Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

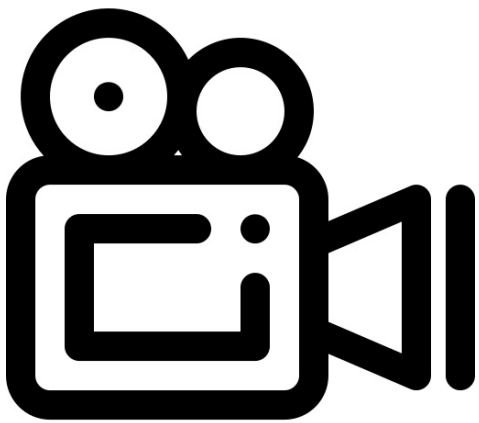
Minimale aufspannende Bäume (Teil 2) & Effiziente Datenstrukturen

7. Vorlesung

Aufzeichnung der Vorlesung

Diese Vorlesung wird aufgezeichnet und live gestreamt.

- Aufzeichnungen nur der Lehrenden durch sich selbst.
- Bei Rückfragen aus dem Auditorium und Diskussion bitte deutlich anzeigen, falls das Mikro stumm geschaltet werden soll.



Organisatorisches

Evaluation:

- Die Lehrevaluation läuft vom 27. Nov. bis zum 10. Dez.
- Bitte teilnehmen!
- Bei mindestens 130 Evaluationen kriegen alle Studenten 5 Bonuspunkte in der Übung

Prioritätsschlangen durch Binary Heaps

Prioritätsschlangen

Bekannt: Warteschlange (FIFO-Prinzip)

Prioritätsschlangen

Bekannt: Warteschlange (FIFO-Prinzip)

Prioritätsschlangen (Bsp.: Notaufnahme im Krankenhaus)

Prioritätsschlangen

Bekannt: Warteschlange (FIFO-Prinzip)

Prioritätsschlangen (Bsp.: Notaufnahme im Krankenhaus)

- Jedes Element in der Schlange hat eine Priorität $p \in \mathbb{R}_+$

Prioritätsschlangen

Bekannt: Warteschlange (FIFO-Prinzip)

Prioritätsschlangen (Bsp.: Notaufnahme im Krankenhaus)

- Jedes Element in der Schlange hat eine Priorität $p \in \mathbb{R}_+$
- Wir entnehmen immer das Element mit der niedrigsten Priorität

Prioritätsschlangen

Bekannt: Warteschlange (FIFO-Prinzip)

Prioritätsschlangen (Bsp.: Notaufnahme im Krankenhaus)

- Jedes Element in der Schlange hat eine Priorität $p \in \mathbb{R}_+$
- Wir entnehmen immer das Element mit der niedrigsten Priorität
- Weitere Operationen
  - `Insert` - ein neues Element einfügen
  - `Delete` - ein Element aus der Schlange löschen
  - `UpdatePriority` - Priorität von Element ändern

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht
  - Laufzeit: $\mathcal{O}(n)$ (n ist Anzahl der Elemente in der Schlange)

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht
  - Laufzeit: $\mathcal{O}(n)$ (n ist Anzahl der Elemente in der Schlange)
- andere Operationen ähnlich

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht
  - Laufzeit: $\mathcal{O}(n)$ (n ist Anzahl der Elemente in der Schlange)
    $\rightarrow$ zu langsam
- andere Operationen ähnlich

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht
  - Laufzeit: $\mathcal{O}(n)$ (n ist Anzahl der Elemente in der Schlange)
    $\rightarrow$ zu langsam
- andere Operationen ähnlich

Beobachtung: Die Elemente müssen nicht komplett sortiert sein, es muss nur das kleinste Element am Anfang stehen.

Einfacher Ansatz

Wir implementieren die Prioritätsschlangen als sortierte Liste

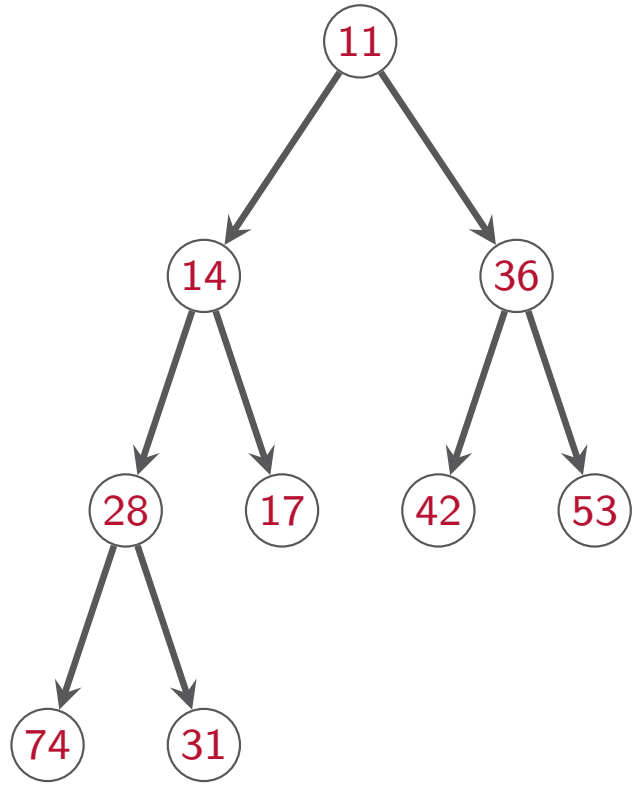
- sortiert nach Priorität, kleinste Priorität steht am Anfang
- Elemente können wie folgt eingefügt werden:
  - Hänge Element am Ende der Liste an
  - Tausche mit vorherigem Element bis es an der richtigen Stelle steht
  - Laufzeit: $\mathcal{O}(n)$ (n ist Anzahl der Elemente in der Schlange)
    $\rightarrow$ zu langsam
- andere Operationen ähnlich

Beobachtung: Die Elemente müssen nicht komplett sortiert sein, es muss nur das kleinste Element am Anfang stehen.

Idee: Nutze Binärbaum, in dem jeder Pfad sortiert ist
$\rightarrow$ Binary Heap

Binary Heap

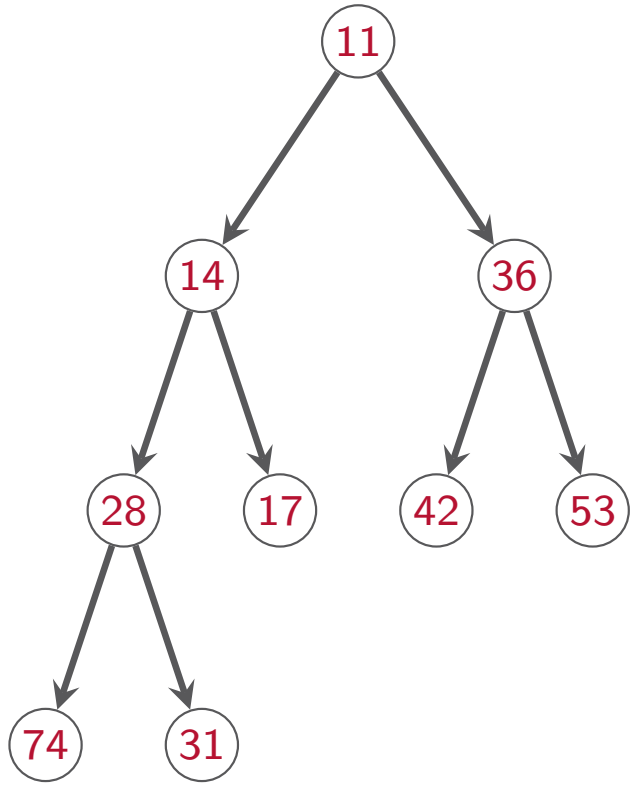
Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

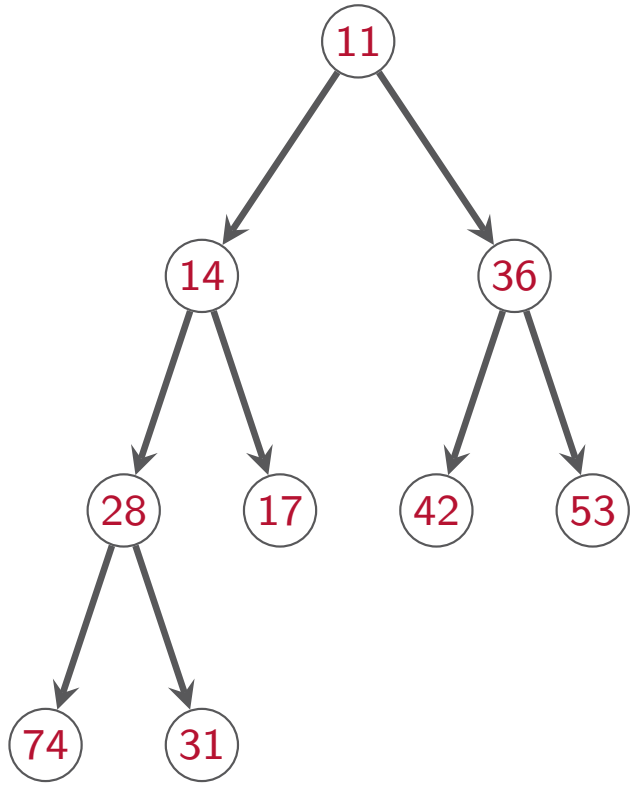
- Es gibt eine spezielle Wurzel r



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

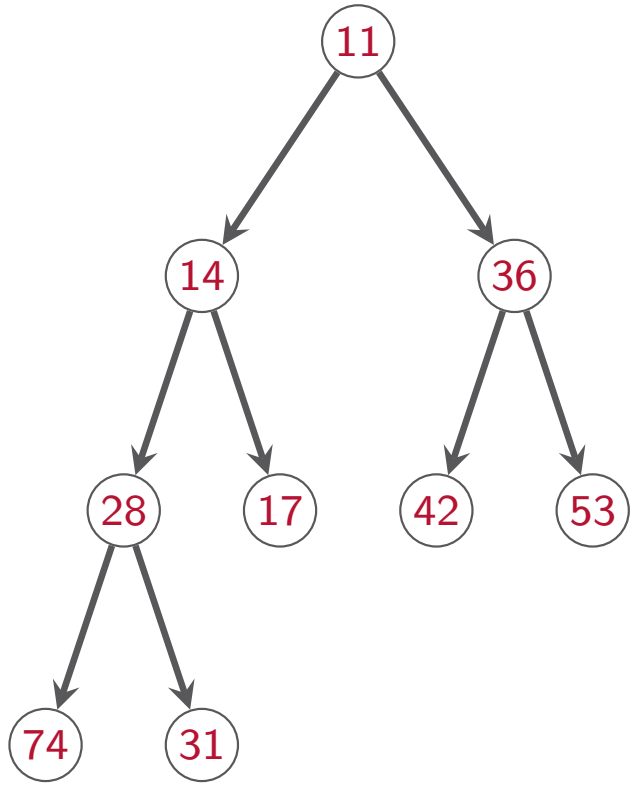
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält eine Priorität $C[v]$



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

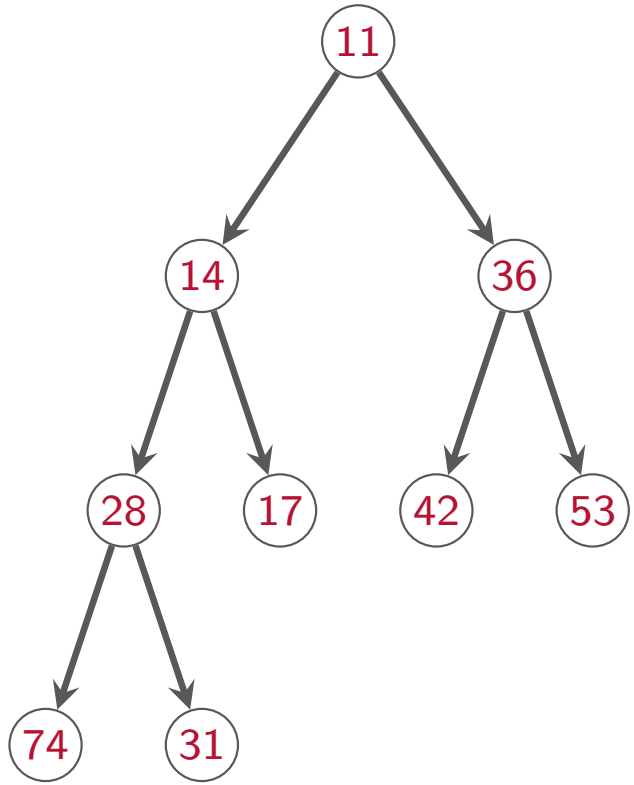
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält eine Priorität $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

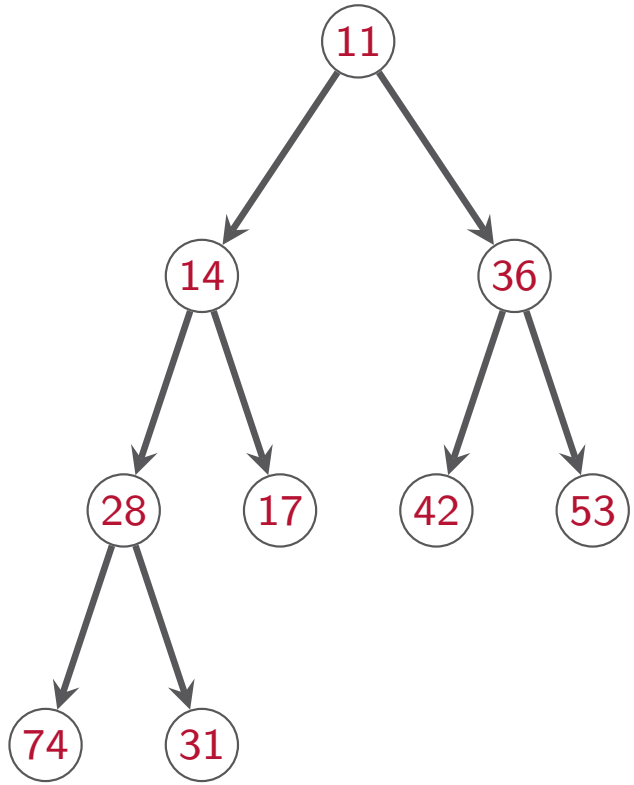
- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält eine Priorität $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt



Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält eine Priorität $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt
- Das letzte "Level" ist von links besetzt



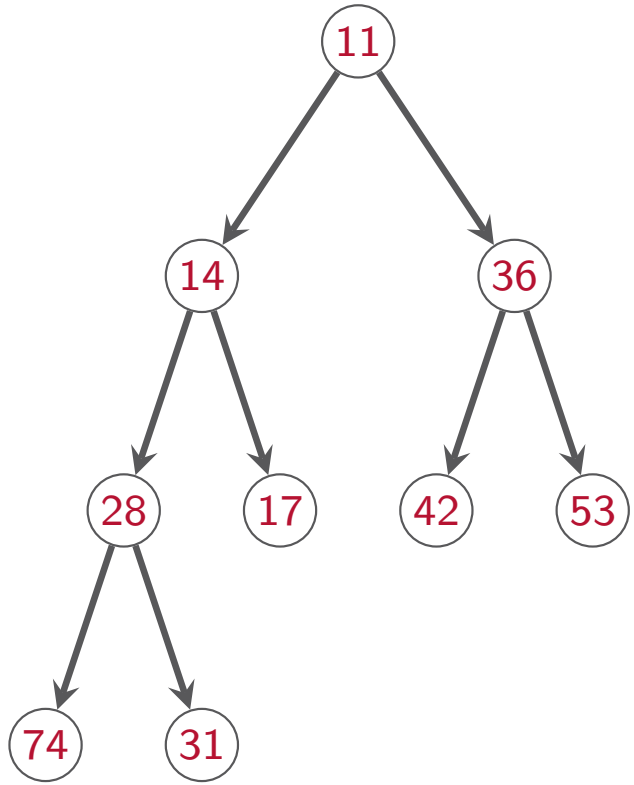
Binary Heap

Ein Binary Heap ist eine auf Bäumen basierende Datenstruktur:

- Es gibt eine spezielle Wurzel r
- Jeder Knoten v im Baum enthält eine Priorität $C[v]$
- Jeder Knoten hat genau zwei Nachfolger oder ist ein Blatt
- Nur das letzte "Level" ist nicht voll besetzt
- Das letzte "Level" ist von links besetzt
- Für jeden Knoten v mit zwei Nachfolgern w_1 und w_2 gilt

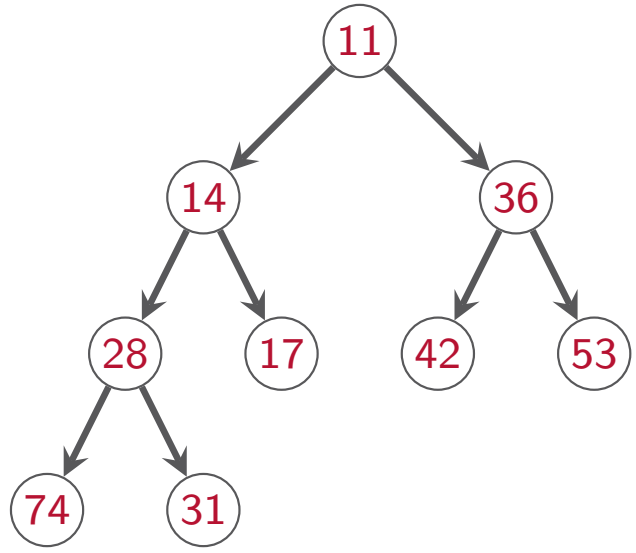
$$C[v] \leq \min(C[w_1], C[w_2])$$

d.h., alle Wurzel-zu-Blatt Pfade sind sortiert



Binary Heap - Repräsentation als Array

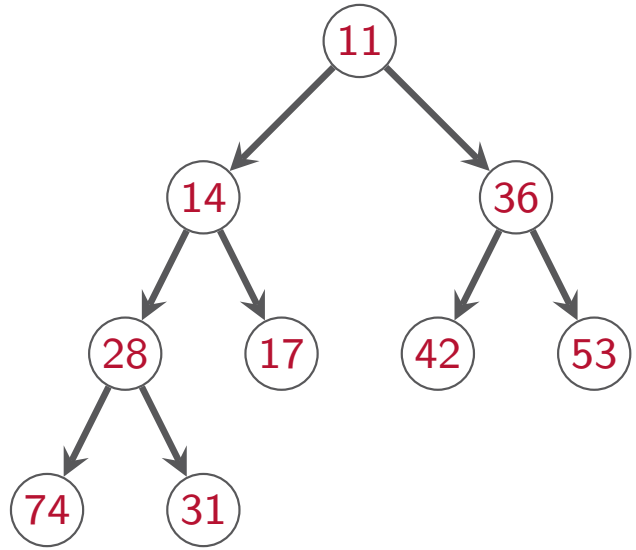
Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:



Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1

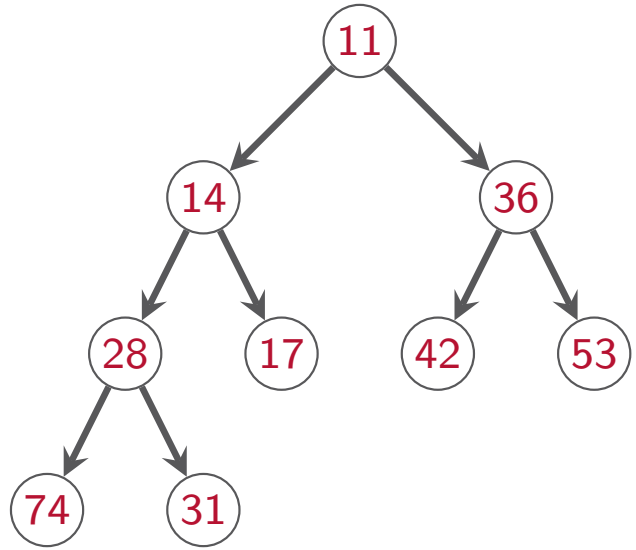


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1
- Nachfolger von Knoten i sind an Positionen $2i$ und $2i + 1$



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

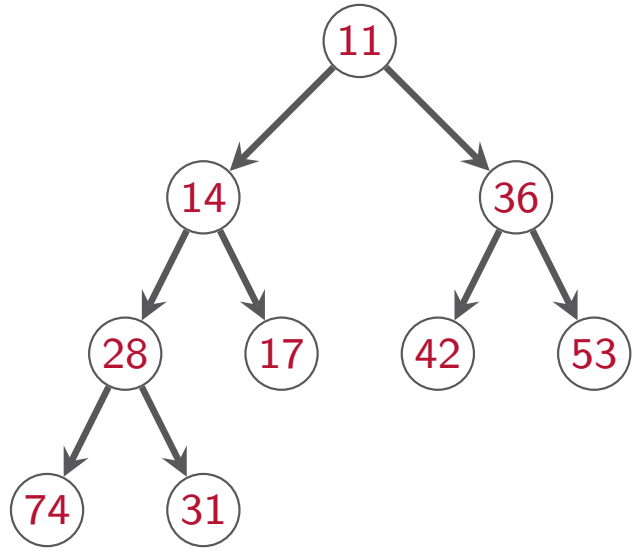
Binary Heap - Repräsentation als Array

Wir können einen Binary Heap mit n Knoten als Array C der Länge n (mit Startindex 1) abspeichern:

- Wurzel an Position 1
- Nachfolger von Knoten i sind an Positionen $2i$ und $2i+1$
- Für jedes $i \in \{1, \ldots, \lfloor n/2 \rfloor\}$ gilt

$$C[i] \leq \min(C[2i], C[2i+1])$$

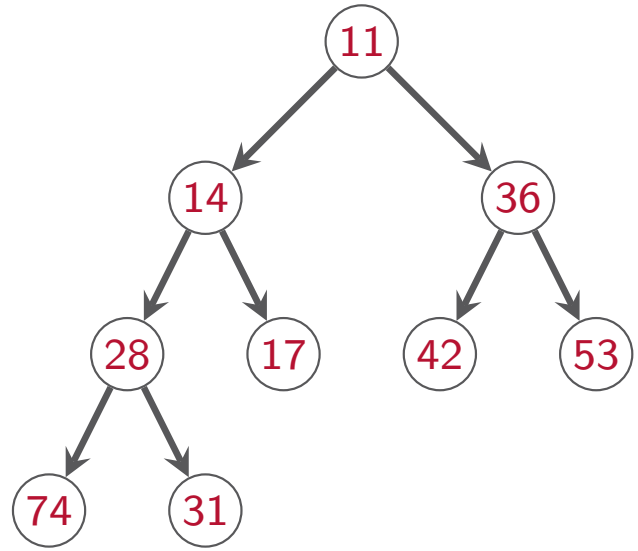
(falls $C[2i+1]$ nicht existiert, nehmen wir es als ∞ an)



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

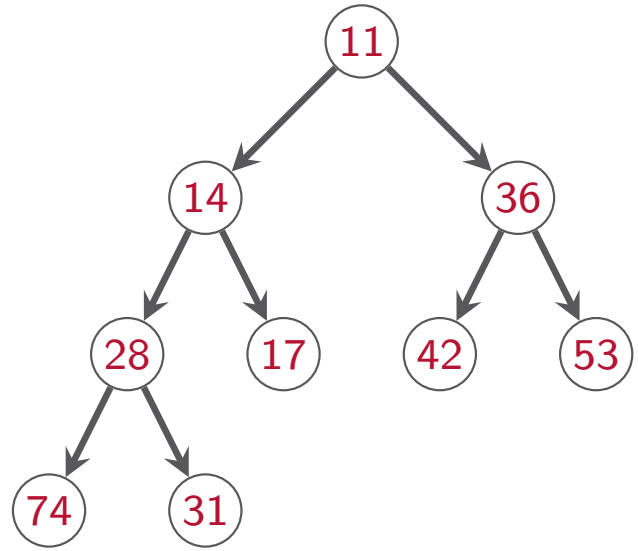


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

- Minimales Element ist immer an Position 1, wir geben also einfach $C[1]$ zurück

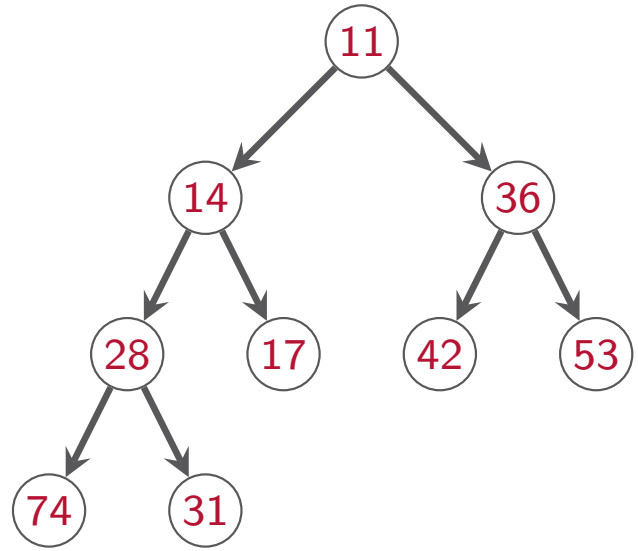


11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `FindMin`

Die Operation `FindMin` gibt das minimale Element im Binary Heap zurück.

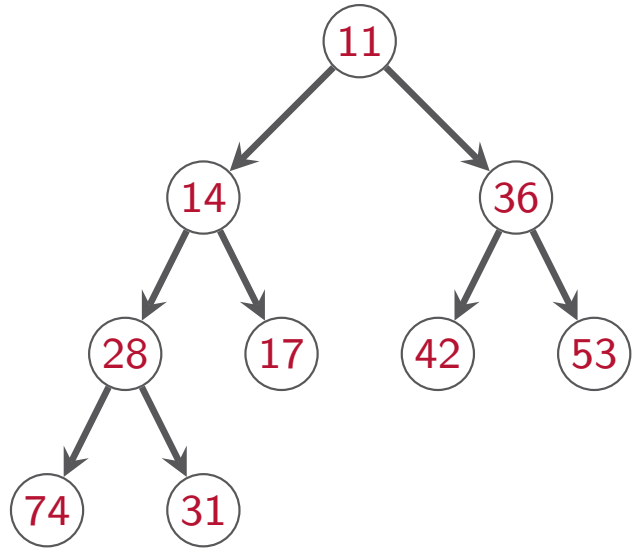
- Minimales Element ist immer an Position 1, wir geben also einfach $C[1]$ zurück
- Laufzeit: $\mathcal{O}(1)$



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

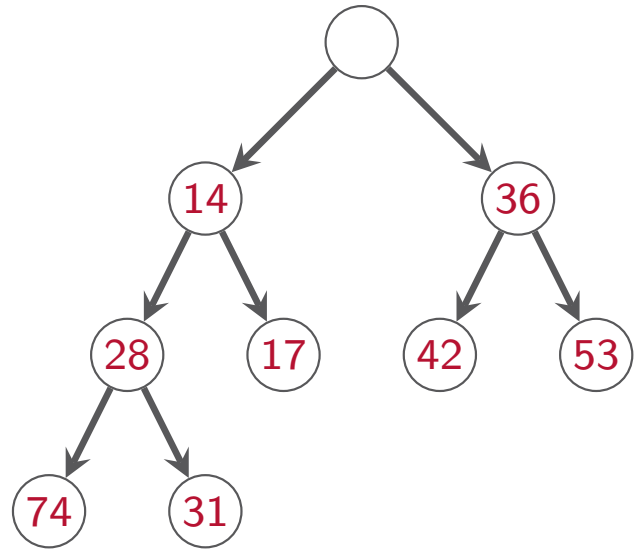
Die Operation `Delete` löscht das Element an Position (oder Knoten) i.



11	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

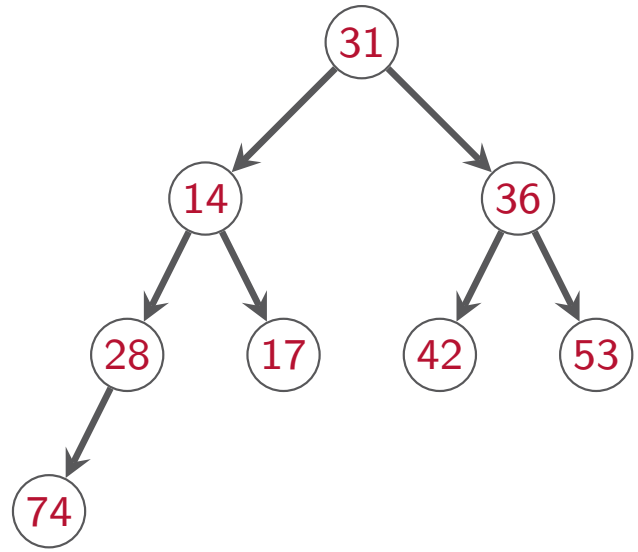


	14	36	28	17	42	53	74	31
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array

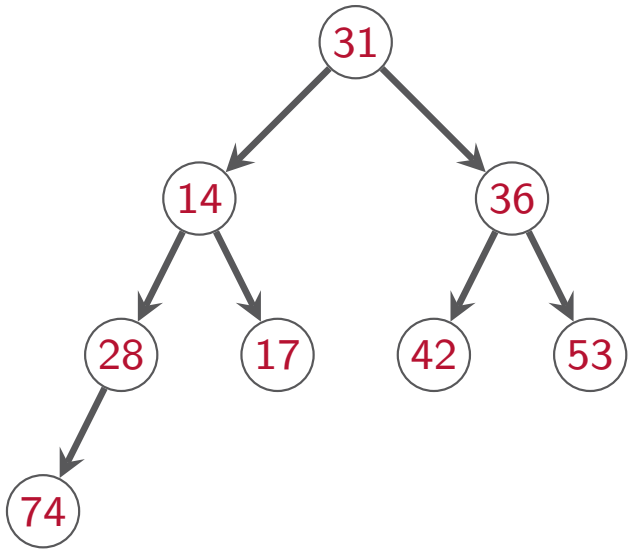


31	14	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen

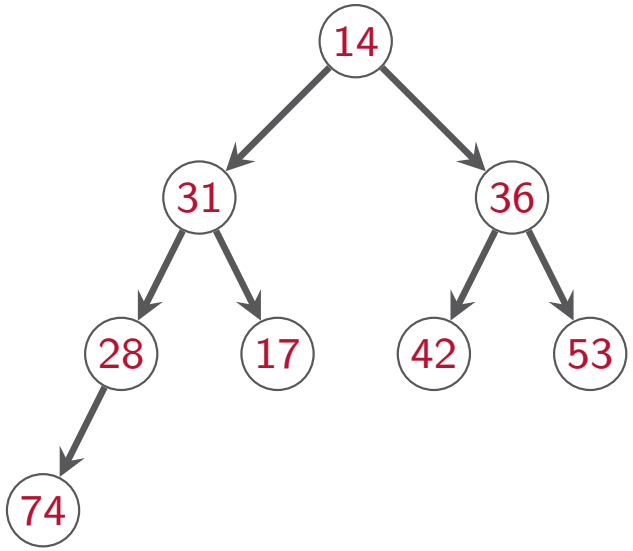


31	14	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente

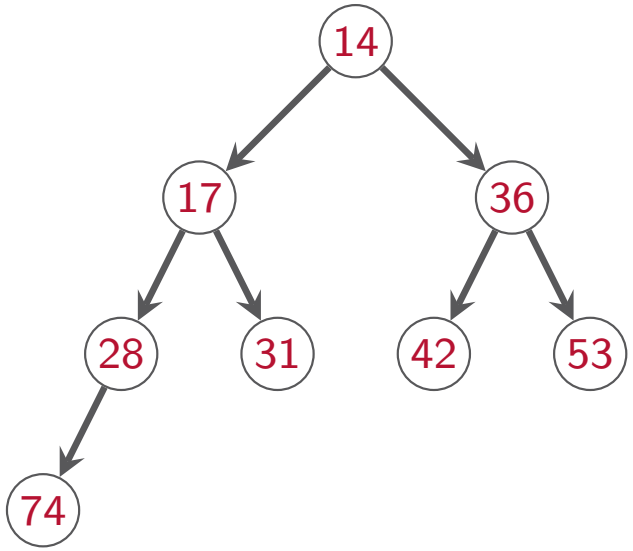


14	31	36	28	17	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente
- Wiederhole rekursiv für das entsprechende Kind

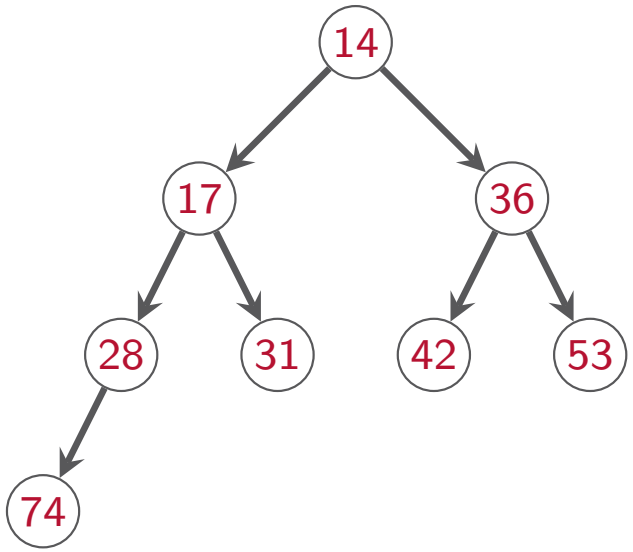


14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Delete`

Die Operation `Delete` löscht das Element an Position (oder Knoten) i.

- Vertausche Positionen i und n und entferne das letzte Element im Array
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Falls $C[i] > \min(C[2i], C[2i+1])$, dann Vertausche Position i mit dem kleineren der beiden Elemente
- Wiederhole rekursiv für das entsprechende Kind
- Laufzeit: $\mathcal{O}(\log n)$

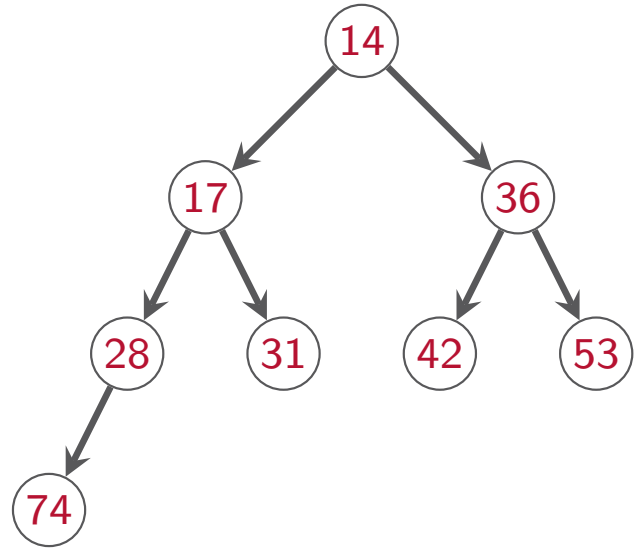


14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.



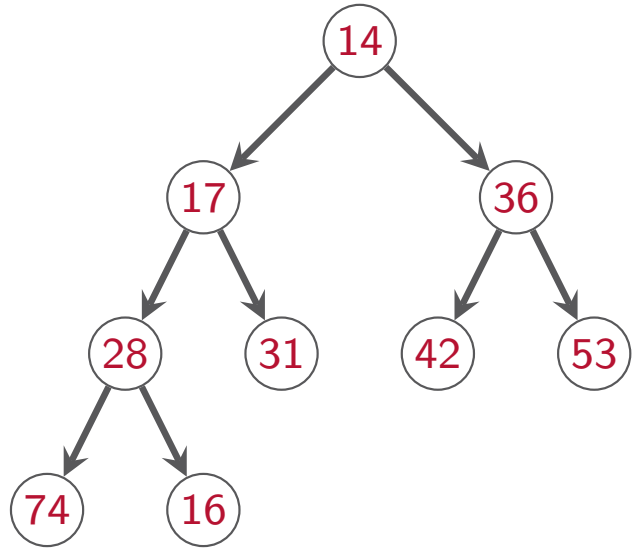
14	17	36	28	31	42	53	74	
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$



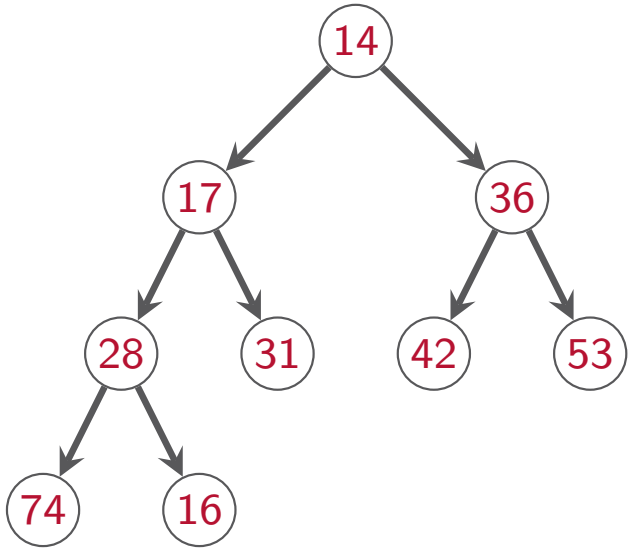
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen



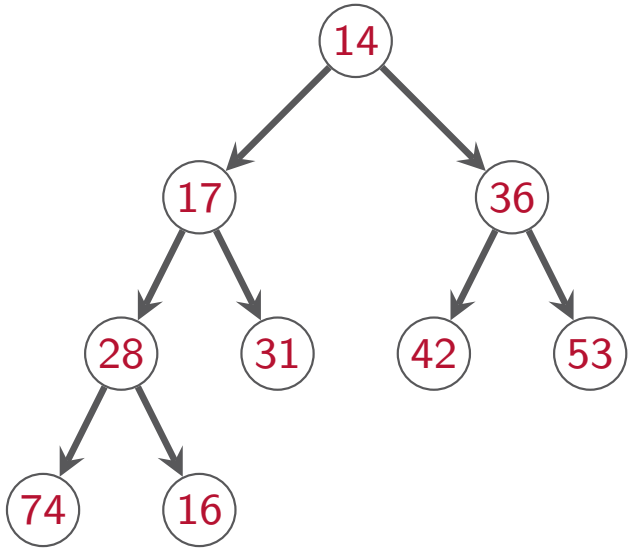
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger



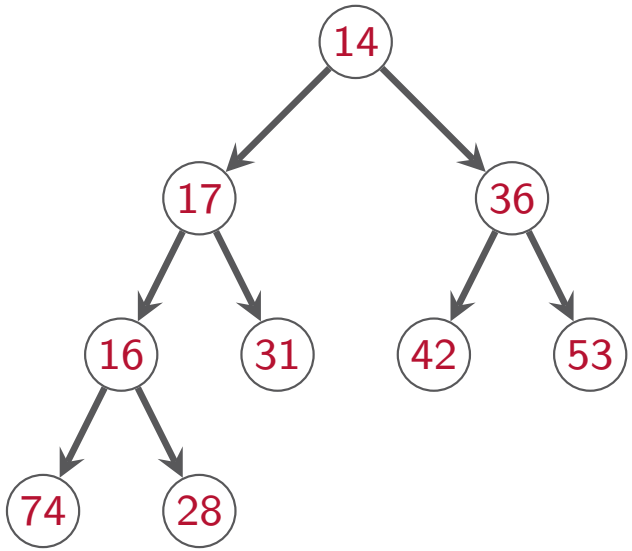
14	17	36	28	31	42	53	74	16
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche Positionen $n+1$ und i



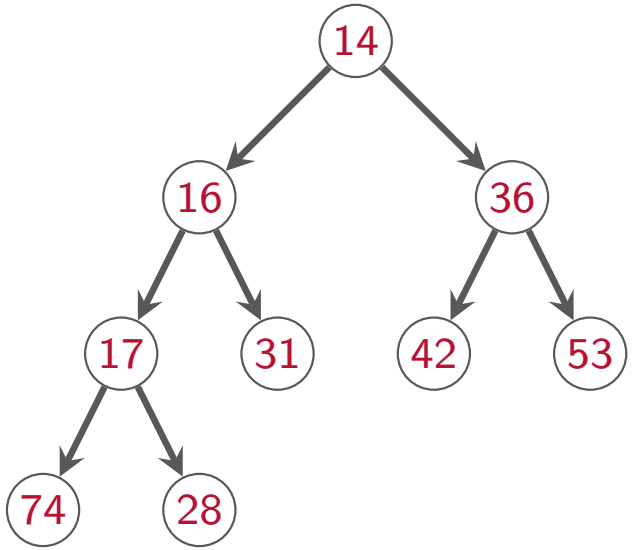
14	17	36	16	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche Positionen $n+1$ und i
- Wiederhole rekursiv für Position i



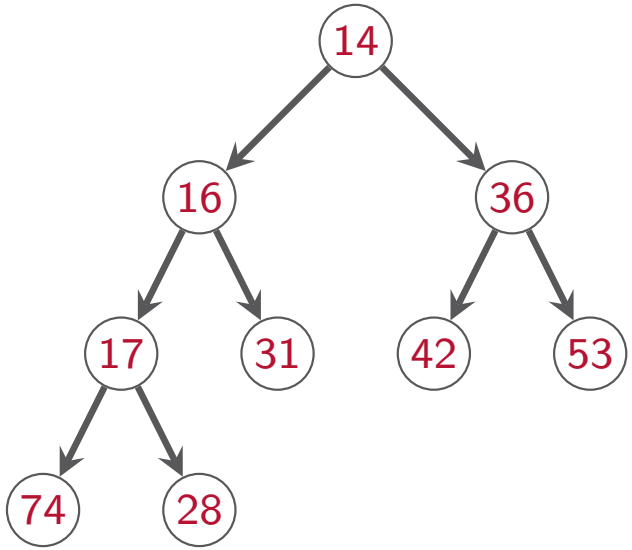
14	16	36	17	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Operation `Insert`

Die Operation `Insert` fügt ein neues Element mit Priorität c hinzu.

Annahme: Das Array hat mindestens eine leere Position am Ende.

- Setze $C[n+1] := c$
- Wir müssen die Heap-Eigenschaft wiederherstellen
- Sei $i := \lfloor (n+1)/2 \rfloor$ der Vorgänger
- Falls $C[n+1] < C[i]$, dann Vertausche Positionen $n+1$ und i
- Wiederhole rekursiv für Position i
- Laufzeit: $\mathcal{O}(\log n)$



14	16	36	17	31	42	53	74	28
1	2	3	4	5	6	7	8	9

Binary Heap - Zusammenfassung

Mit einem Binary Heap können wir eine Menge von Werten verwalten und die folgenden Operationen durchführen:

- `FindMin` - gebe das minimale Element zurück $\mathcal{O}(1)$
- `Delete` - entferne Element an Position i $\mathcal{O}(\log n)$
- `Insert` - füge neues Element mit Wert c hinzu $\mathcal{O}(\log n)$

Beachte: Die Werte können beliebige Objekte sein, solange wir sie miteinander vergleichen können (z.B. gewichtete Kanten in einem Graphen)

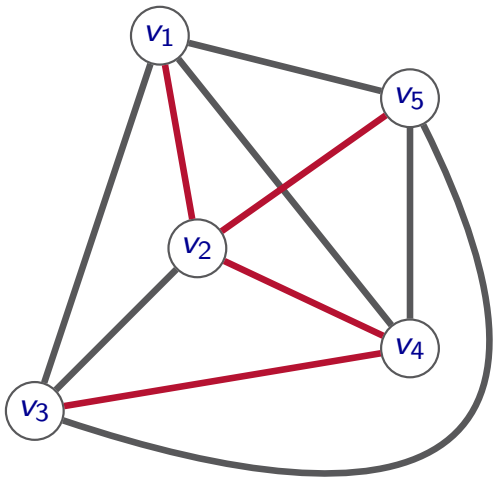
Damit können wir Prioritätsschlangen implementieren, in denen alle gewünschten Operationen maximal Zeit $\mathcal{O}(\log n)$ benötigen.

Aufspannende Bäume (Fortgesetzt)

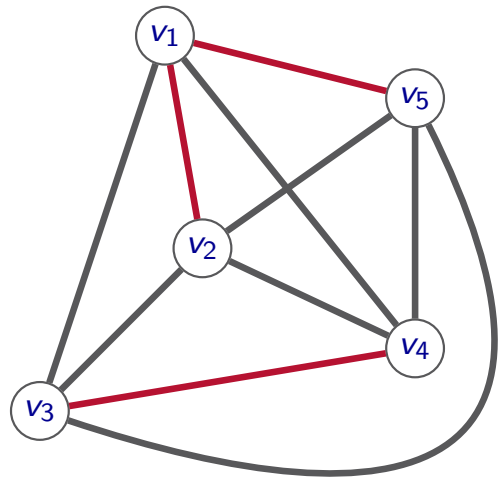
Aufspannender Baum (Spannbaum)

Definition

Sei $G = (V, E)$ ein ungerichteter, zusammenhängender Graph. Ist der Teilgraph $T = (V, E')$ von G mit $E' \subseteq E$ ein Baum, dann heißt T aufspannender Baum (Spannbaum, spanning tree) von G.



Spannbaum T



Kein Spannbaum G'

Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

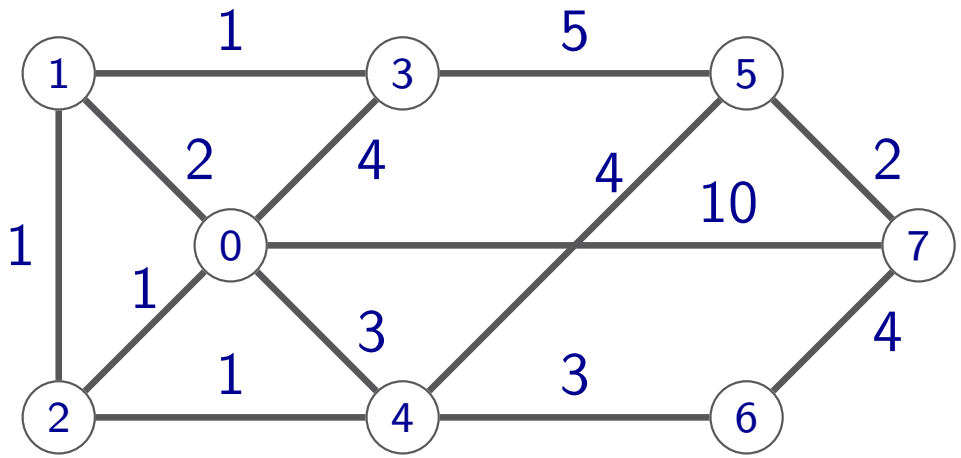
Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.

Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.

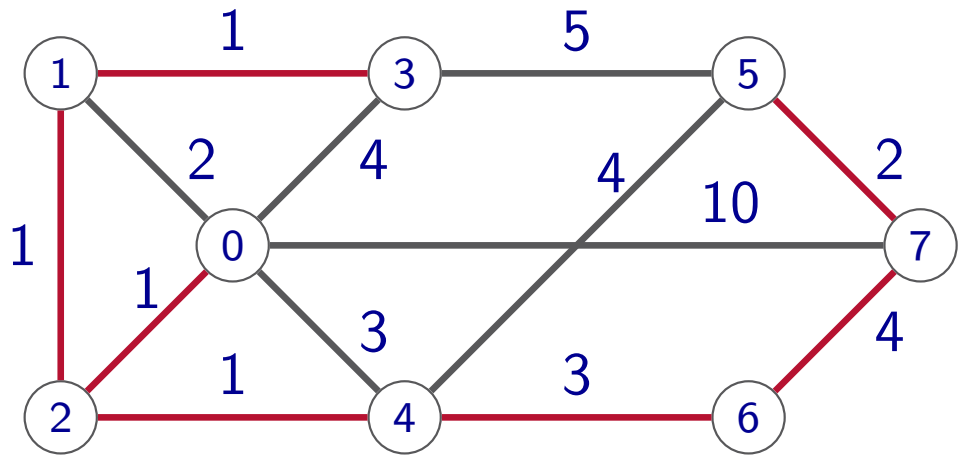


Minimaler aufspannender Baum

Minimaler Spannbaum Problem (MST)

Gegeben: Ein ungerichteter, zusammenhängender, gewichteter Graph $G = (V, E, c)$ mit Kantenkosten $c(e) \in \mathbb{R}$ für alle $e \in E$.

Gesucht: Eine kostenminimale Teilmenge $T \subseteq E$ der Kanten, so dass der Teilgraph $G[T] = (V, T)$ aufspannender Baum ist.



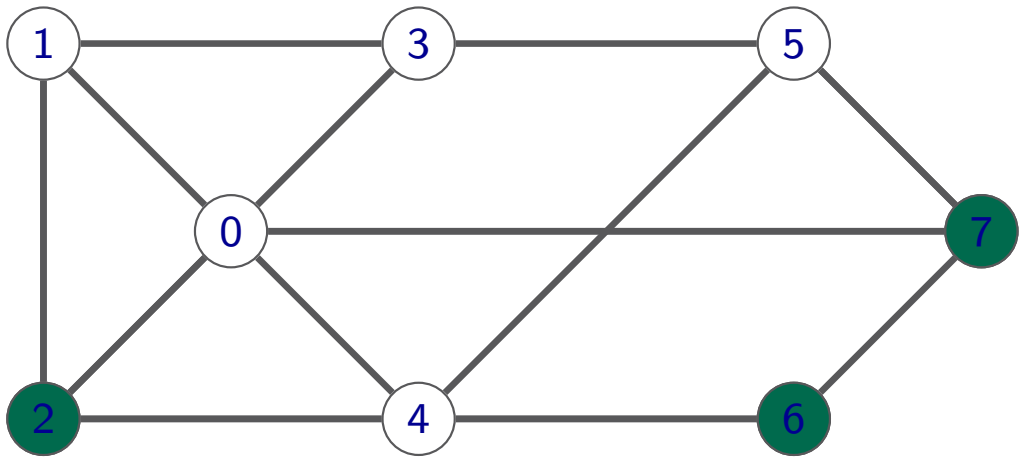
Die Kosten einer Teilmenge $T \subseteq E$ ist die Summe der Kantenkosten, $c(T) = \sum_{e \in T} c(e)$.

Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge.

Beispiel: $S = \{2, 6, 7\}$

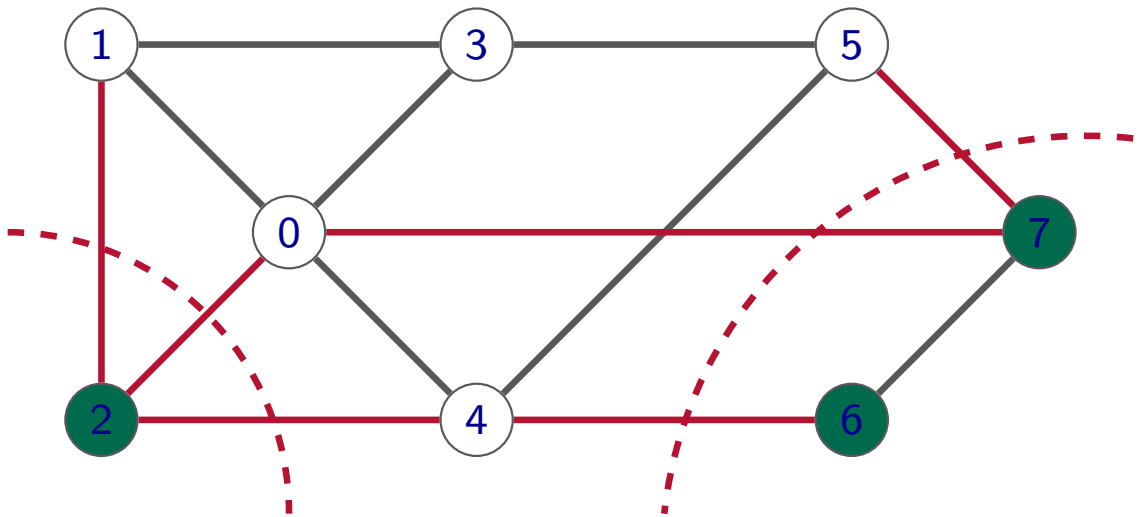


Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge. Der Schnitt von S ist die Kantenmenge $\delta(S) \subseteq E$, die genau einen Endknoten in S besitzen.

Beispiel: $S = \{2, 6, 7\}$, Schnitt $\delta(S) =$ rote Kanten

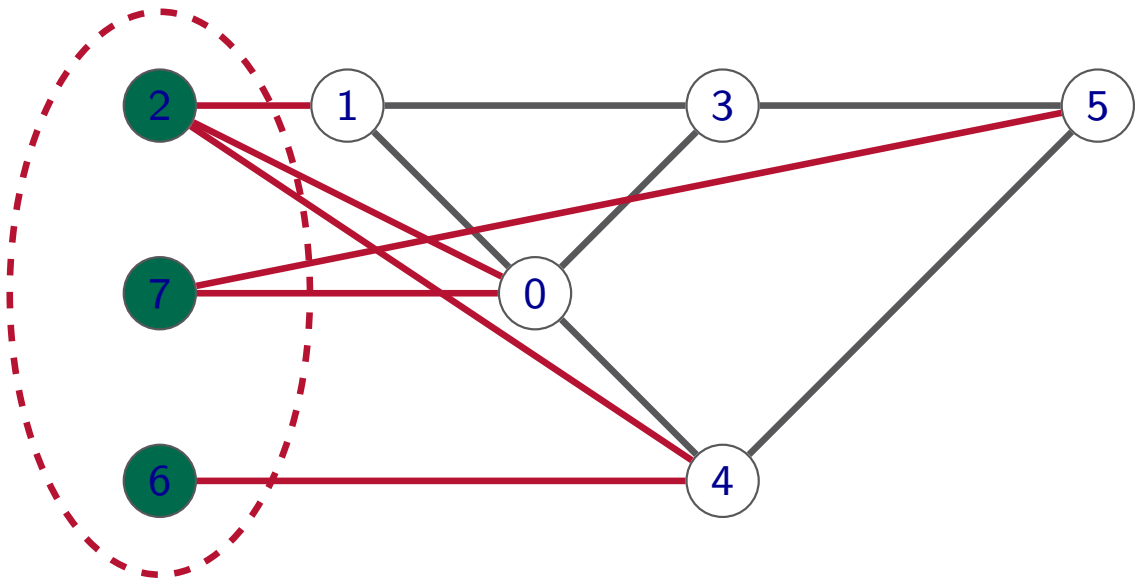


Schnitte in Graphen

Definition

Sei $G = (V, E)$ ein ungerichteter Graph und $S \subseteq V$ eine Knotenmenge. Der Schnitt von S ist die Kantenmenge $\delta(S) \subseteq E$, die genau einen Endknoten in S besitzen.

Beispiel: $S = \{2, 6, 7\}$, Schnitt $\delta(S) =$ rote Kanten

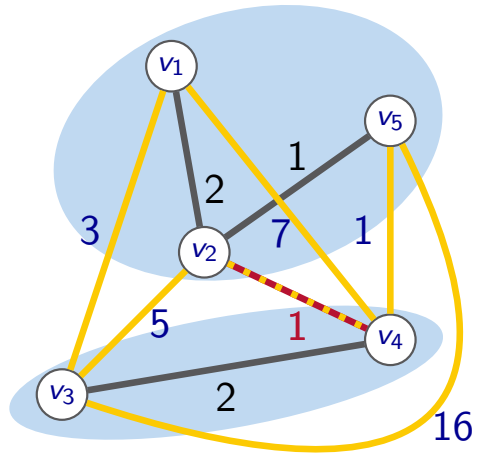


Schnitteigenschaft

Sei $G = (V, E, c)$ ein einfacher zusammenhängender Graph.

Satz (Schnitteigenschaft)

Sei $S \subset V$ und e eine Kante im Schnitt $\delta(S)$ mit minimalen Kantenkosten $c(e)$.
(1) Dann existiert ein MST, der e enthält.
(2) Sei T' eine Kantenmenge, die in einem MST enthalten ist und keine Kante aus $\delta(S)$ enthält. Dann existiert ein MST der $T' \cup \{e\}$ enthält.



Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Mehrere Optionen einen Schnitt auszuwählen (Prim, Kruskal).

Greedy Algorithmen

Greedy: Triff in jedem Schritt eine lokal optimale Entscheidung.

Schnitteigenschaft

- Beginne mit leerer Kantenmenge T.
- Solange T nicht zusammenhängend in G: Wähle einen Schnitt, der keine Kante aus T enthält, und füge eine Schnittkante mit minimalen Kosten zu T hinzu.

Mehrere Optionen einen Schnitt auszuwählen (Prim, Kruskal).

Kreiseigenschaft

- Beginne mit Kantenmenge $T = E$.
- Solange T nicht kreisfrei: Wähle Kreis in T und entferne kostenmaximale Kante aus dem Kreis.

Keine effiziente Implementierung bekannt.

Algorithmus von Prim (1957)

Algorithmus von Prim

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

```
1 Setze T = ∅.                    (T enthält zukünftige Baumkanten)
2 Für beliebigen Startknoten s setze S = {s}.
3 while |T| < |V| − 1 do
4 |   Wähle kostenminimale Kante e aus dem Schnitt von S.
5 |   Füge e zu T hinzu.
6 |   S ← S ∪ e
7 return T
```

Satz(Einfache Analyse; letzte Vorlesung)

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}(n \cdot m)$.

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n+m)\log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n-1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n + m) \log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n - 1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an
- jede Kante wird höchstens einmal zum Binary Heap hinzugefügt und höchstens einmal aus dem Binary Heap entfernt

Algorithmus von Prim - Verbesserte Analyse

Satz

Sei $G = (V, E, c)$ zusammenhängend. Prim's Algorithmus liefert einen MST mit einer Laufzeit von $\mathcal{O}((n+m)\log n)$.

Wir analysieren die Laufzeit erneut, wobei wir den Schnitt von S als Binary Heap verwalten:

- die While-Schleife wird $(n-1)$ mal durchlaufen
- wir initialisieren leeren Heap mit einem Array der Länge m in Zeit $\mathcal{O}(m)$
- eine kostenminimale Kante kann nun in Zeit $\mathcal{O}(1)$ gefunden werden
- nach jedem Update $S \leftarrow S \cup e$ passen wir den Binary Heap an
- jede Kante wird höchstens einmal zum Binary Heap hinzugefügt und höchstens einmal aus dem Binary Heap entfernt

  $\rightarrow \mathcal{O}(n + m + m\log m) = \mathcal{O}((n+m)\log(n^2)) = \mathcal{O}((n+m)\log(n))$

Algorithmus von Kruskal

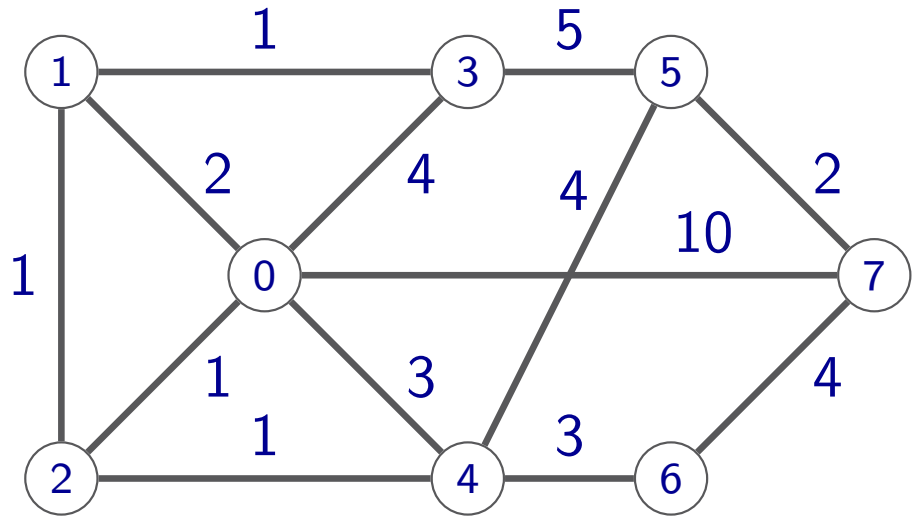
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



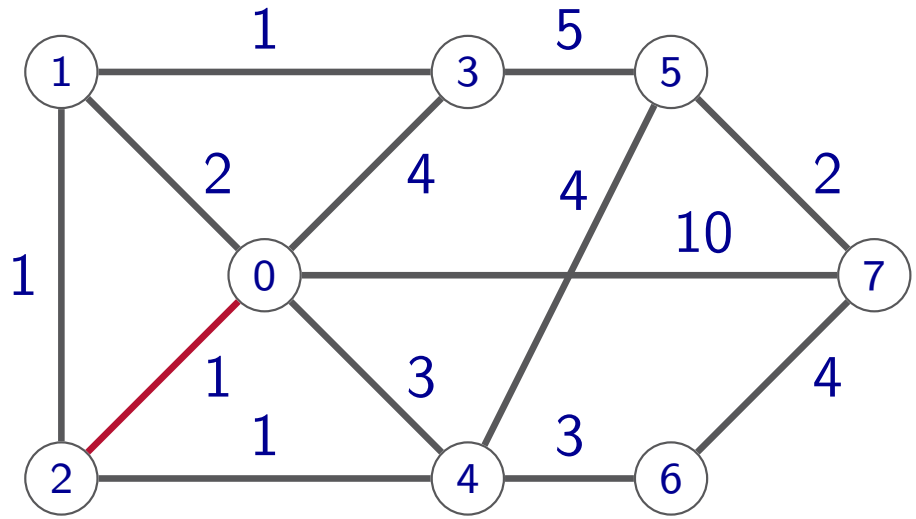
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



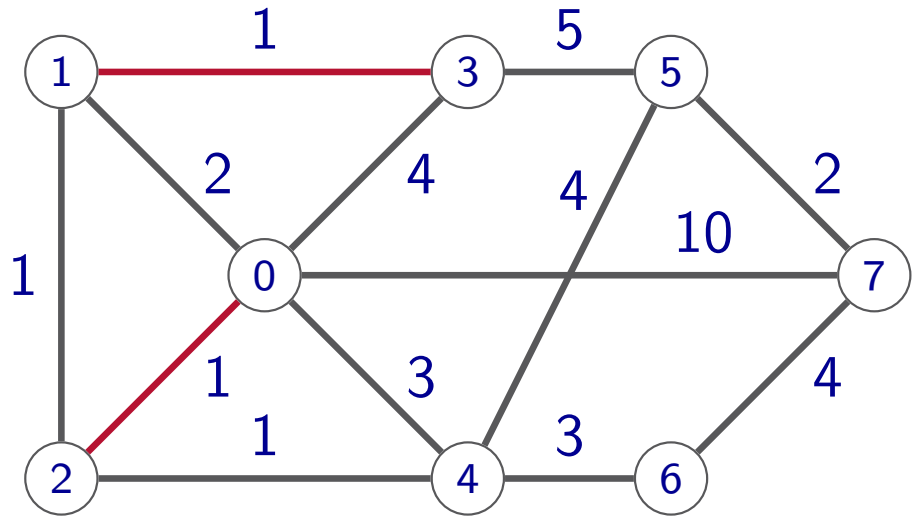
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



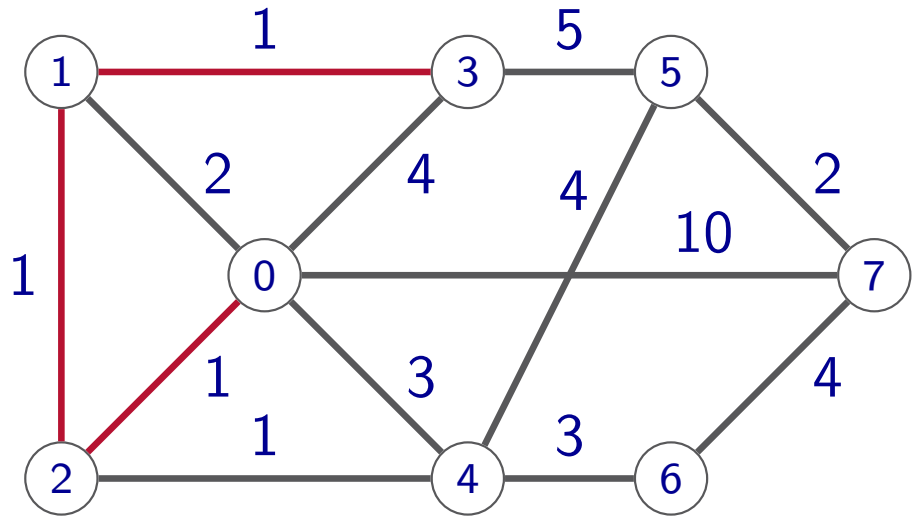
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



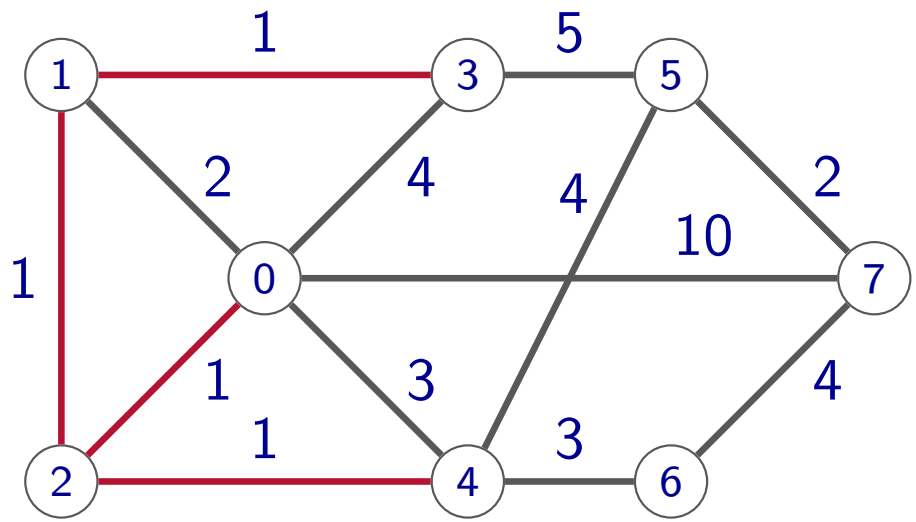
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



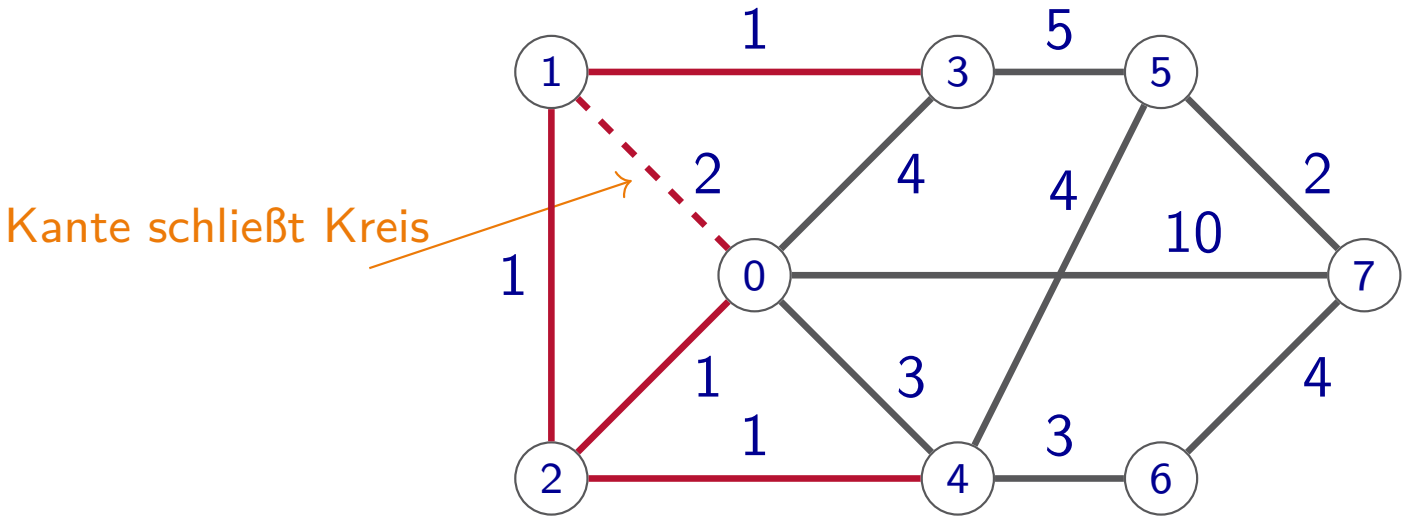
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



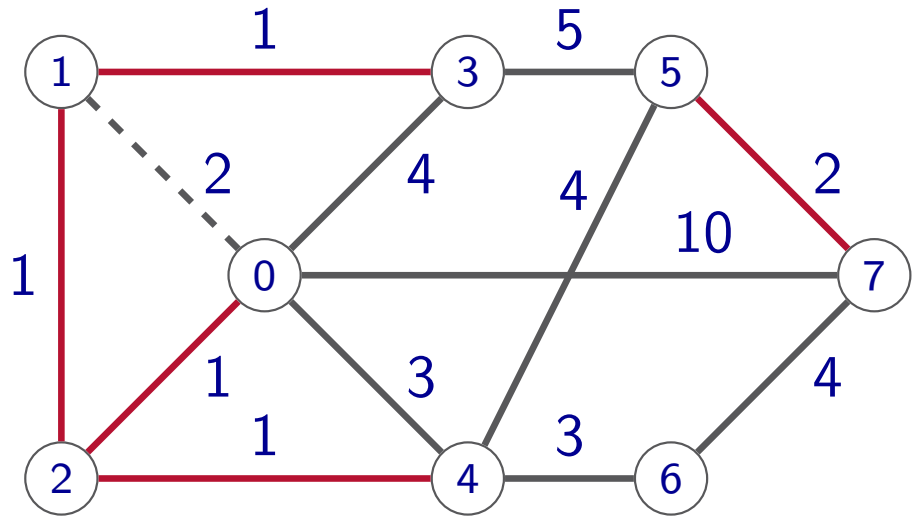
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



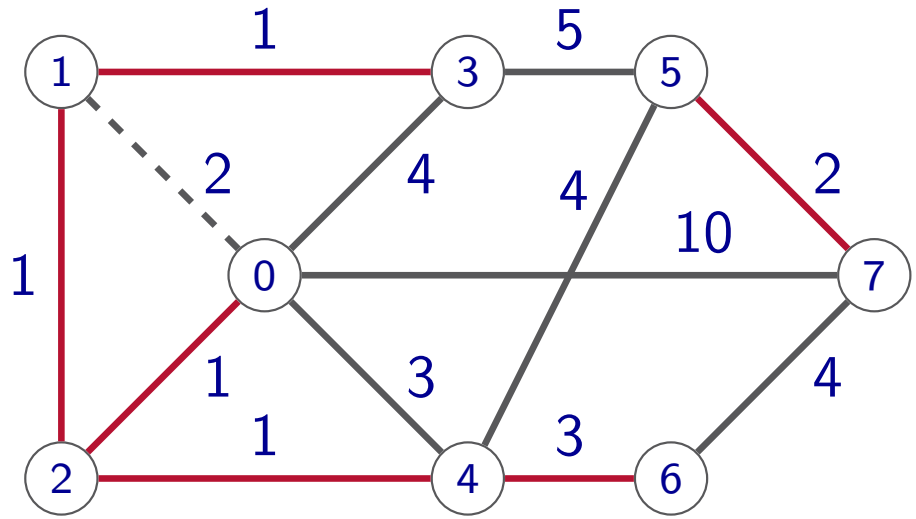
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



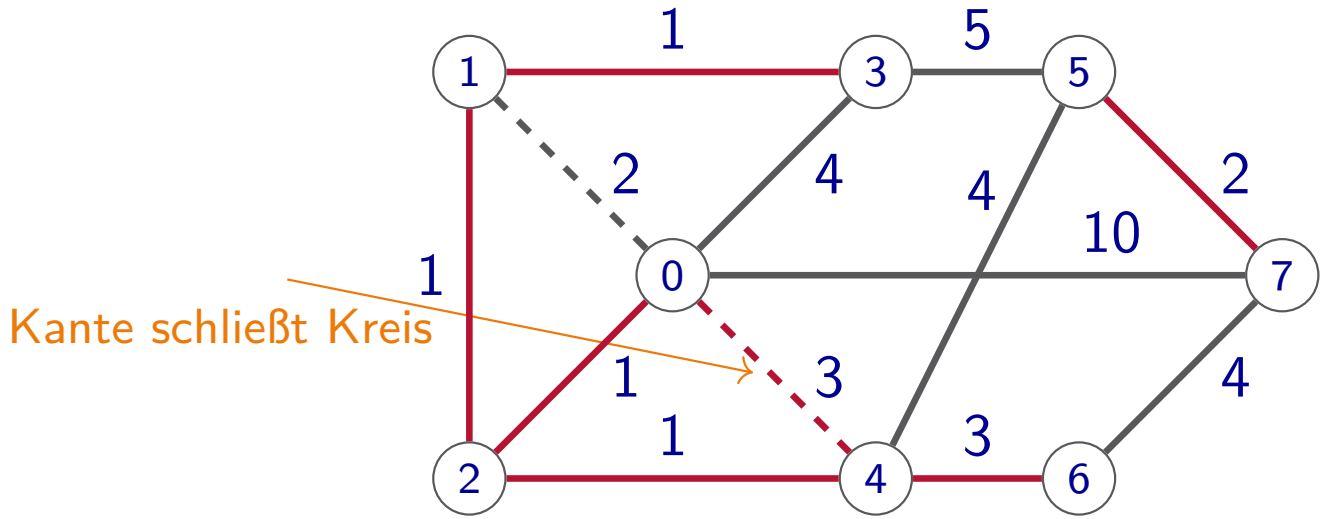
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



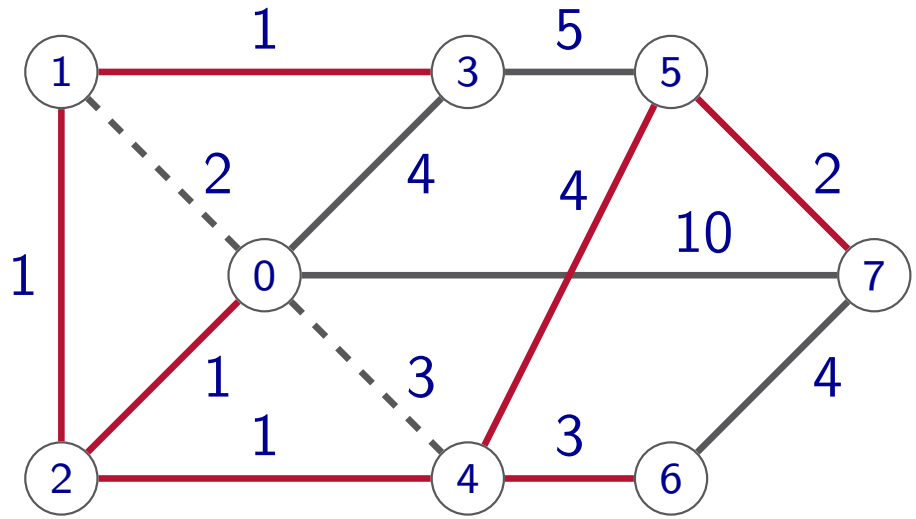
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \cdot n)$.

Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \cdot n)$.

Beweis.

- Laufzeit:
  - Sortieren der Kanten in $\mathcal{O}(m \log m)$.

Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \cdot n)$.

Beweis.

- Laufzeit:
  - Sortieren der Kanten in $\mathcal{O}(m \log m)$.
  - m mal Kreistest in Graph mit $\leq n$ Kanten in $\mathcal{O}(n)$ (BFS/DFS)

Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \cdot n)$.

Beweis.

- Laufzeit:
  - Sortieren der Kanten in $\mathcal{O}(m \log m)$.
  - m mal Kreistest in Graph mit $\leq n$ Kanten in $\mathcal{O}(n)$ (BFS/DFS)

    $\Rightarrow$ Laufzeit: $\mathcal{O}(m \log m) + \mathcal{O}(m \cdot n) = \mathcal{O}(m \cdot n)$

Analyse von Kruskal's Algorithmus

- Korrektheit:
  - In jeder Iteration wählt Kruskals Alg. eine kostenminimale Kante e, die mit T keinen Kreis schließt.

Analyse von Kruskal's Algorithmus

- Korrektheit:
  - In jeder Iteration wählt Kruskals Alg. eine kostenminimale Kante e, die mit T keinen Kreis schließt.
  - Diese Kante e ist im Schnitt einer der Zusammenhangskomponenten des Graphen (V, T) und erfüllt damit die Schnitteigenschaft.

Analyse von Kruskal's Algorithmus

- Korrektheit:
  - In jeder Iteration wählt Kruskals Alg. eine kostenminimale Kante e, die mit T keinen Kreis schließt.
  - Diese Kante e ist im Schnitt einer der Zusammenhangskomponenten des Graphen (V, T) und erfüllt damit die Schnitteigenschaft.
  - Algorithmus terminiert mit einem aufspannenden Baum.
    $\Rightarrow$ kostenminimal, wegen erfüllter Schnitteigenschaft $\square$

Kruskal's Algorithmus - Verbesserte Analyse

Kann die Laufzeit von Kruskal's Algorithmus ebenso verbessert werden?

Kruskal's Algorithmus - Verbesserte Analyse

Kann die Laufzeit von Kruskal's Algorithmus ebenso verbessert werden?

Idee:

- Verwalte Zusammenhangskomponenten von T

Kruskal's Algorithmus - Verbesserte Analyse

Kann die Laufzeit von Kruskal's Algorithmus ebenso verbessert werden?

Idee:

- Verwalte Zusammenhangskomponenten von T
- Wenn die Endpunkte einer Kante $\{v, w\}$ in verschiedenen Komponenten liegen, können wir sie hinzufügen, sonst bildet sich ein Kreis

Kruskal's Algorithmus - Verbesserte Analyse

Kann die Laufzeit von Kruskal's Algorithmus ebenso verbessert werden?

Idee:

- Verwalte Zusammenhangskomponenten von T
- Wenn die Endpunkte einer Kante $\{v, w\}$ in verschiedenen Komponenten liegen, können wir sie hinzufügen, sonst bildet sich ein Kreis
- Bei Hinzufügen einer Kante $\{v, w\}$ zu T vereinigen wir die Komponenten von v und w

Kruskal's Algorithmus - Verbesserte Analyse

Kann die Laufzeit von Kruskal's Algorithmus ebenso verbessert werden?

Idee:

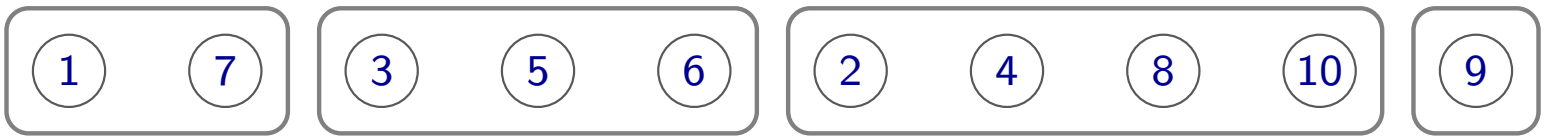
- Verwalte Zusammenhangskomponenten von T
- Wenn die Endpunkte einer Kante $\{v, w\}$ in verschiedenen Komponenten liegen, können wir sie hinzufügen, sonst bildet sich ein Kreis
- Bei Hinzufügen einer Kante $\{v, w\}$ zu T vereinigen wir die Komponenten von v und w

  $\rightarrow$ Union-Find Datenstruktur

Union-Find Datenstruktur

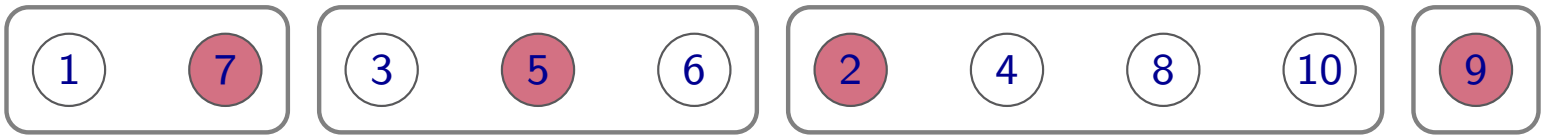
Union-Find Datenstruktur

Eine Union-Find Datenstruktur verwaltet eine Partition $\mathcal{P}$ einer festen, endlichen Menge M.



Union-Find Datenstruktur

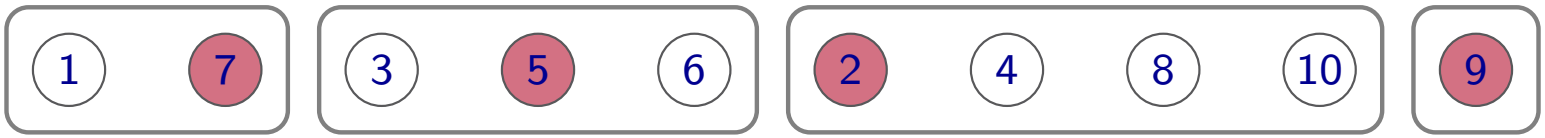
Eine Union-Find Datenstruktur verwaltet eine Partition $\mathcal{P}$ einer festen, endlichen Menge M.



- Jede Klasse $B \in \mathcal{P}$ hat einen Repräsentaten (kann die Datenstruktur wählen)

Union-Find Datenstruktur

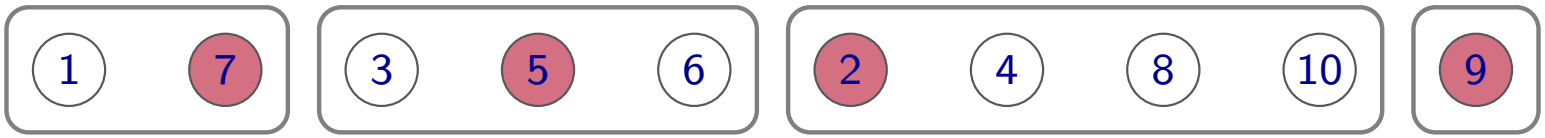
Eine Union-Find Datenstruktur verwaltet eine Partition $\mathcal{P}$ einer festen, endlichen Menge M.



- Jede Klasse $B \in \mathcal{P}$ hat einen Repräsentaten (kann die Datenstruktur wählen)
- Operationen:
  - `Find`: Gegeben ein $i \in M$, gib den Repräsentaten der Klasse von i zurück (z.B. $\texttt{Find}(3) = 5$)
  - `Union`: Gegeben $i, j \in M$, vereinige die Klassen von i und j

Union-Find Datenstruktur

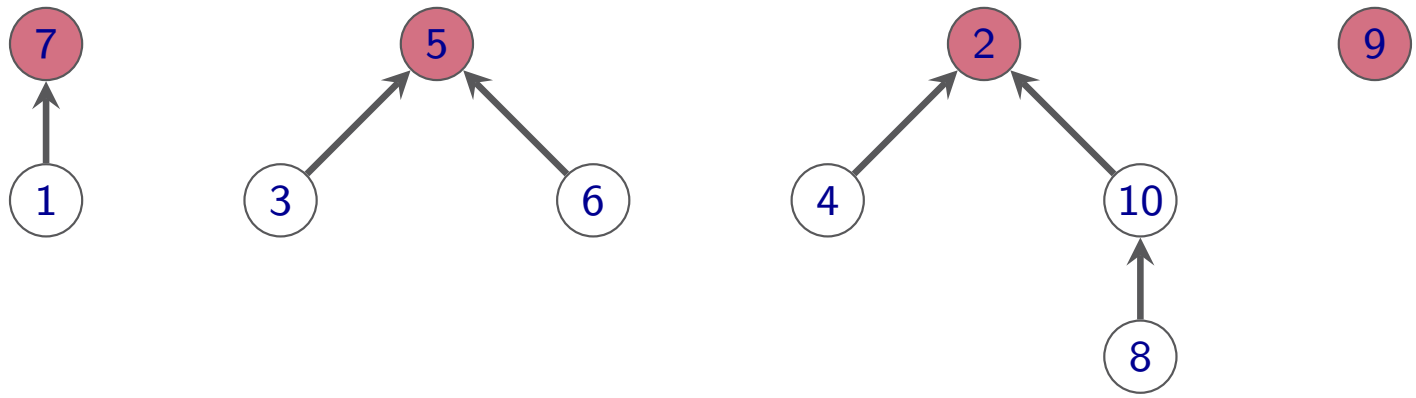
Eine Union-Find Datenstruktur verwaltet eine Partition $\mathcal{P}$ einer festen, endlichen Menge M.



- Jede Klasse $B \in \mathcal{P}$ hat einen Repräsentaten (kann die Datenstruktur wählen)
- Operationen:
  - `Find`: Gegeben ein $i \in M$, gib den Repräsentaten der Klasse von i zurück (z.B. $\texttt{Find}(3) = 5$)
  - `Union`: Gegeben $i, j \in M$, vereinige die Klassen von i und j
- Bei Initialisierung haben alle Klassen Größe 1

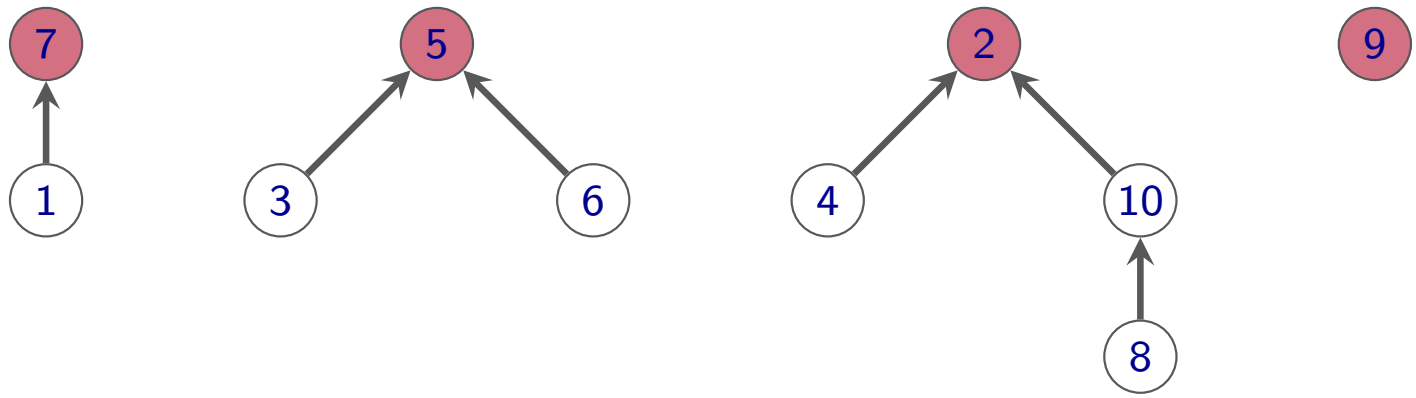
Union-Find Datenstruktur - Umsetzung

Idee: Wir stellen jede Klasse als Baum dar, wobei der Repräsentat in der Wurzel liegt



Union-Find Datenstruktur - Umsetzung

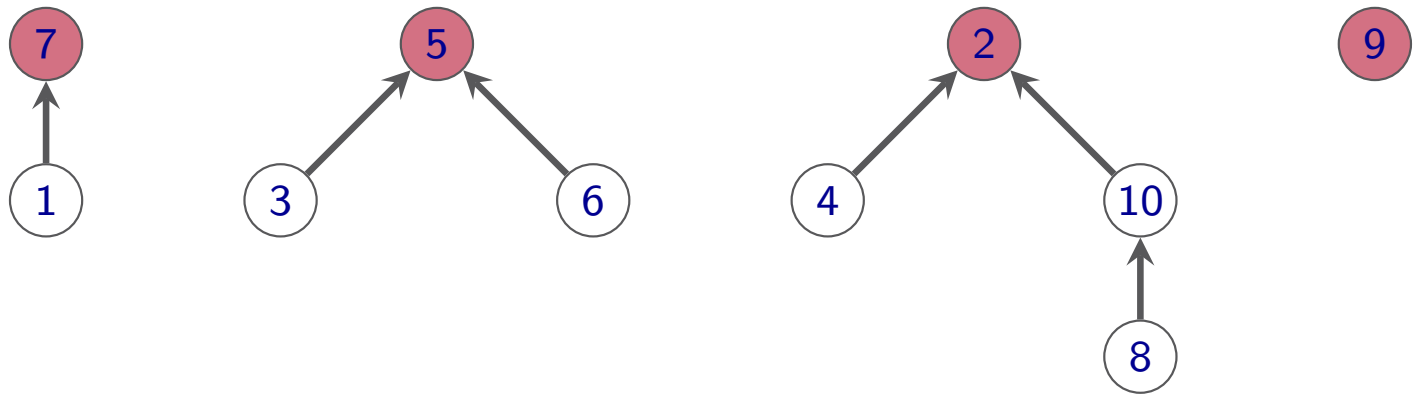
Idee: Wir stellen jede Klasse als Baum dar, wobei der Repräsentat in der Wurzel liegt



- Jedes Element i speichert seinen Vorgänger als $\text{parent}(i)$ (z.B. $\text{parent}(8) = 10$)

Union-Find Datenstruktur - Umsetzung

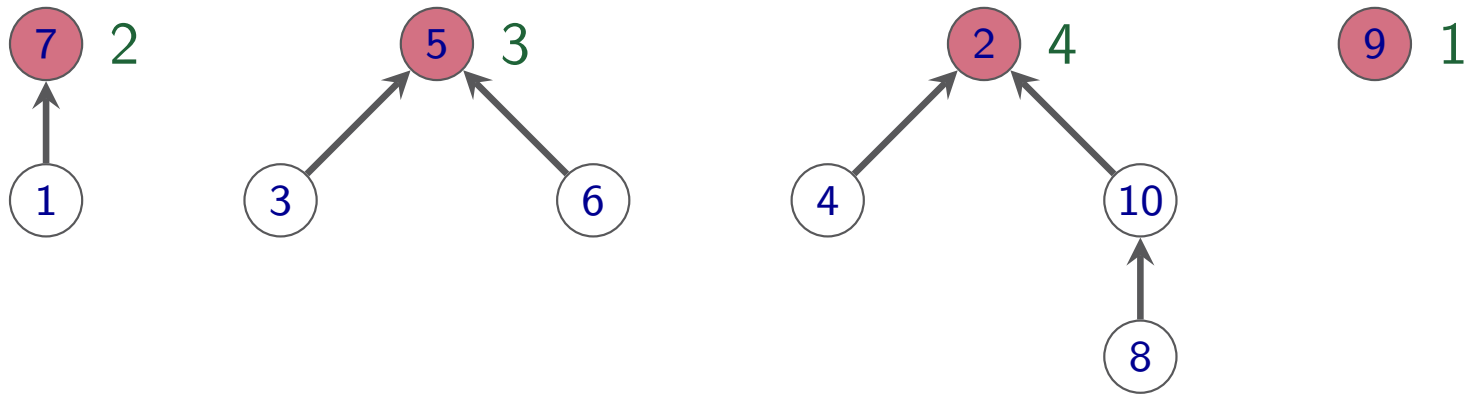
Idee: Wir stellen jede Klasse als Baum dar, wobei der Repräsentat in der Wurzel liegt



- Jedes Element i speichert seinen Vorgänger als $\text{parent}(i)$ (z.B. $\text{parent}(8) = 10$)
- Für Wurzel-Knoten i (d.h., Repräsentaten) setzen wir $\text{parent}(i) = i$

Union-Find Datenstruktur - Umsetzung

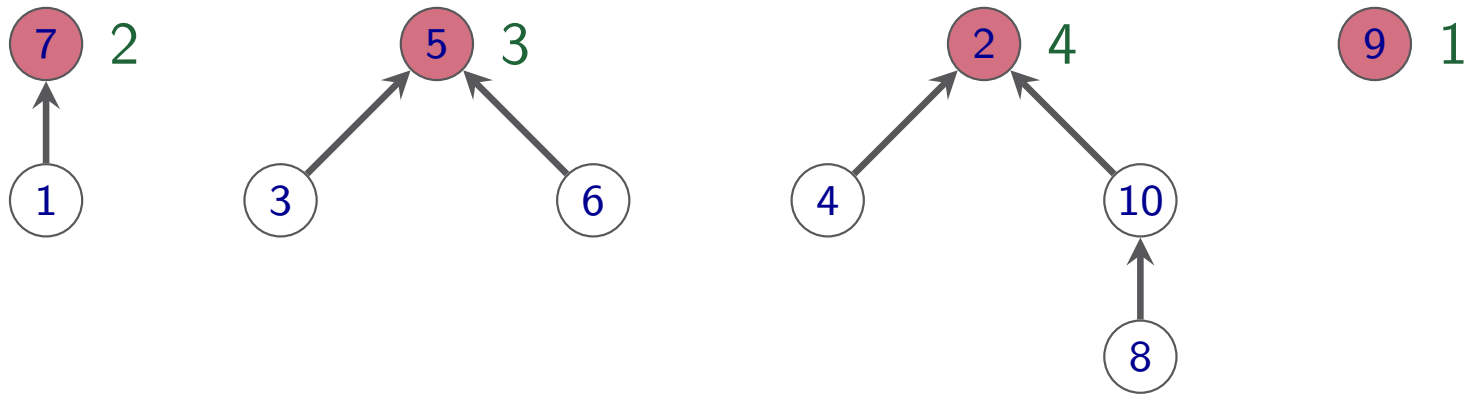
Idee: Wir stellen jede Klasse als Baum dar, wobei der Repräsentat in der Wurzel liegt



- Jedes Element i speichert seinen Vorgänger als $\text{parent}(i)$ (z.B. $\text{parent}(8) = 10$)
- Für Wurzel-Knoten i (d.h., Repräsentaten) setzen wir $\text{parent}(i) = i$
- Jede Wurzel i speichert die Anzahl der Elemente in der Klasse

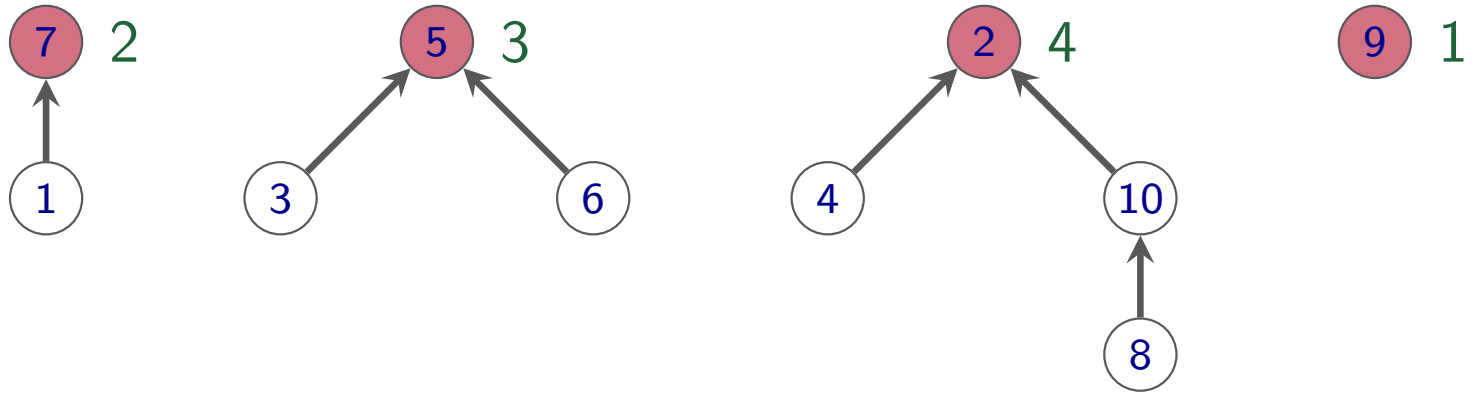
Union-Find Datenstruktur - Umsetzung

Idee: Wir stellen jede Klasse als Baum dar, wobei der Repräsentat in der Wurzel liegt

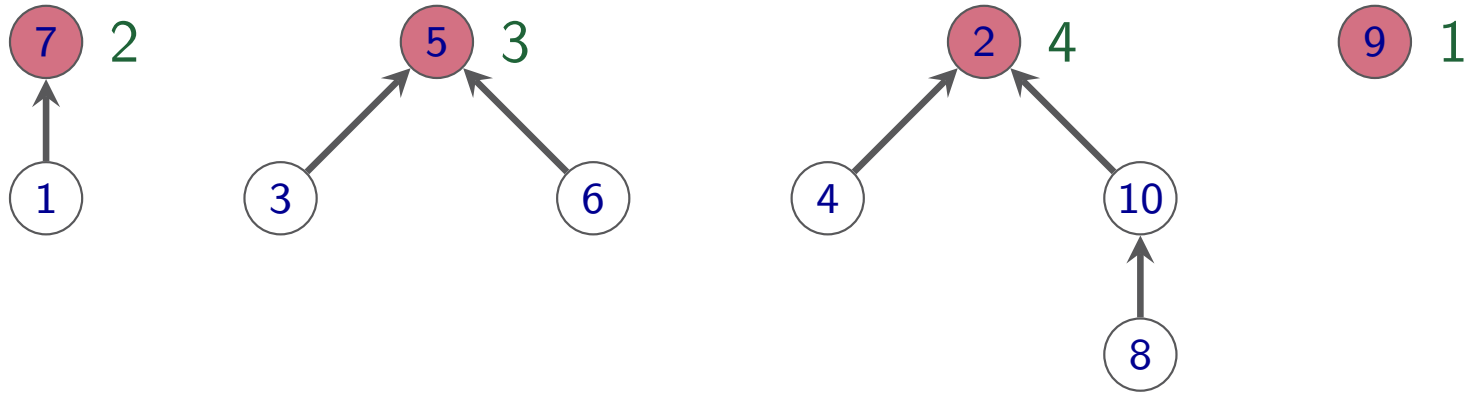


- Jedes Element i speichert seinen Vorgänger als $\text{parent}(i)$ (z.B. $\text{parent}(8) = 10$)
- Für Wurzel-Knoten i (d.h., Repräsentaten) setzen wir $\text{parent}(i) = i$
- Jede Wurzel i speichert die Anzahl der Elemente in der Klasse
- Eine Komponente mit k Elementen hat Tiefe höchstens $\lceil \log_2 k \rceil$

Union-Find Datenstruktur - Operation `Find`



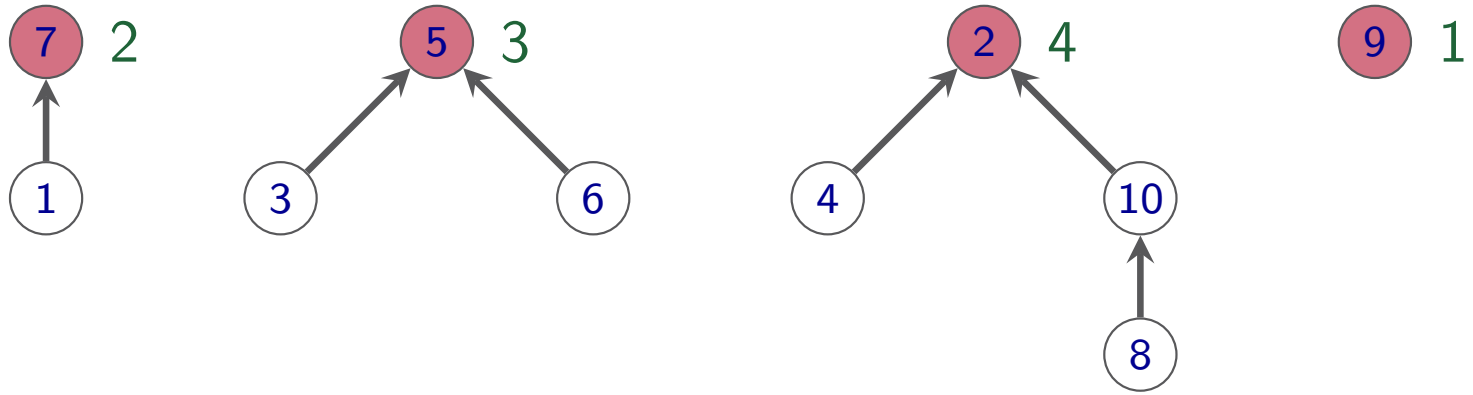
Union-Find Datenstruktur - Operation `Find`



`Find`(i)

```
1 while i ≠ parent(i) do
2     i := parent(i)
3 return i
```

Union-Find Datenstruktur - Operation `Find`

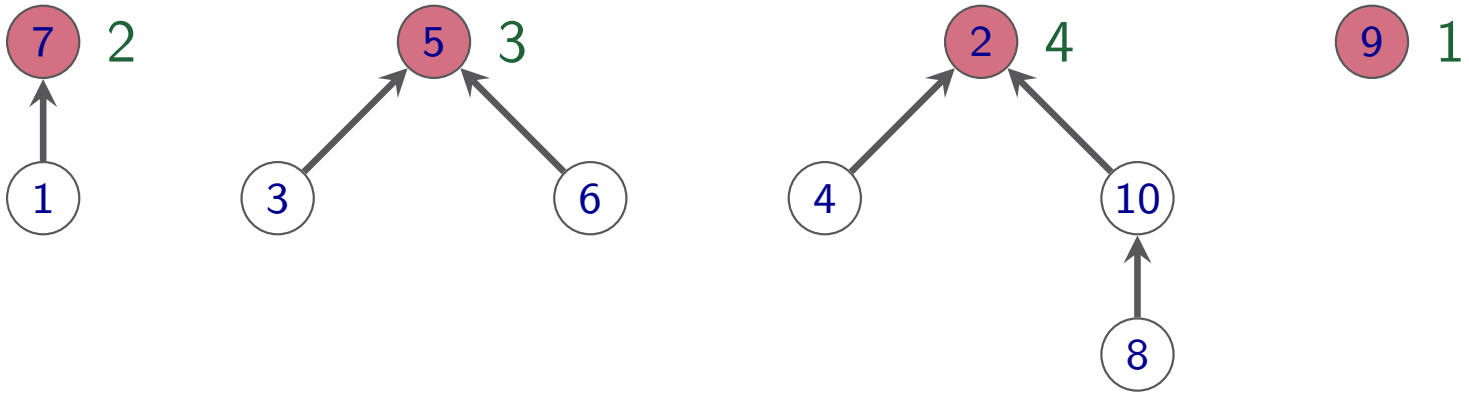


`Find`(i)

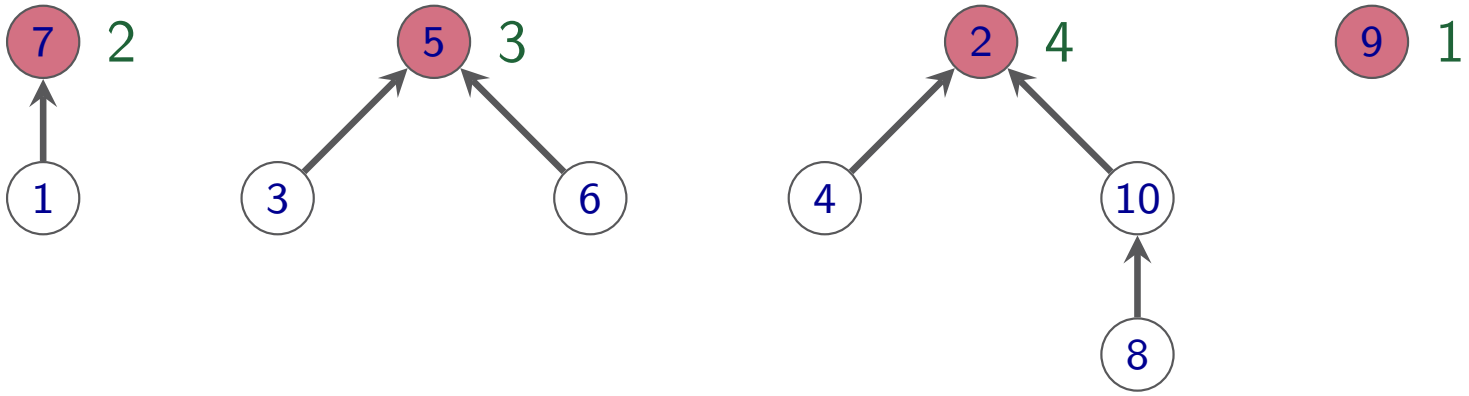
```
1 while i ≠ parent(i) do
2     i := parent(i)
3 return i
```

Laufzeit: $\mathcal{O}(\log n)$, da jeder Baum Tiefe höchstens $\lceil \log_2 n \rceil$ hat

Union-Find Datenstruktur - Operation `Union`



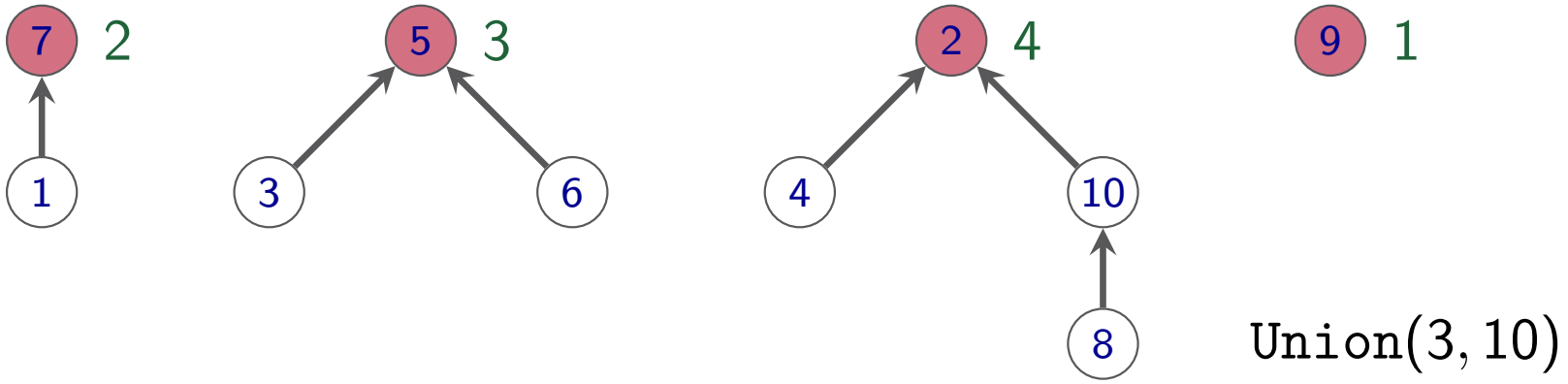
Union-Find Datenstruktur - Operation Union



Union(i, j)

```
1 i := Find(i); j := Find(j)
2 if size(i) ≥ size(j) then
3     parent(j) := i
4     size(i) := size(i) + size(j)
5 else
6     parent(i) := j
7     size(j) := size(i) + size(j)
```

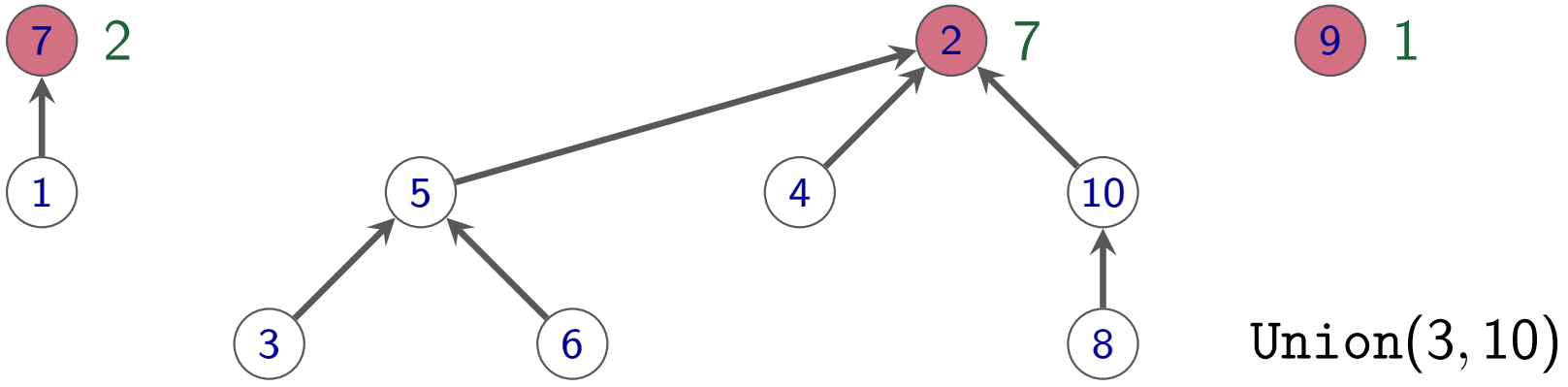
Union-Find Datenstruktur - Operation Union



Union(i, j)

```
1 i := Find(i); j := Find(j)
2 if size(i) ≥ size(j) then
3     parent(j) := i
4     size(i) := size(i) + size(j)
5 else
6     parent(i) := j
7     size(j) := size(i) + size(j)
```

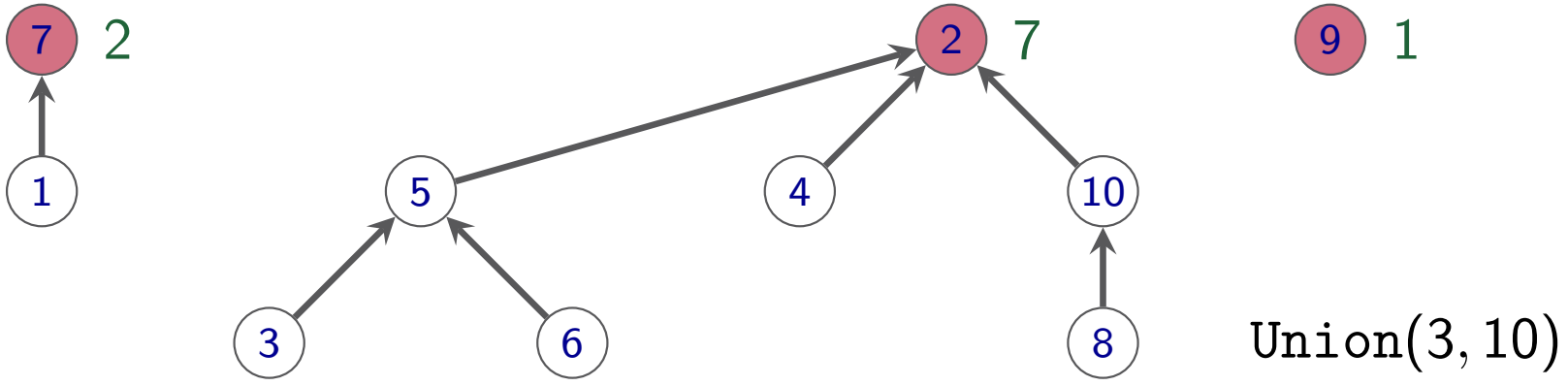
Union-Find Datenstruktur - Operation Union



Union(i, j)

```
1 i := Find(i); j := Find(j)
2 if size(i) ≥ size(j) then
3 |   parent(j) := i
4 |   size(i) := size(i) + size(j)
5 else
6 |   parent(i) := j
7 |   size(j) := size(i) + size(j)
```

Union-Find Datenstruktur - Operation Union

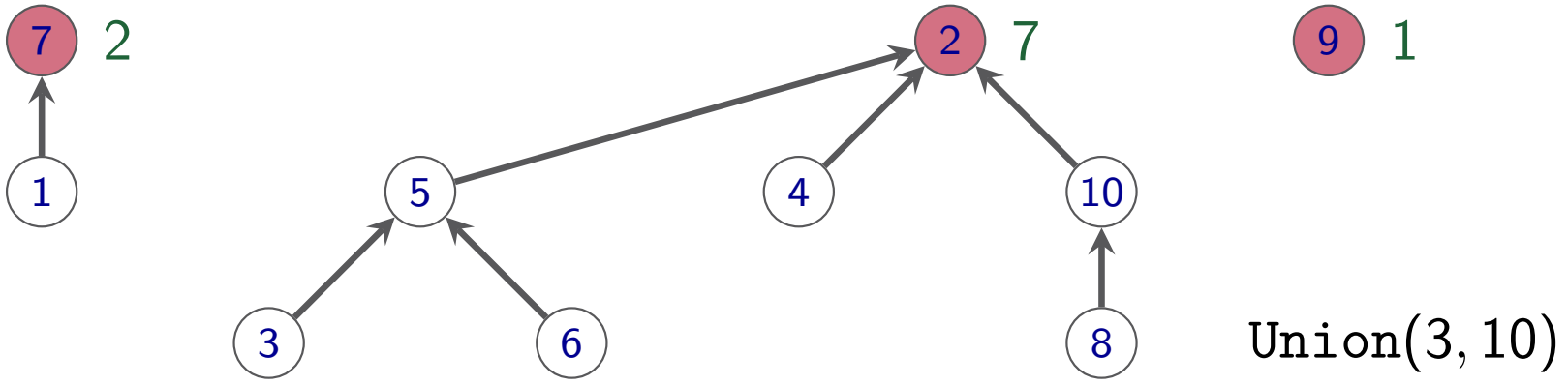


Union(i, j)

Laufzeit: $\mathcal{O}(\log n)$

```
1 i := Find(i); j := Find(j)
2 if size(i) ≥ size(j) then
3     parent(j) := i
4     size(i) := size(i) + size(j)
5 else
6     parent(i) := j
7     size(j) := size(i) + size(j)
```

Union-Find Datenstruktur - Operation Union



Union(i, j)

```
1 i := Find(i); j := Find(j)
2 if size(i) ≥ size(j) then
3 |   parent(j) := i
4 |   size(i) := size(i) + size(j)
5 else
6 |   parent(i) := j
7 |   size(j) := size(i) + size(j)
```

Laufzeit: $\mathcal{O}(\log n)$

Korrektheit:

- Sei d die Tiefe und s die Größe des Baumes, der i und j enthält (nach Union(i, j))
- Wir müssen zeigen, dass $d \leq \lceil \log_2 s \rceil$ gilt

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)
- Wir haben $s = s_i + s_j$

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)
- Wir haben $s = s_i + s_j$
- Außerdem gilt $d_i \leq \lceil \log(s_i) \rceil$ und $d_j \leq \lceil \log(s_j) \rceil$ nach Voraussetzung

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)
- Wir haben $s = s_i + s_j$
- Außerdem gilt $d_i \leq \lceil \log(s_i) \rceil$ und $d_j \leq \lceil \log(s_j) \rceil$ nach Voraussetzung
- O.B.d.A. nehme an, dass $s_i \leq s_j$

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)
- Wir haben $s = s_i + s_j$
- Außerdem gilt $d_i \leq \lceil \log(s_i) \rceil$ und $d_j \leq \lceil \log(s_j) \rceil$ nach Voraussetzung
- O.B.d.A. nehme an, dass $s_i \leq s_j$
- Fall 1: $d_i < d_j$. Dann ist

$$d \leq d_j \leq \lceil \log(s_j) \rceil \leq \lceil \log(s) \rceil$$

Union-Find Datenstruktur - Operation `Union`

Korrektheit (Fortgesetzt):

- Sei d_k die Tiefe und s_k die Größe des Baumes, der k enthält, bevor `Union`(i, j)
- Wir haben $s = s_i + s_j$
- Außerdem gilt $d_i \leq \lceil \log(s_i) \rceil$ und $d_j \leq \lceil \log(s_j) \rceil$ nach Voraussetzung
- O.B.d.A. nehme an, dass $s_i \leq s_j$
- Fall 1: $d_i < d_j$. Dann ist

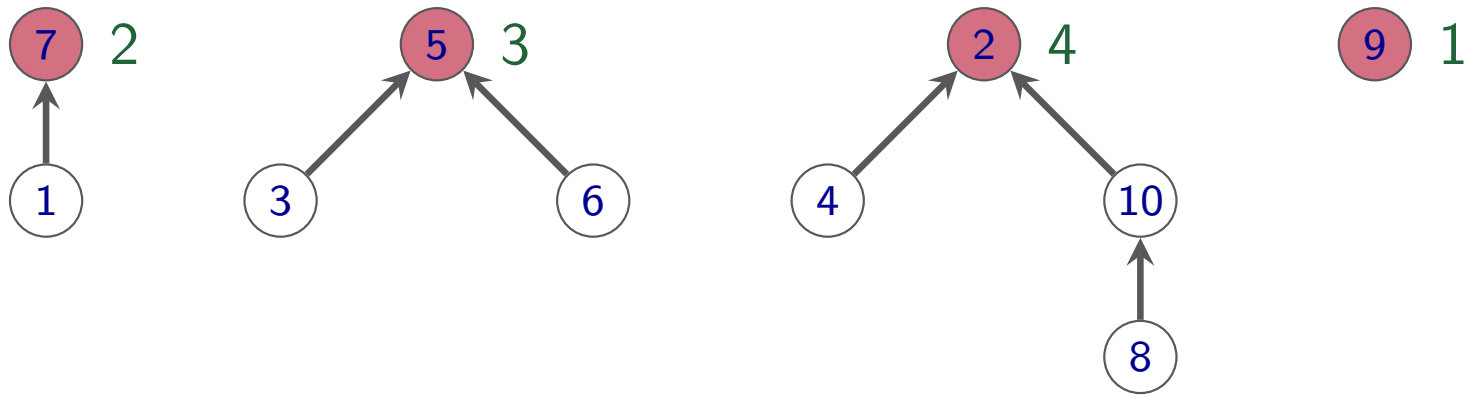
$$d \leq d_j \leq \lceil \log(s_j) \rceil \leq \lceil \log(s) \rceil$$

- Fall 2: $d_i \geq d_j$. Dann ist

$$d \leq 1 + d_i \leq 1 + \lceil \log(s_i) \rceil = \lceil 1 + \log(s_i) \rceil = \lceil \log(2s_i) \rceil \leq \lceil \log(s) \rceil$$

Union-Find Datenstruktur - Zusammenfassung

Eine Union-Find Datenstruktur verwaltet eine Partition $\mathcal{P}$ einer festen, endlichen Menge M.



- Jede Klasse $B \in \mathcal{P}$ hat einen Repräsentaten (kann die Datenstruktur wählen)
- Operationen `Union` und `Find` in Zeit $\mathcal{O}(\log n)$
- **Anmerkung:** Performance kann weiter verbessert werden mit zusätzlichen Tricks

Algorithmus von Kruskal (Fortgesetzt)

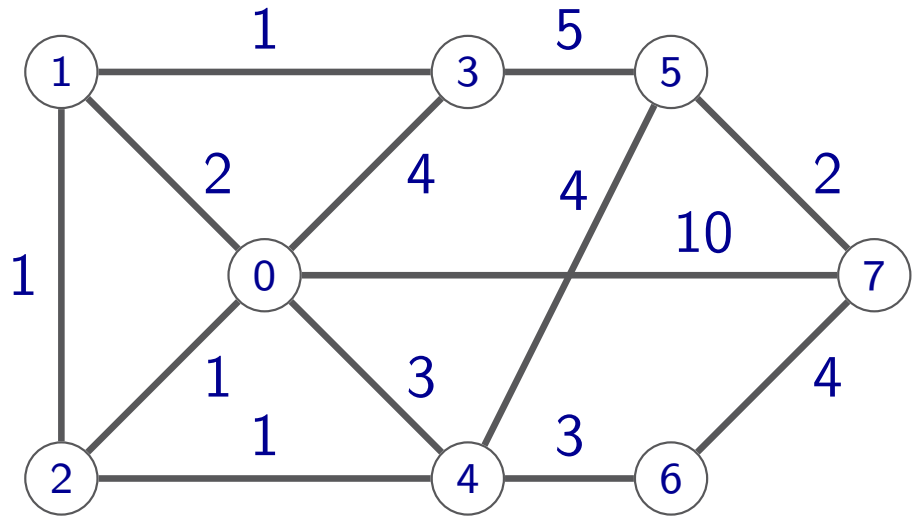
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



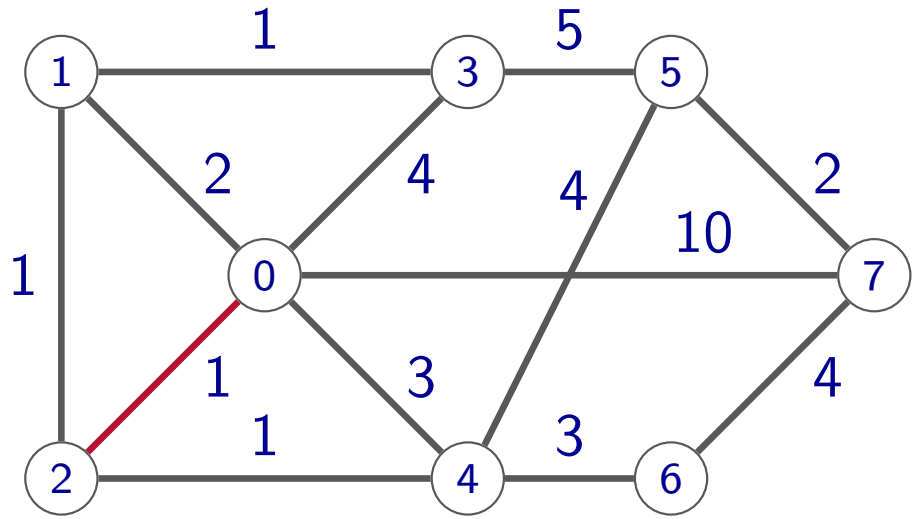
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



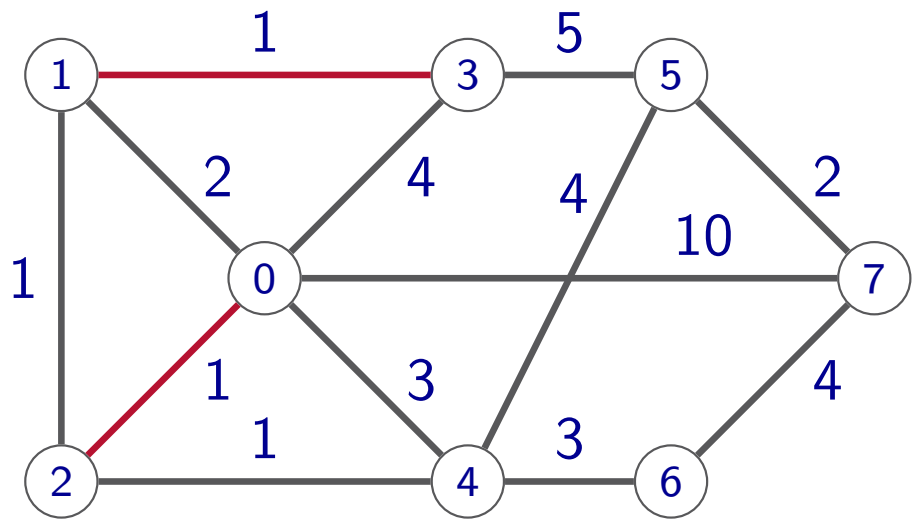
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



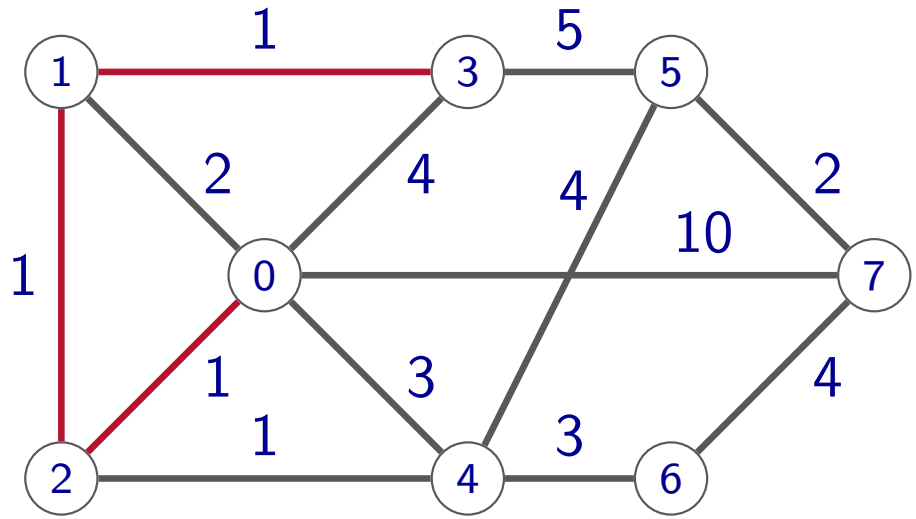
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



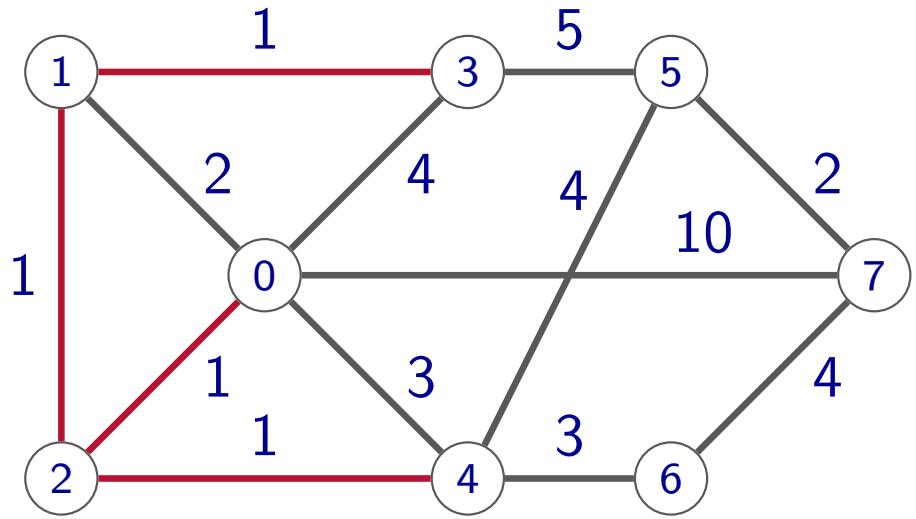
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



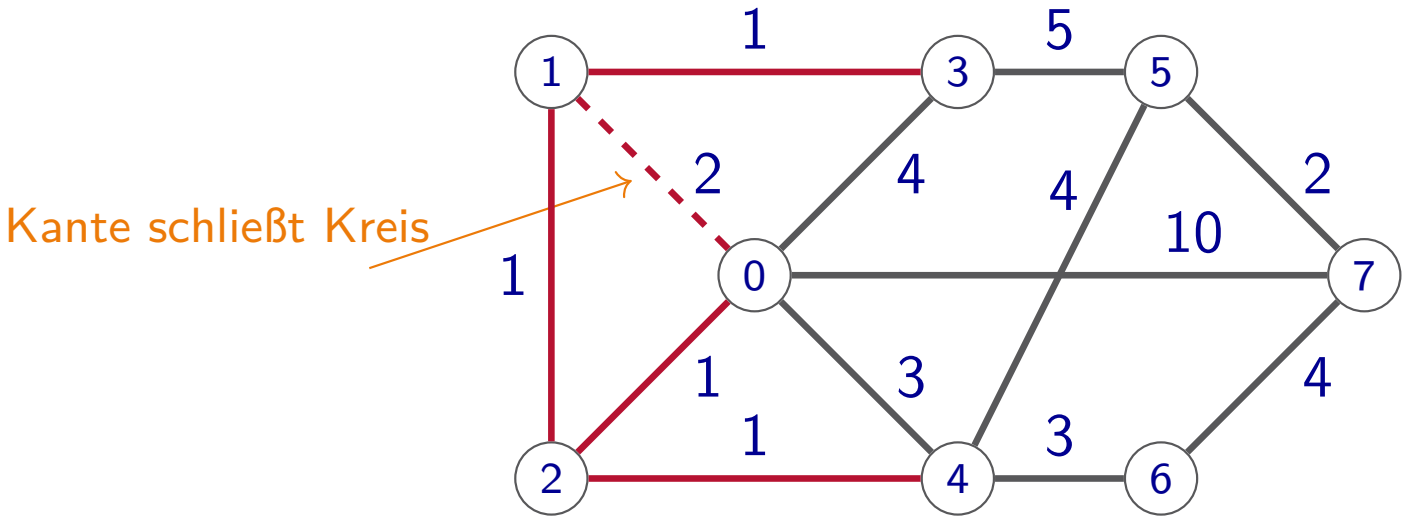
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



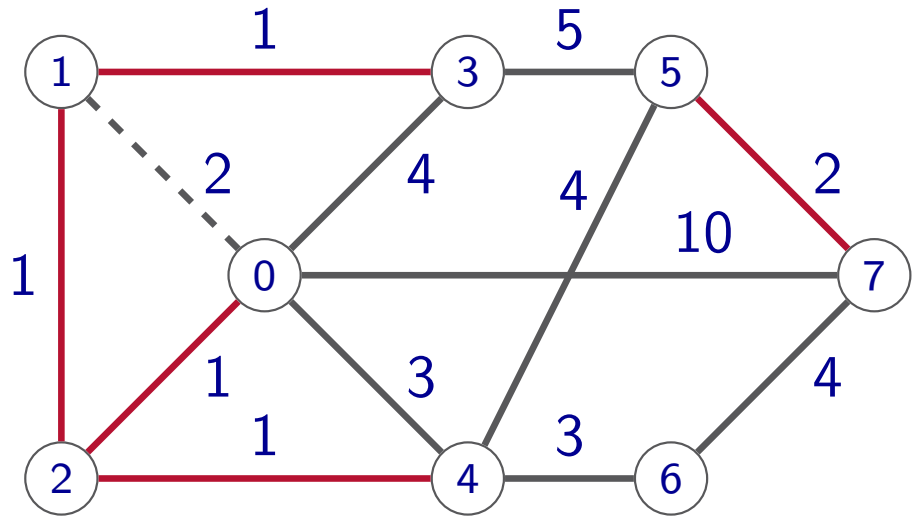
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



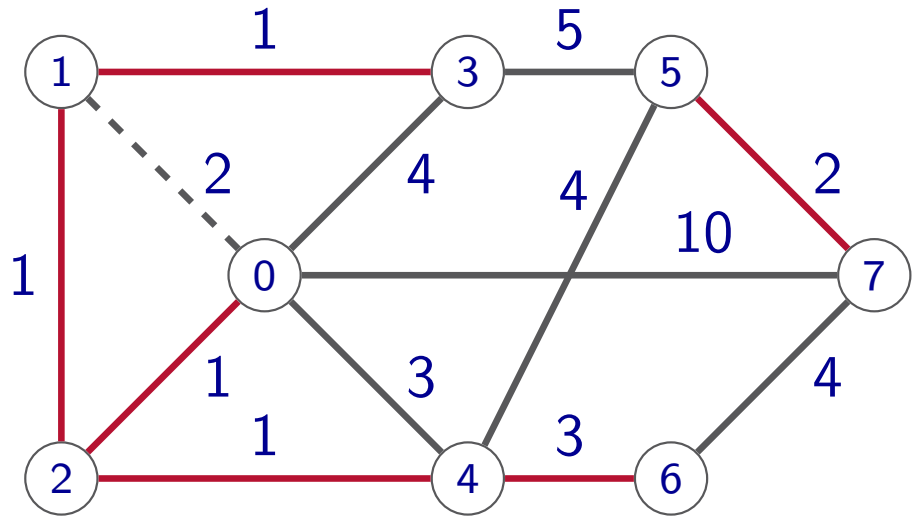
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



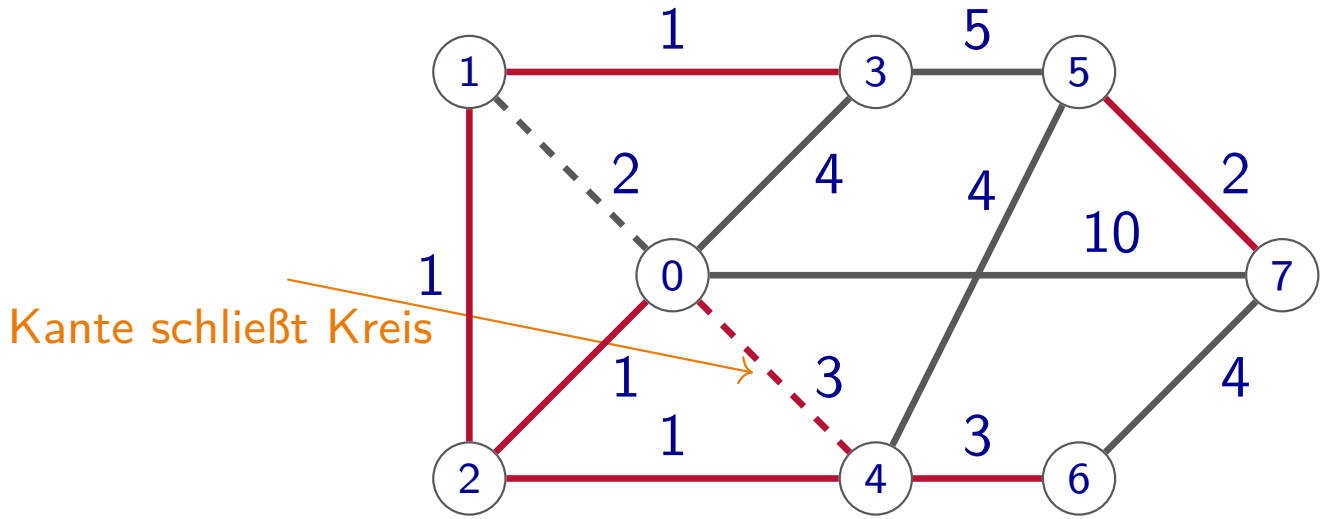
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



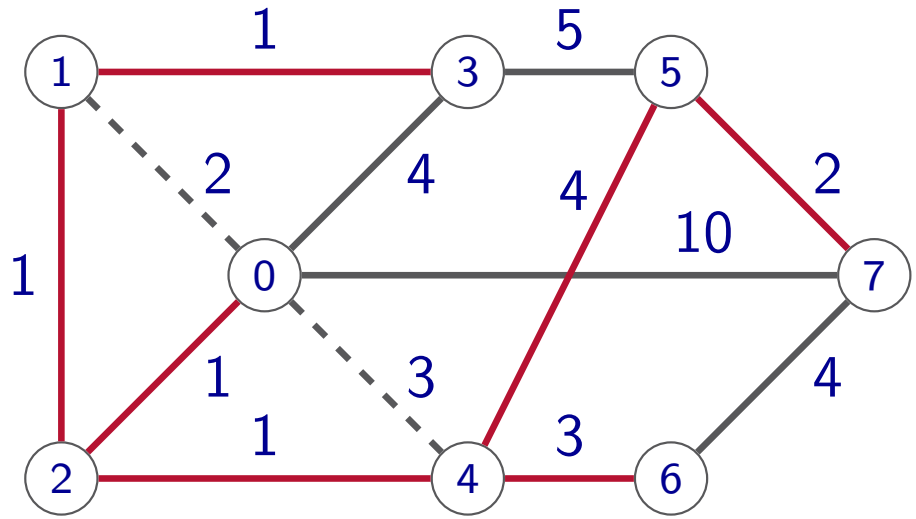
Algorithmus von Kruskal (1956)

Schnitteigenschaft wird etwas indirekter genutzt.

Algorithmus von Kruskal (G)

Input : zus.-hängender Graph $G = (V, E, c)$ mit $c(e) \geq 0$ für $e \in E$
Output : minimaler aufspannender Baum T

1 Setze $T = \emptyset$. (T enthält die zukünftigen Baumkanten)
2 Sortiere die Kanten in E, sodass $c(e_1) \leq c(e_2) \leq ... \leq c(e_m)$.
3 **for** $i = 1$ to m **do**
4 **if** $T \cup \{e_i\}$ kreisfrei **then**
5 Füge e_i zu T hinzu
6 **return** T



Verbesserte Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \log n)$.

Verbesserte Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \log n)$.

Beweis.
Wir verwalten die Zusammenhangskomponenten von T mit einer Union-Find Datenstruktur und analysieren erneut die Laufzeit:

- Sortieren der Kanten in $\mathcal{O}(m \log m)$.

Verbesserte Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \log n)$.

Beweis.
Wir verwalten die Zusammenhangskomponenten von T mit einer Union-Find Datenstruktur und analysieren erneut die Laufzeit:

- Sortieren der Kanten in $\mathcal{O}(m \log m)$.
- Jeder Kreistest in Zeit $\mathcal{O}(\log n)$ (teste ob Endpunkte der Kante in der gleichen Komponente liegen)

Verbesserte Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \log n)$.

Beweis.
Wir verwalten die Zusammenhangskomponenten von T mit einer Union-Find Datenstruktur und analysieren erneut die Laufzeit:

- Sortieren der Kanten in $\mathcal{O}(m \log m)$.
- Jeder Kreistest in Zeit $\mathcal{O}(\log n)$ (teste ob Endpunkte der Kante in der gleichen Komponente liegen)
- Wenn wir Kante $e_i = \{v_i, w_i\}$ zu T hinzufügen, vereinigen wir die Komponenten von v_i und w_i in Zeit $\mathcal{O}(\log n)$

Verbesserte Analyse von Kruskal's Algorithmus

Satz

Für einen zusammenhängenden Graphen $G = (V, E, c)$ liefert der Algorithmus von Kruskal einen MST in Laufzeit $\mathcal{O}(m \log n)$.

Beweis.
Wir verwalten die Zusammenhangskomponenten von T mit einer Union-Find Datenstruktur und analysieren erneut die Laufzeit:

- Sortieren der Kanten in $\mathcal{O}(m \log m)$.
- Jeder Kreistest in Zeit $\mathcal{O}(\log n)$ (teste ob Endpunkte der Kante in der gleichen Komponente liegen)
- Wenn wir Kante $e_i = \{v_i, w_i\}$ zu T hinzufügen, vereinigen wir die Komponenten von v_i und w_i in Zeit $\mathcal{O}(\log n)$

  $\Rightarrow$ Laufzeit: $\mathcal{O}(m \log m) + \mathcal{O}(m \cdot \log n) = \mathcal{O}(m \cdot \log n)$

Zusammenfassung

Minimale Aufspannende Bäume (MST) Problem

- Schnitteigenschaft
- Die Algorithmen von Kruskal und Prim berechnen einen MST.
- Prim: fügt der Zus.-hangskomponente S die günstigste Kante des derzeitigen Schnitts von S hinzu (garantiert so Kreisfreiheit). $O((m+n)\log n)$
- Kruskal: jeweils die günstigste Kante ausgewählt, die keinen Kreis erzeugt. $\mathcal{O}(m \log n)$
- Verbesserte Laufzeit mit effizienten Datenstrukturen (Binary Heaps und Union-Find Datenstruktur)