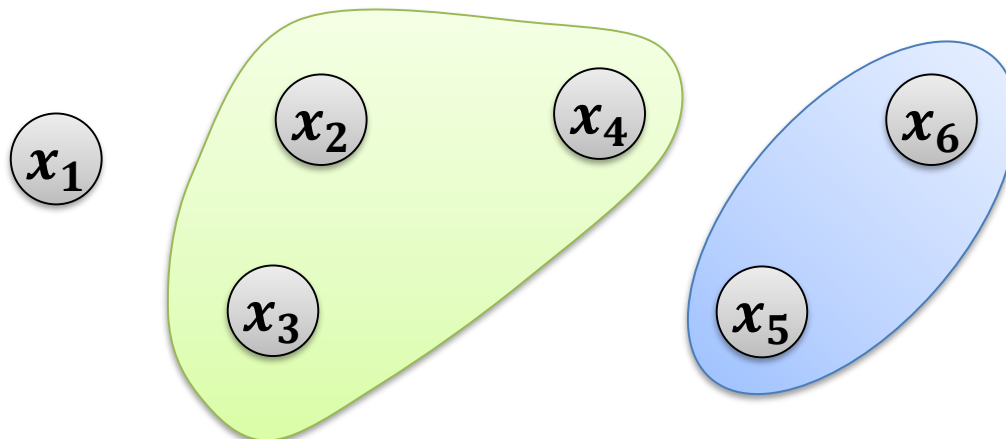
1.  $A = \emptyset$
  2. Sortiere Kanten aufsteigend nach Kantengewicht
  3. **for**  $e = \{u, v\} \in E$  (in sorted order) **do**
  4.     **if**  $u$  and  $v$  are in different components **then**
  5.          $A = A \cup \{e\}$
- Müssen **Komponenten** des durch  $A$  bestimmten Graphen effizient **verwalten** können
  - **Laufzeit:**  $O(m \log n)$  für's Sortieren, sowie die Gesamtzeit, um die Komponenten zu verwalten...

## Union-Find / Disjoint Sets:

- 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  $S2$



## Kruskals Algorithmus

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

## Beste Union-Find Datenstruktur

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

$$O(\underline{m} \cdot \underline{\alpha(n)})$$

- $\alpha(n)$  ist die Inverse der Ackermannfunktion und wächst extrem langsam (für alle in der Praxis möglichen  $n$ ,  $\alpha(n) \leq 5$ )

## Laufzeit Kruskal

- Kanten sortieren:  $O(m \cdot \log n)$ 
  - Falls die Gewichte ganze Zahlen im Bereich  $0, \dots, n^{O(1)}$  sind, kann man mit Radix-Sort in Linearzeit sortieren.
- Union-Find-Operationen:  $O(m \cdot \alpha(n))$
- Insgesamt:  $O(m \cdot \log n)$ 
  - besser, falls Kantengewichte schneller sortiert werden können

Beide Algorithmen sind typische Beispiele für sogenannte

## **Greedy Algorithmen**

- Bei Greedy Algorithmen wird eine Lösung schrittweise aufgebaut.
- In jedem Schritt wird ein aktuell bestes “Element” hinzugefügt.
- Bereits berechnete Teile der Lösung werden nicht mehr verändert.

## **Prim und Kruskal Algorithmen zur Berechnung eines MST**

- Wir beginnen mit einer leeren Kantenmenge
- In jedem Schritt wird die im Moment beste Kante hinzugenommen
  - Bei Prim: beste Kante, die den Teilbaum zusammenhängend belässt.
  - Bei Kruskal: beste Kante, so dass man alles zu einem Baum erweitern kann.
- Eine gewählte Kante wird nie wieder verworfen