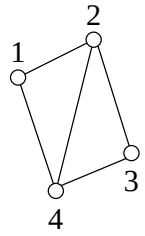
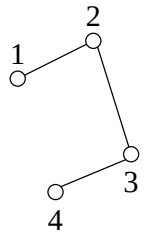
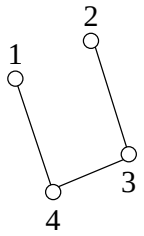
7 Minimaler Spannbaum und Datenstrukturen

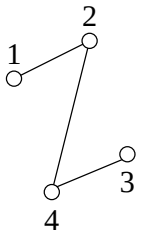
Hier betrachten wir als Ausgangspunkt stets zusammenhängende, ungerichtete Graphen.

$G =$ 

So sind

$G_1 =$ 

$G_2 =$ 

$G_3 =$ 

Spannbäume (aufspannende Bäume) von G. Beachte immer 3 Kanten bei 4 Knoten.

Definition 7.1(Spannbaum): *Ist $G = (V, E)$ zusammenhängend, so ist ein Teilgraph $H = (V, F)$ ein Spannbaum von G, genau dann, wenn H zusammenhängend ist und für alle $f \in F$ $H \setminus \{f\} = (V, F \setminus \{f, \cdot\})$ nicht mehr zusammenhängend ist. (H ist minimal zusammenhängend.)*

Folgerung 7.1: *Sei H zusammenhängend, Teilgraph von G.*

$$H \text{ Spannbaum von } G \iff H \text{ ohne Kreis.}$$

Beweis. „$\Rightarrow$“ Hat H einen Kreis, dann gibt es eine Kante f, so dass $H \setminus \{f\}$ zusammenhängend ist. ($A \Rightarrow B$ durch $\neg B \Rightarrow \neg A$)

„$\Leftarrow$ “ Ist H kein Spannbaum, dann gibt es eine Kante f, so dass $H \setminus \{f\}$ zusammenhängend ist. Dann haben wir einen Kreis in H, der f enthält. ($B \Rightarrow A$ durch $\neg A \Rightarrow \neg B$) □

Damit hat jeder Spannbaum genau $n - 1$ Kanten, wenn $|V| = n$ ist. Vergleiche Seite 35. Die Baumkanten einer Tiefen- bzw. Breitensuche ergeben einen solchen Spannbaum. Wir suchen Spannbäume minimaler Kosten.

Definition 7.2: *Ist $G = (V, E)$ und $K : E \to \mathbb{R}$ (eine Kostenfunktion) gegeben. Dann ist $H = (V, F)$ ein* minimaler Spannbaum *genau dann, wenn:*

- *H ist Spannbaum von G*
- *$K(H) = \sum_{f \in F} K(f)$ ist minimal unter den Knoten aller Spannbäume. ($K(H)$ = Kosten von H).*

Wie finde ich einen minimalen Spannbaum? Systematisches Durchprobieren. Verbesserung durch branch-and-bound.

- Eingabe: $G = (V, E), K :\to \mathbb{R}, E = \{e_1, \ldots, e_n\}$ (also Kanten)
- Systematisches Durchgehen aller Mengen von $n-1$ Kanten (alle Bitvektoren $b_1, \ldots, b_n$ mit genau $n-1$ Einsen), z.B, rekursiv.
- Testen, ob kein Kreis. Kosten ermitteln. Den mit den kleinsten Kosten ausgeben.
- Zeit: Mindestens
$$\binom{m}{n-1} = \frac{m!}{(n-1)! \cdot \underbrace{(m-n+1)!}_{\geq 0 \text{ für } n-1 \leq m}}$$

Wieviel ist das? Sei $m = 2 \cdot n$, dann

$$\begin{aligned}
\frac{(2n)!}{(n-1)!(n+1)!} &= \frac{2n \cdot (2n-1) \cdot (2n-2) \cdot \cdots \cdot n}{(n+1) \cdot n \cdot (n-1) \ldots 1} \\
&\geq \underbrace{\frac{2n-2}{n-1}}_{=2} \cdot \underbrace{\frac{2n-3}{n-2}}_{>2} \cdot \cdots \cdot \underbrace{\frac{n}{1}}_{>2} \\
&\geq \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{n-1 \text{ mal}} = 2^{n-1}
\end{aligned}$$

Also Zeit $\Omega(2^n)$.

Regel: Sei $c \geq 0$, dann:

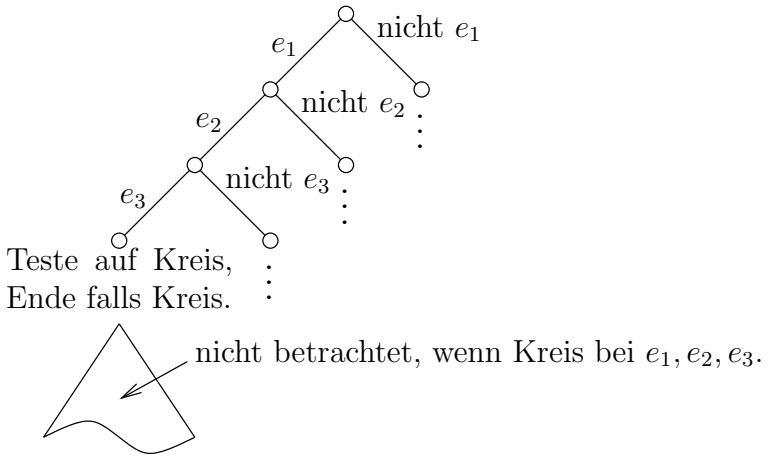
$$\frac{a}{b} \leq \frac{a-c}{b-c} \Leftrightarrow b \leq a.$$

Das c im Nenner hat mehr Gewicht.

Definition 7.3(Ω-Notation): *Ist $f, g : \mathbb{N} \to \mathbb{R}$, so ist $f(u) = \Omega(g(u))$ genau dann, wenn es eine Konstante $C > 0$ gibt, mit $|f(n)| \geq C \cdot g(n)$ für alle hinreichend großen n. (vergleiche O-Notation, Seite 40)*

Bemerkung: *O-Notation:* Obere Schranke,
Ω-Notation: Untere Schranke.

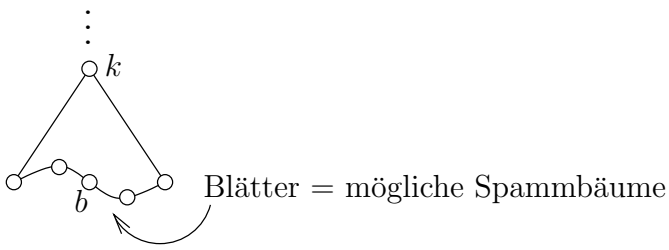
Verbesserung: Backtracking, frühzeitiges Erkennen von Kreisen. Dazu Aufbau eines Baumes (Prozeduraufrufbaum):



Alle Teilmengen mit e_1, e_2, e_3 sind in einem Schritt erledigt, wenn ein Kreis vorliegt.

Einbau einer Kostenschranke in den Backtracking-Algorithmus: *Branch-and-bound.*

Kosten des Knotens $k = \sum_{\substack{f \text{ in } k \\ \text{gewählt}}} K(f)$.



Es ist für Blätter wie b unter k.

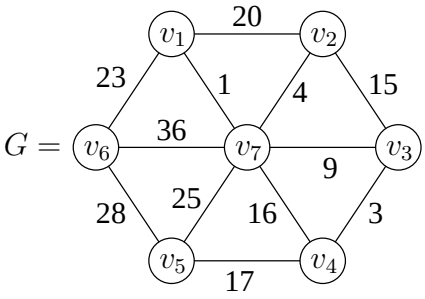
$$\text{Kosten von } k \leq \text{Kosten von } b.$$

Eine Untere Schranke an die Spannbäume unter k.

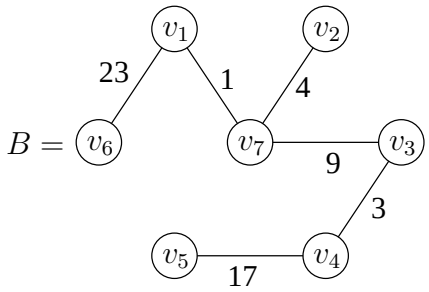
Also weiteres Rausschneiden möglich, wenn Kosten von $k \geq$ Kosten eines bereits gefundenen Baumes. Im allgemeinen ohne weiteres keine bessere Laufzeit nachweisbar.

Besser: Irrtumsfreier Aufbau eines minimalen Spannbaumes.

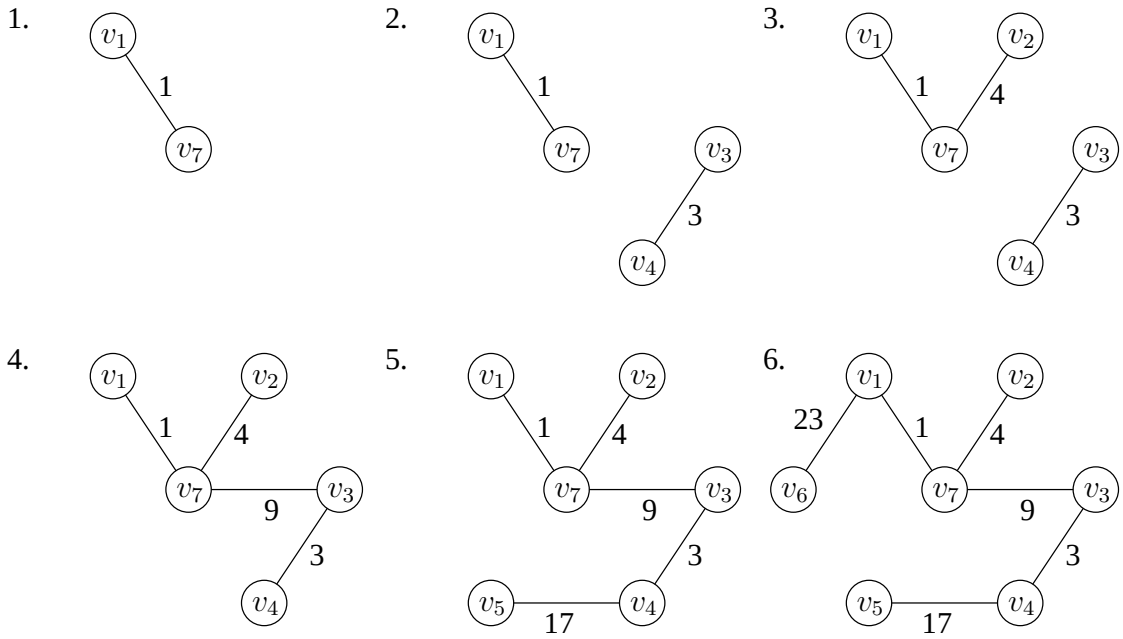
Betrachten wir den folgenden Graphen G mit Kosten an den Kanten.

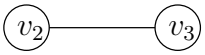


Dazu ist B ein Spannbaum mit minimalen Kosten.



Der Aufbau eines minimalen Spannbaums erfolgt schrittweise wie folgt:



Nimm die günstigste Kante, die keinen Kreis ergibt, also nicht  ,  in Schritt 5 bzw 6.

Implementierung durch Partition von V

$\{v_1\}, \{v_2\}, \ldots, \{v_7\}$	keine Kante
$\{v_1, v_7\}, \{v_2\}, \{v_3\}, \{v_4\}, \{v_5\}, \{v_6\}$	$\{v_1, v_7\}$
$\{v_1, v_7\}, \{v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}$
$\{v_1, v_7, v_2\}, \{v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}, \{v_7, v_2\}$
$\{v_1, v_7, v_2, v_3, v_4\}, \ldots$	$\{v_1, v_7\}, \{v_3, v_4\}, \{v_7, v_2\}, \{v_7, v_3\}$
v_5 hinzu	$\vdots$
v_6 hinzu	

Problem: Wie Partition darstellen?

7.1 Algorithmus Minimaler Spannbaum (Kruskal 1956)

Algorithmus 10: Minimaler Spannbaum (Kruskal 1956)

Input : $G = (V, E)$ zusammenhängend, $V = \{1, \ldots, n\}$, Kostenfunktion $K : E \to \mathbb{R}$

Output : F = Menge von Kanten eines minimalen Spannbaumes.

```
1  F = ∅;
2  P = {{1}, ..., {n}};                          /* P ist die Partition */
3  E nach Kosten sortieren; /* Geht in O(|E| · log |E|) = O(|E| · log |V|), */
                                  /* |E| ≤ |V|², log |E| = O(log |V|) */
4  while |P| > 1 do                              /* solange P ≠ {{1, 2, ..., n}} */
5  |  {v, w} = kleinstes (erstes) Element von E;
6  |  {v, w} aus E löschen;
7  |  if {v, w} induziert keinen Kreis in F then
8  |  |  W_v = die Menge mit v aus P;
9  |  |  W_w = die Menge mit w aus P;
10 |  |  W_v und W_w in P vereinigen;
11 |  |  F = F ∪ {{v, w}}; /* Die Kante kommt zum Spannbaum dazu. */
12 |  end
13 end
```

Der Algorithmus arbeitet nach dem Greedy-Prinzip (*greedy = gierig*):

> Es wird zu jedem Zeitpunkt die günstigste Wahl getroffen (= Kante mit den kleinsten Kosten, die möglich ist) und diese genommen. Eine einmal getroffene Wahl bleibt bestehen. Die lokal günstige Wahl führt zum globalen Optimum.

Zur Korrektheit des Algorithmus:

Beweis. Sei F_l = der Inhalt von F nach dem l-ten Lauf der Schleife. P_l = der Inhalt von P nach dem l-ten Lauf der Schleife.

Wir verwenden die Invariante: „F_l lässt sich zu einem minimalen Spannbaum fortsetzen." (D.h., es gibt $F \supseteq F_l$, so dass F ein minimaler Spannbaum ist.)

P_l stellt die Zusammenhangskomponenten von F_l dar.

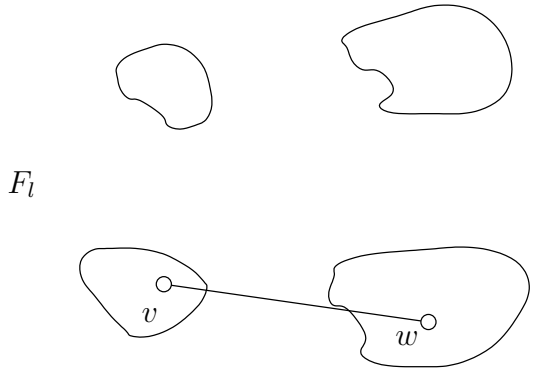
Die Invariante gilt für $l = 0$.

Gelte sie für l und finde ein $l + 1$-ter Lauf statt.

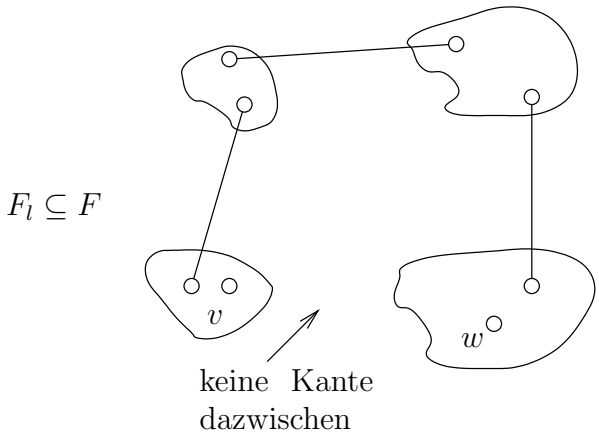
1. Fall Die Kante $\{v, w\}$ wird nicht genommen. Alles bleibt unverändert. Invariante gilt für $l + 1$.

2. Fall $\{v, w\}$ wird genommen. $\{v, w\}$ = Kante von minimalen Kosten, die zwei Zusammenhangskomponenten von F_l verbindet.

Also liegt folgende Situation vor:



Zu zeigen: Es gibt einen minimalen Spannbaum, der $F_{l+1} = F_l \cup \{v, w\}$ enthält. Nach Invariante für F_l gibt es mindestens Spannbaum $F \supseteq F_l$. Halten wir ein solches F fest. Dieses F kann, muss aber nicht, $\{v, w\}$ enthalten. Zum Beispiel:



Falls $F\{v, w\}$ enthält, gilt die Invariante für $l+1$ (sofern die Operation auf P richtig implementiert ist). Falls aber F $\{v, w\}$ nicht enthält, argumentieren wir so:

$F \cup \{v, w\}$ enthält einen Kreis (sonst F nicht zusammenhängend). Dieser Kreis muss mindestens eine weitere Kante haben, die zwei verschiedene Komponenten von F_l verbindet, also nicht zu F_l gehört. Diese Kante gehört zu der Liste von Kanten E_l, hat also Kosten, die höchstens größer als die von $\{v, w\}$ sind.

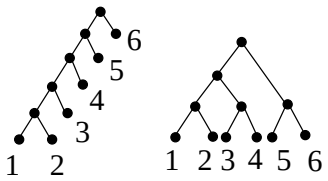
Tauschen wir in F diese Kante und $\{v, w\}$ aus, haben wir einen minimalen Spannbaum mit $\{v, w\}$. Also gilt die Invariante für $F_{l+1} = F_l \cup \{v, w\}$ und P_{l+1} (sofern Operationen richtig implementiert sind.)

Quintessenz: Am Ende ist oder hat F_l nur noch eine Komponente, da $|P_l| = 1$. Also minimaler Spannbaum wegen Invariante.

Termination: Entweder wird die Liste E_l kleiner oder P_l wird kleiner. □

Laufzeit:

- Initialisierung: $O(n)$
- Kanten sortieren: $O(|E| \cdot log|E|)$ (wird später behandelt), also $O(|E| \cdot log|V|)$!
- Die Schleife:
  - $n - 1 \leq$ #Läufe $\leq |E|$
    Müssen $\{\{1\}, \ldots, \{n\}\}$ zu $\{\{1, \ldots, n\}\}$ machen. Jedesmal eine Menge weniger, da zwei vereinigt werden. Also $n - 1$ Vereinigungen, egal wie vereinigt wird.



    Binärer Baum mit n Blättern hat immer genau $n - 1$ Nicht-Blätter.
  - Für $\{v, w\}$ suchen, ob es Kreis in F induziert, d.h. ob v, w innerhalb einer Menge von P oder nicht.

    Bis zu $|E|$-mal

  - W_v und W_w in P vereinigen

    Genau $n - 1$-mal.

Die Zeit hängt von der Darstellung von P ab.

Einfache Darstellung: Array $P[1, \ldots, n]$, wobei eine Partition von $\{1, \ldots, n\}$ der Indexmenge dargestellt wird durch:

- u, v in gleicher Menge von $P \longleftrightarrow P[u] = P[v]$
  (u, v sind Elemente der Grundmenge = Indizes)
- Die Kante $\{u, v\}$ induziert Kreis $\longleftrightarrow$ u, v in derselben Zusammenhangskomponente von $F \longleftrightarrow P[u] = P[v]$
- W_u und W_v vereinigen:

Prozedur union(v, w)

```
1 a = P[v];
2 b = P[w];                                   /* a ≠ b, wenn W_u ≠ W_v */
3 for i = 1 to n do
4 |   if P[i] == a then
5 |   |   P[i] = b;
6 |   end
7 end
```

Damit ist die Laufzeit der Schleife:

7.-11.	Bestimmen der Mengen und Testen auf Kreis geht jeweils in $O(1)$, also insgesamt $O(n)$. Die Hauptarbeit liegt im Vereinigen der Mengen. Es wird $n-1$-mal vereinigt, also insgesamt $n \cdot O(n) = O(n^2)$.
5. und 6.	Insgesamt $O(\|E\|)$.
4.	$O(n)$ insgesamt, wenn $\|P\|$ mitgeführt.

Also $O(n^2)$, selbst wenn E bereits sortiert ist.

Um die Laufzeit zu verbessern, müssen wir das Vereinigen der Mengen schneller machen. Dazu stellen wir P durch eine *Union-Find-Struktur* dar.

7.2 Union-Find-Struktur

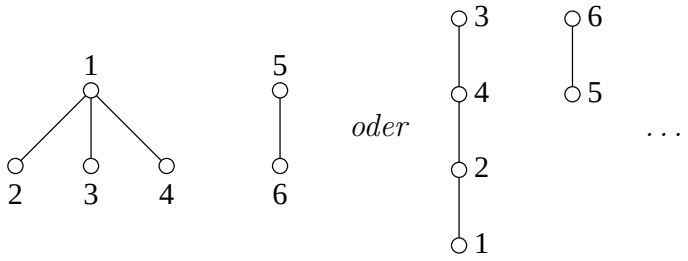
- Darstellung einer Partition P einer Grundmenge (hier klein, d.h. als Indexmenge verwendbar)
- *Union*(U, W): Für $U, W \in P$ soll $U, W \in P$ durch $U \cup W \in P$ ersetzt werden.
- *Find(u)*: Für u aus der Grundmenge soll der Name der Menge $U \in P$ mit $u \in U$ geliefert werden. (Es geht nicht um das Finden von u, sondern darum in welcher Menge u sich befindet!)

Für *Kruskal* haben wir $\leq 2 \cdot |E|$-mal Find(u) + $|V| - 1$-mal Union(u,w) $\Rightarrow$ Zeit $\Omega(|E| + |V|)$, sogar falls $|E|$ bereits sortiert ist.

Definition 7.4(Datenstruktur Union-Find): *Die Mengen werden wie folgt gespeichert:*

(a) Jede Menge von P als ein Baum mit Wurzel.

Sei $P = \{\{1,2,3,4\},\{5,6\}\}$, dann etwa Wurzel = Name der Menge. Die Bäume können dann so aussehen:



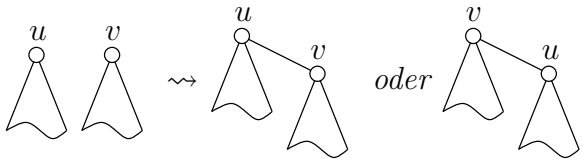
Die Struktur der Bäume ist zunächst egal.

(b) Find(u)*:*

1. *u in den Bäumen finden. Kein Problem, solange Grundmenge Indexmenge sein kann.*
2. *Gehe von u aus hoch bis zur Wurzel.*
3. *(Namen der) Wurzel ausgeben.*

Union (u, v)*:* *(u, v sind Namen von Mengen, also Wurzeln.)*

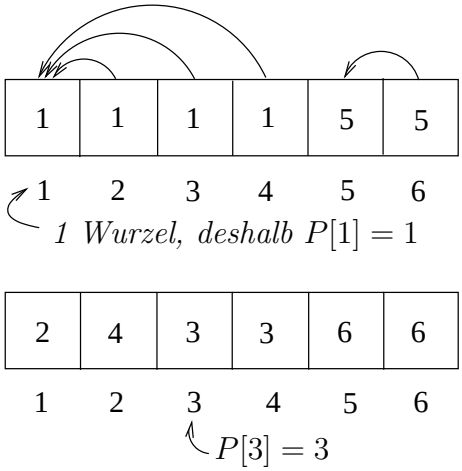
1. *u, v in den Bäumen finden.*
2. *Hänge u unter v (oder umgekehrt).*



(c) Bäume in Array (Vaterarray) speichern:

$$P[1,\ldots,n] \text{ of } \{1,\ldots,n\}, \quad P[v] = \text{Vater von } v.$$

Aus (a) weiter:



Wieder wichtig: Indices = Elemente. Vaterarray kann beliebige Bäume darstellen.

Prozedur union(u, v)

```
   /* u und v müssen Wurzeln sein! */
1  P[u] = v;                                   /* u hängt unter v. */
```

Prozedur find(v)

```
1 while P[v] ≠ v do                            /* O(n) im worst case */
2 |  v = P[v];
3 end
4 return v
```

Beispiel 7.1: *Wir betrachten die folgenden Operationen.*

1. ① ② ③ ④ $\cdots$ Ⓝ

$P =$

1	*2*	*3*	$\ldots$	N
1	*2*	*3*	$\ldots$	N

2. $\curvearrowright$ *union(1,2) in* $O(1)$

② ③ ④ $\cdots$ Ⓝ

①

$P =$

2	*2*	*3*	$\ldots$	N
1	*2*	*3*	$\ldots$	N

3. $\curvearrowright$ *union(2,3) in* $O(1)$

③ ④ $\cdots$ Ⓝ

②

①

$P =$

2	*3*	*3*	$\ldots$	N
1	*2*	*3*	$\ldots$	N

4. $\curvearrowright$ *union(3,4) in* $O(1)$

④ $\cdots$ Ⓝ

③

②

①

Das Entstehen langer dünner Bäume.

$\vdots$

N. $\curvearrowright$ *union(N-1,N) in* $O(1)$

$P =$

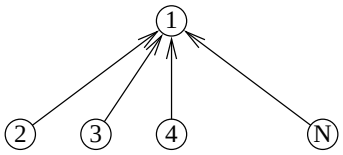
1	*2*	*3*	...	*N*	*N*
1	*2*	*3*	...	*N-1*	*N*

Find in $\Omega(N)$.

Beispiel 7.2: *Wir Vertauschen die Argumente bei der union-Operation.*

$P =$ ① ② ③ ④ $\cdots$ Ⓝ *und union(2,1); union(3,1); ... union(N,1).*

Dann erhalten wir den folgenden Baum



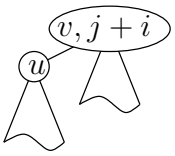
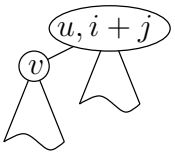
und die Operation find(2) geht in $O(1)$.

$\Rightarrow$ Union-by-Size; kleinere Anzahl nach unten.

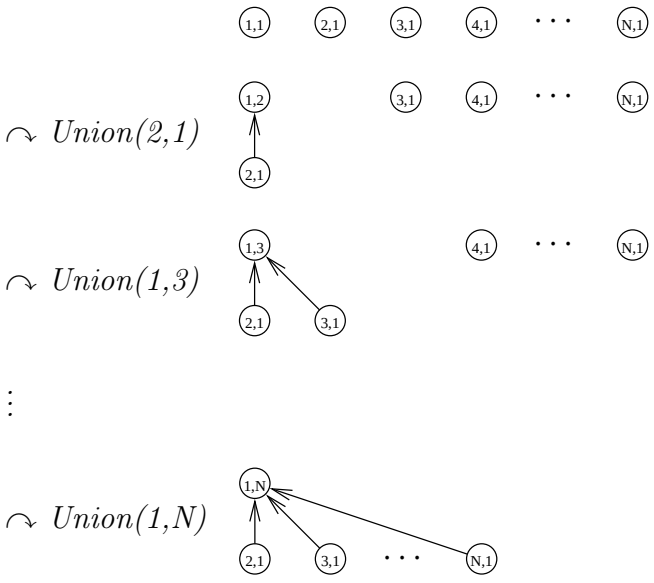
$\Rightarrow$ Maximale Tiefe (längster Weg von Wurzel zu Blatt) $= log_2 N$.
Beachte: Das Verhältnis von $log_2 N$ zu $N = 2^{log_2 N}$ ist wie N zu 2^N.

7.3 Algorithmus Union-by-Size

Prozedur union$((u,i),(v,j))$

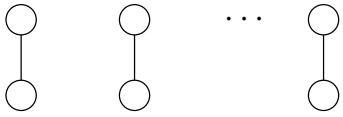
```
   /* i, j sind die Anzahlen der Elemente in dem Mengen. Muss
      mitgeführt werden. */
1  if i ≤ j then
2  |   
   |                                                  /* u unter v. */
3  else
4  |   
   |                                                  /* v unter u. */
5  end
```


Beispiel 7.3: *Union by Size:*

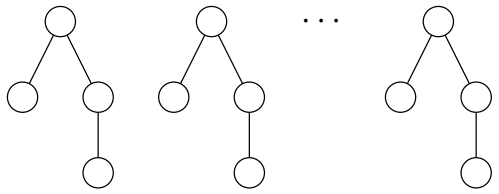


Wie können tiefe Bäume durch Union-by-Size entstehen?

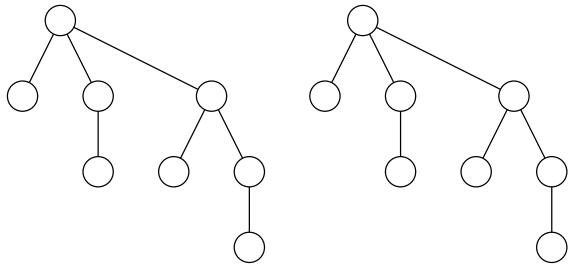
1. 

2. 

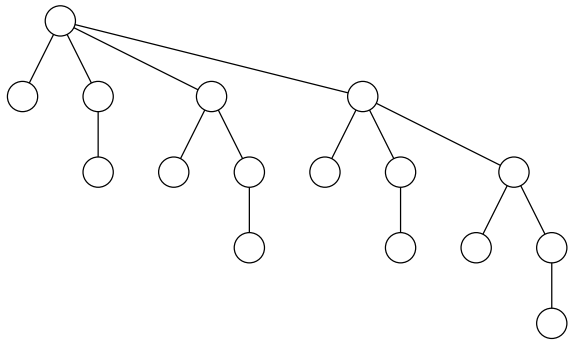
3.



4.



5.



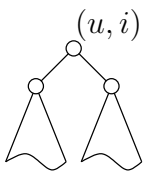
Vermutung: N Elemente $\Rightarrow$ Tiefe $\leq log_2 N$.

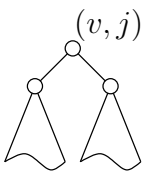
Satz 7.1: *Beginnend mit* $P = \{\{1\}, \{2\}, \ldots, \{n\}\}$ *gilt, dass mit Union-by-Size für jeden entstehenden Baum* T *gilt: Tiefe(*T*)* $\leq \log_2 |T|$. *(*$|T|$ *= #Elemente von* T *= #Knoten von* T*)*

Beweis. Induktion über die Anzahl der ausgeführten Operationen union(u, v), um T zu bekommen.

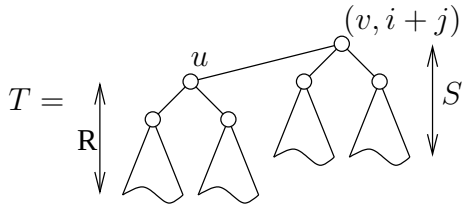
Induktionsanfang: kein union(u,v) ✓ Tiefe$(u) = 0 = \log_2 1$ $(2^0 = 1)$

Induktionsschluss: T durch union(u, v).

$R =$ 

$S =$ 

Sei $i \leq j$, dann



1. Fall: keine größere Tiefe als vorher. Die Behauptung gilt nach Induktionsvoraussetzung, da T mehr Elemente hat erst recht.

2. Fall: Tiefe vergrößert sich echt. Dann aber

$$\text{Tiefe}(T) = \text{Tiefe}(R) + 1 \leq (\log_2 |R|) + 1 = \log_2(2|R|) \leq |T|$$

$(2^x = |R| \Rightarrow 2 \cdot 2^x = 2^{x+1} = 2|R|)$

Die Tiefe wird immer nur um 1 größer. Dann mindestens Verdopplung der #Elemnte.

□

Mit Bäumen und Union-by-Size Laufzeit der Schleife:

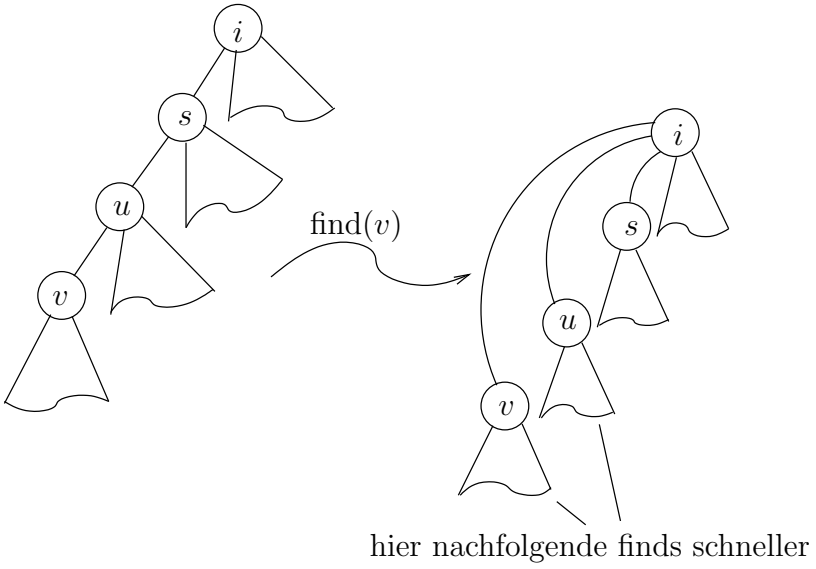
$\leq 2|E|$-mal Find(u): $O(|E| \log |V|)$ $\quad$ ($\log |V|$ ... Tiefe der Bäume)

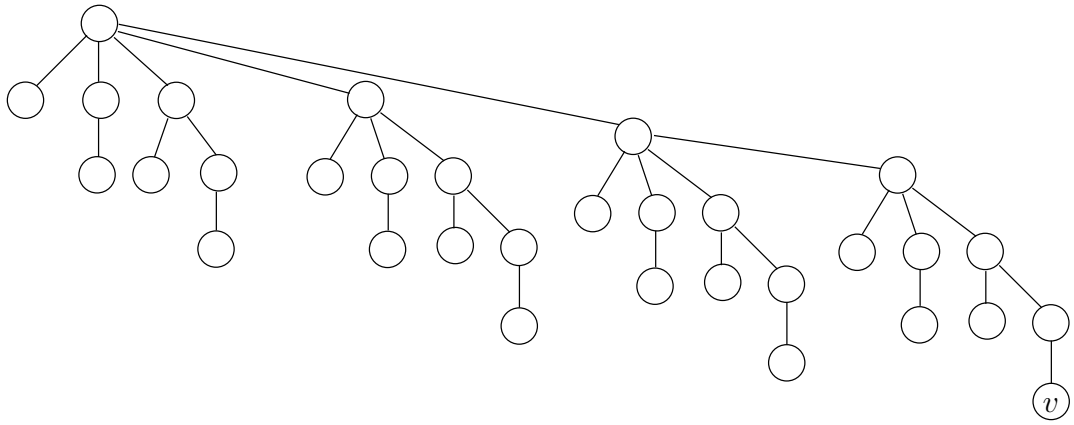
$n-1$-mal Union(u, v): $O(n)$

Rest $O(|E|)$. Also insgesamt $O(|E| \cdot \log |V|)$

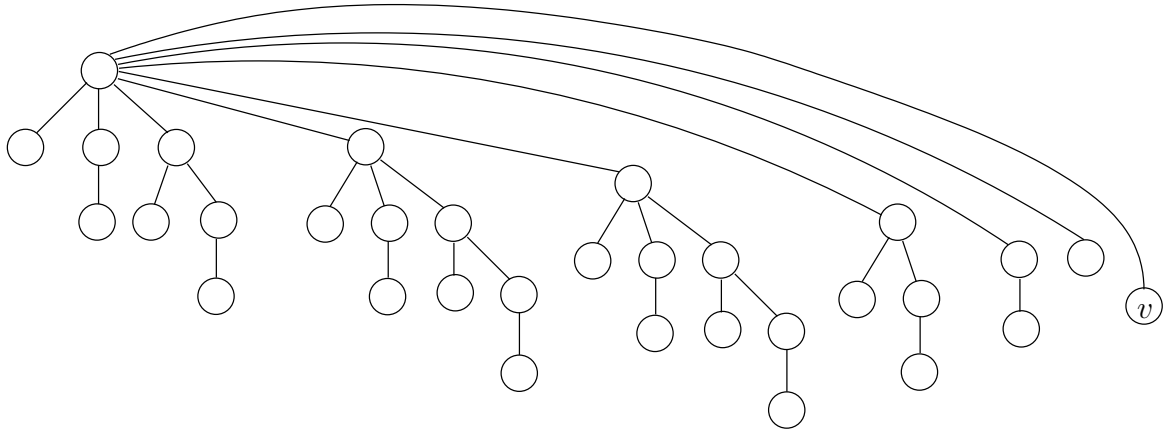
Vorne: $O(|V|^2)$. Für dichte Graphen ist $|E| = \Omega(\frac{|V|^2}{\log |V|})$. → So keine Verbesserung erkennbar.

Beispiel 7.1 (*Wegkompression*):





$\curvearrowright$ find(v)



Im Allgemeinen $\Omega(\log N)$ und $O(\log N)$

7.4 Algorithmus Wegkompression

Prozedur find(v)

```
1  S = ∅;          /* leerer Keller, etwa als Array implementieren */
2  push(S, v);
3  while P[v] ≠ v do            /* In v steht am Ende die Wurzel. */
4  |  v = P[v];
5  |  push(S, v);
6  end
   /* Auch Schlange oder Liste als Datenstruktur ist möglich. Die
      Reihenfolge ist egal. */
7  foreach w ∈ S do
   |  /* Alle unterwegs gefundenen Knoten hängen jetzt unter der
   |     Wurzel. */
8  |  P[w] = v;
9  end
10 return v
```

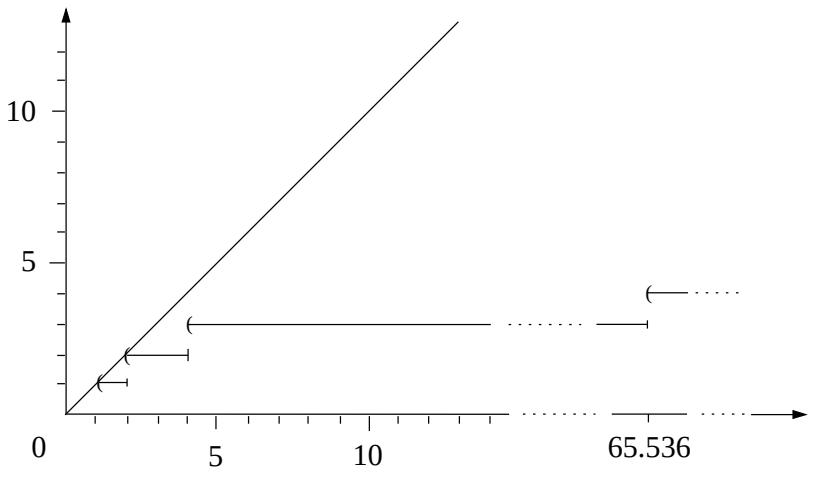
Laufzeit $O(\log |V|)$ reicht immer, falls Union-by-Size dabei.

Es gilt sogar (einer der frühen Höhepunkte der Theorie der Datenstrukturen):

> $n-1$ Unions, m Finds mit Union-by-Size und Wegkompression in Zeit $O(n + (n+m) \cdot log^*(n))$ bei anfangs $P = \{\{1\}, \{2\}, \ldots, \{n\}\}$.

Falls $m \geq \Omega(n)$ ist das $O(n + m \cdot \log^*(n))$. (Beachten Sie die Abhängigkeit der Konstanten: O-Konstante hängt von Ω-Konstante ab!)

Was ist $\log^*(n)$?



Die Funktion $\log^*$.

$$
\begin{aligned}
\log^* 1 &= 0 \\
\log^* 2 &= 1 \\
\log^* 3 &= 2 \\
\log^* 4 &= 1 + \log^* 2 = 2 \\
\log^* 5 &= 3 \\
\log^* 8 &= \log^* 3 + 1 = 3 \\
\log^* 16 &= \log^* 4 + 1 = 3 \\
\log^* 2^{16} &= 4 \\
\log^* 2^{2^{16}} &= 5 \qquad 2^{16} = 65.536
\end{aligned}
$$

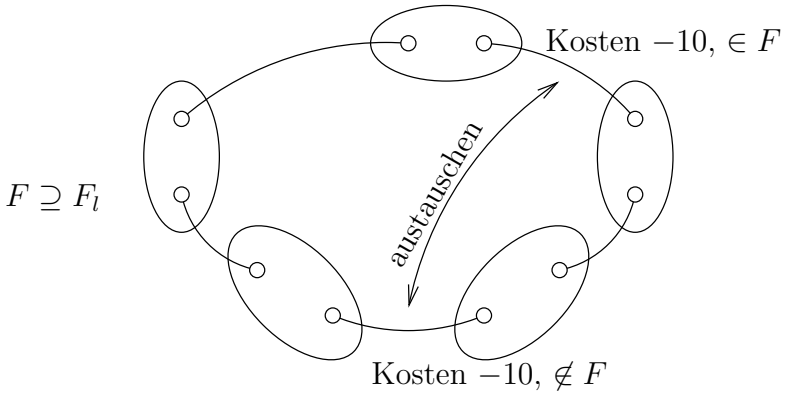
$$
\begin{aligned}
\log^* n &= \min\{s | \log^{(s)}(n) \leq 1\} \\
\log^* 2^{(2^{(2^{(2^{(2^2)})})})} &= 7 \\
\log^{(s)}(n) &= \underbrace{\log(\log(\ldots(\log}_{s\text{-mal}}(n))\ldots))
\end{aligned}
$$

Kruskal bekommt dann eine Zeit von $O(|V|+|E|\cdot\log^*|V|)$. Das ist fast $O(|V|+|E|)$ bei vorsortierten Kanten, natürlich nur $\Omega(|V|+|E|)$ in jedem Fall.

Nachtrag zur Korrektheit von Kruskal:

1. Durchlauf: Die Kante mit minimalen Kosten wird genommen. Gibt es mehrere, so ist egal, welche.

$l+1$**-ter Lauf:** Die Kante mit minimalen Kosten, die zwei Komponenten verbindet, wird genommen.



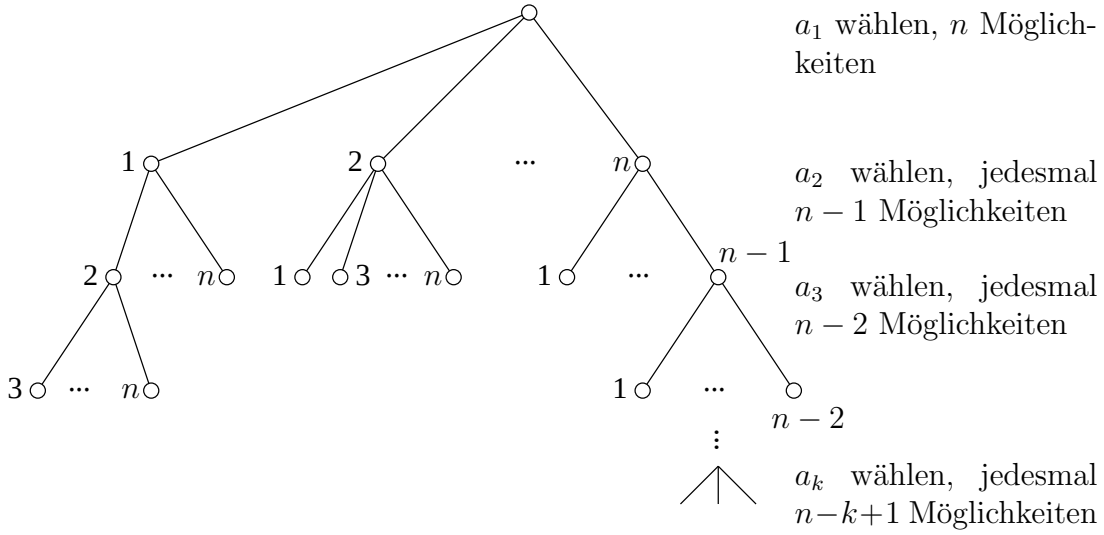
Alle Kanten, die zwei Komponenten verbinden, haben Kosten ≥ -10.

Beachte: Negative Kosten sind bei Kruskal kein Problem, nur bei branch-and-bound (vorher vergrößern)!

Binomialkoeffizient $\binom{n}{k}$ für $n \geq k \geq 0$

$$\begin{aligned}
\binom{n}{k} &= \frac{n!}{(n \cdot k) \cdot k!} = \frac{\overbrace{n(n-1)\dots(n-k+1)}^{\text{oberste } k \text{ Faktoren von } n!}}{k!} \qquad 0! = 1! = 1 \\
\binom{n}{k} &= \#\text{Teilmengen von } \{1,\dots,n\} \text{ mit genau } k \text{ Elementen} \\
\binom{n}{1} &= n, \quad \binom{n}{2} = \frac{n(n-1)}{2}, \quad \binom{n}{k} \leq n^k
\end{aligned}$$

Beweis. Auswahlbaum für alle Folgen $(a_q, \dots, a_k)$ mit $a_i \in \{1, \dots, n\}$, alle a_i verschieden.



Der Baum hat $\underbrace{n(n-1)\dots\overbrace{(n-k+1)}^{=n-(k-1)}}_{\substack{k \text{ Faktoren (nicht } k-1\text{, da} \\ 0,1,\dots,k+1 \text{ genau } k \text{ Zahlen)}}} = \frac{n!}{(n-k)!}$ Blätter.

Jedes Blatt entspricht genau einer Folge $(a_1, \dots, a_k)$. Jede Menge aus k Elementen kommt $k!$-mal vor: $\{a_1, \dots, a_k\}$ ungeordnet, geordnet als

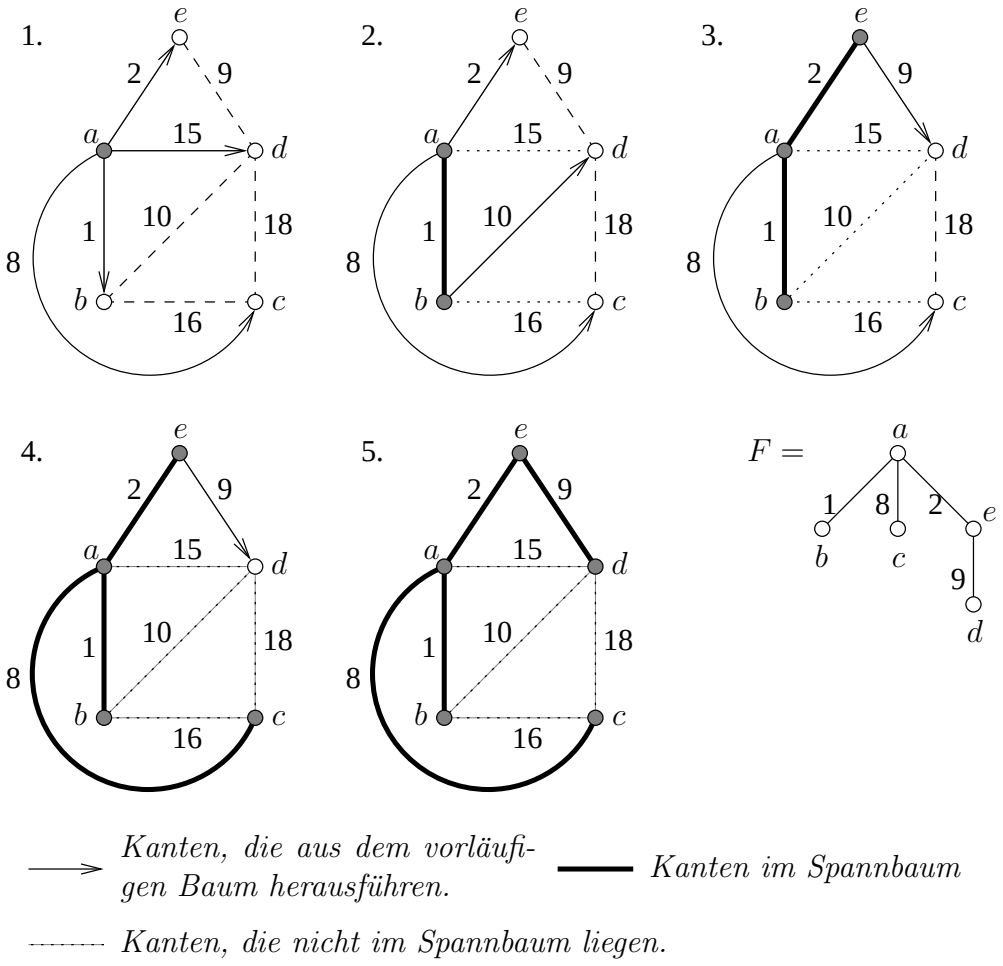
$$(a_1, \dots, a_k), (a_2, a_1, \dots, a_k), \dots, (a_k, a_{k-1}, \dots, a_2, a_1).$$

Das sind $k!$ Permutationen. Also $\frac{n!}{(n-k)!k!} = \binom{n}{k}$ Mengen mit k verschiedenen Elementen. □

7.5 Algorithmus Minimaler Spannbaum nach Prim (1963)

Dieser Algorithmus verfolgt einen etwas anderen Ansatz als der von Kruskal. Hier wird der Spannbaum schrittweise von einem Startknoten aus aufgebaut. Dazu werden in jedem Schritt die Kanten betrachtet, die aus dem bereits konstruierten Baum herausführen. Von allen diesen Kanten wählen wir dann eine mit *minimalem Gewicht* und fügen sie dem Baum hinzu. Man betrachte das folgende Beispiel.

Beispiel 7.4:



Algorithmus 11: Minimaler Spannbaum (Prim)

Input : $G = (V, E)$ zusammenhängend, $V = \{1, \ldots, n\}$, Kostenfunktion $K : E \to \mathbb{R}$

Output : $F =$ Menge von Kanten eines minimalen Spannbaumes.

```
1 Wähle einen beliebigen Startknoten s ∈ V;
2 F = ∅;
  /* Q enthält immer die Knoten, die noch bearbeitet werden
     müssen. */
3 Q = V \ {s};
4 while Q ≠ ∅ do
  |  /* V \ Q sind die Knoten im Baum. In M sind alle Kanten,
  |     die einen Knoten in V \ Q und einen in Q haben. Also alle
  |     Kanten, die aus dem Baum herausführen. */
5 |  M = {{v, w} ∈ E | v ∈ V \ Q, w ∈ Q};
6 |  {v, w} = eine Kante mit minimalen Kosten in M;
7 |  F = F ∪ {{v, w}};
8 |  Q = Q \ {w};                    /* w ist jetzt Teil des Baumes. */
9 end
```

Korrektheit mit der Invariante:

- Es gibt einen minimalen Spannbaum $F \supseteq F_l$.

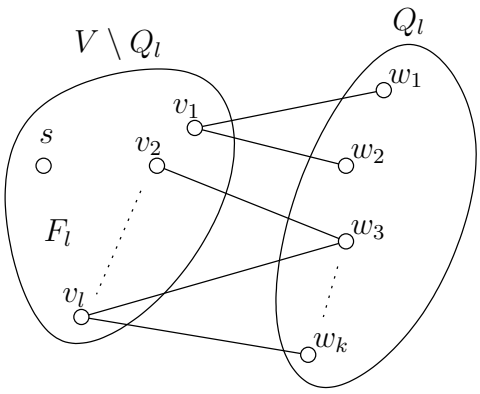
$$F_0 = \emptyset$$

- Q_l ist die Menge Q des Algorithmus nach dem l-ten Schritt. Also die Menge der Knoten, die nach l Schritten noch nicht im Spannbaum sind.

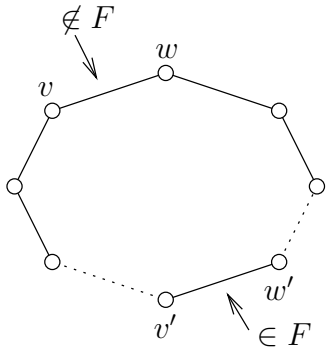
$$\begin{aligned} Q_0 &= V \setminus \{s\} \\ Q_l &= V \setminus \{u \mid u \in F_l \text{ oder } u = s\} \end{aligned}$$

Beweis. $l = 0$ ✓. Jeder Spannbaum enthält den Startknoten. Das ergibt sich aus der Definition des Spannbaumes.

Gelte die Invariante für l und finde ein $l + 1$-ter Lauf der Schleife statt. Sei $F \supseteq F_l$ ein minimaler Spannbaum, der nach Induktionsvoraussetzung existiert.



Werde $\{v, w\}$ im $l+1$-ten Lauf genommen. Falls $\{v, w\} \in F$, dann gilt die Invariante auch für $l+1$. Sonst gilt $F \cup \{\{v, w\}\}$ enthält einen Kreis, der $\{v, w\}$ enthält. Dieser enthält mindestens eine weitere Kante $\{v', w'\}$ mit $v' \in V \setminus Q_l,\ w' \in Q_l$.



Es ist $K(\{v, w\}) \leq K(\{v', w'\})$ gemäß Prim. Also, da F minimal, ist $K(\{v, w\} = K(\{v', w'\})$. Wir können in F die Kante $\{v', w'\}$ durch $\{v, w\}$ ersetzen und haben immernoch einen minimalen Spannbaum. Damit gilt die Invariante für $l + 1$. □

Laufzeit von Prim:

- $n - 1$ Läufe durch 4-8.
- 5. und 6. einmal $O(|E|)$ reicht sicherlich, da $|E| \geq |V| - 1$.

Also $O(|V| \cdot |E|)$, bis $O(n^3)$ bei $|V| = n$. Bessere Laufzeit durch bessere Verwaltung von Q.

- Maßgeblich dafür, ob $w \in Q$ in den Baum aufgenommen wird, sind die minimalen Kosten *einer Kante* $\{v, w\}$ für $v \notin Q$.
- Array $key[1 \ldots n]$ of real mit der Intention: Für alle $w \in Q$ ist $key[w] =$ minimale Kosten einer Kante $\{v, w\}$, wobei $v \notin Q$. ($key[w] = \infty$, gdw. keine solche Kante existiert)

- Außerdem Array $kante[1, \ldots, n]$ of $\{1, \ldots, n\}$. Für alle $w \in Q$ gilt:

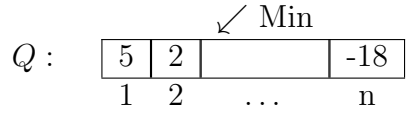
$$kante[w] = v \Longleftrightarrow \{v, w\} \text{ ist eine Kante minimaler Kosten mit } v \notin Q.$$

Wir werden Q mit einer Datenstruktur für die *priority queue* (Vorrangwartesschlange) implementieren:

- Speichert Menge von Elementen, von denen jedes einen Schlüsselwert hat. (keine Eindeutigkeitsforderung).
- Operation `Min` gibt uns (ein) Elemnet mit minimalem Schlüsselwert.
- `DeleteMin` löscht ein Element mit minimalem Schlüsselwert.
- `Insert`(v, s) = Element v mit Schlüsselwert s einfügen.

Wie kann man eine solche Datenstruktur implementieren?

1. Möglichkeit: etwa als Array



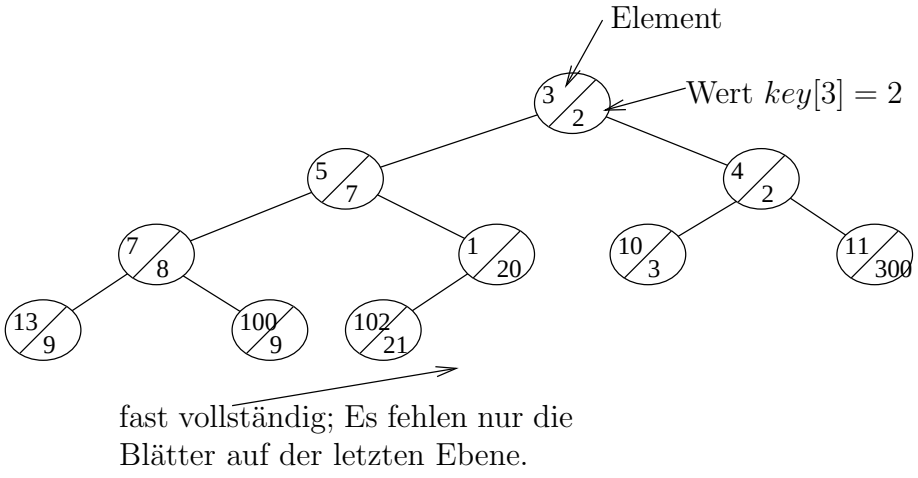
- Indices = Elemente
- Einträge in Q = Schlüsselwerte, Sonderwert (etwa $-\infty$) bei „nicht vorhanden“.

Zeiten:

- `Insert`(v, s) in $O(1)$ (sofern keine gleichen Elemente mehrfach auftreten)
- `Min` in $O(1)$
- `DeleteMin` in $O(1)$ fürs Finden des zu löschenden Elementes, dann aber $O(n)$, um ein neues Minimum zu ermitteln.

2. Möglichkeit: Darstellung als Heap.

Heap = „fast vollständiger“ binärer Baum, dessen Elemente (= Knoten) bezüglich Funktionswerten $key[j]$ nach oben hin kleiner werden.

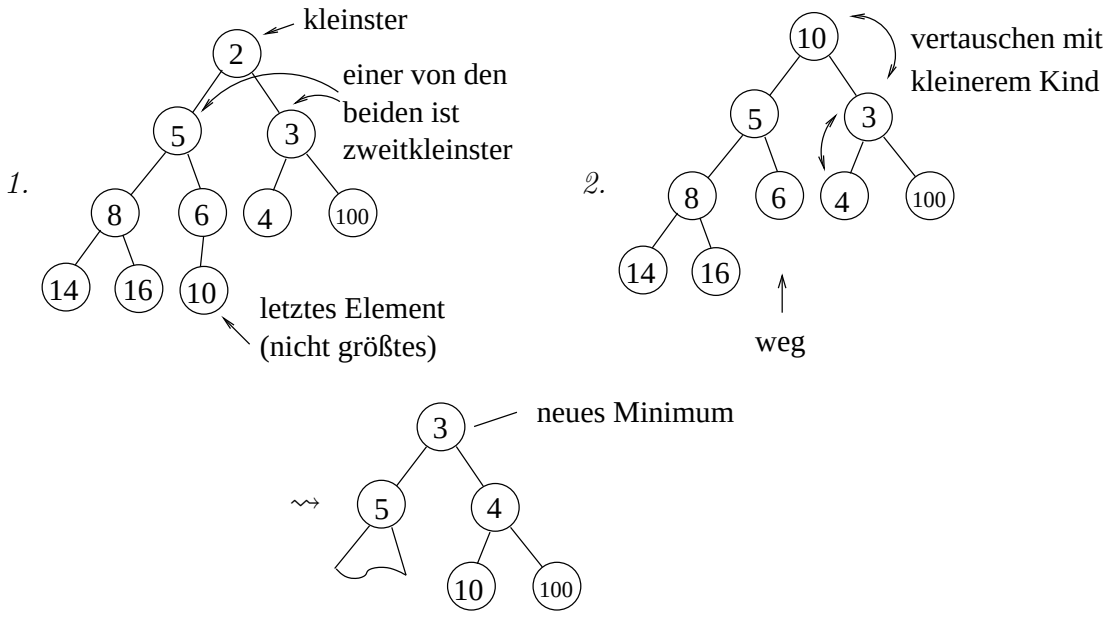


Tiefe n, dann $2^n - 1$ innere Knoten und 2^n Blätter. Insgesamt $2^{n+1} - 1$ Elemente.

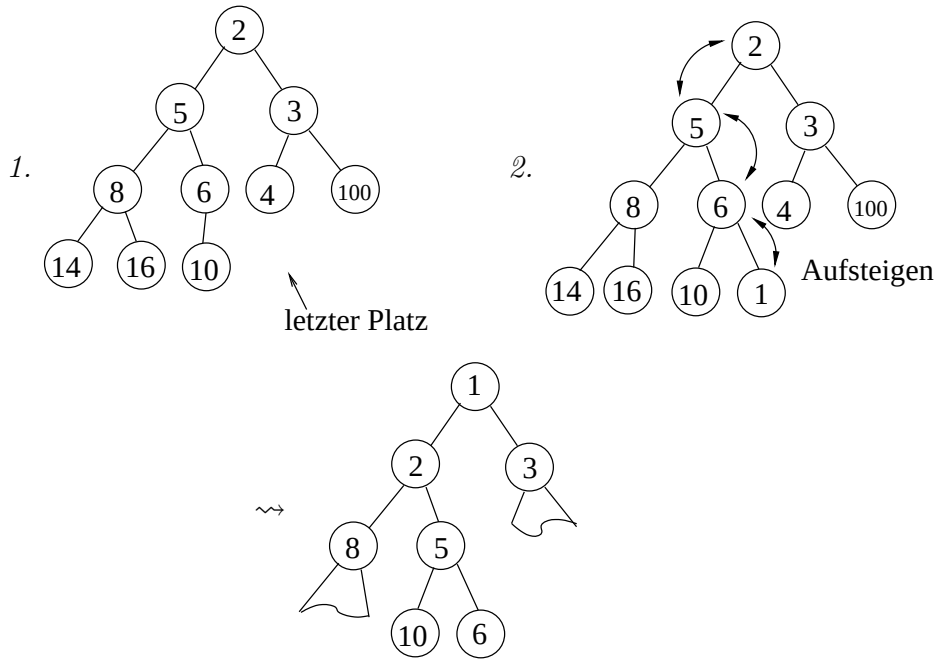
Heapeigenschaft: Für alle u, v gilt:

$$u \text{ Vorfahr von } v \Longrightarrow key[u] \leq key[v]$$

Beispiel 7.5(Minimum löschen): *Es sind nur die Schlüsselwerte key[i] eingezeichnet.*



Beispiel 7.6(Einfügen): *Element mit Wert 1 einfügen.*



Die Operationen der Priority Queue mit Heap, sagen wir Q.

Prozedur Min(Q)

```
return Element der Wurzel;
/* Laufzeit: O(1) */
```

Prozedur DeleteMin(Q)

```
Nimm „letztes“ Element, setze es auf Wurzel;
x = Wurzel(-element);
while key[x] ≥ key[linker Sohn] oder key[x] ≥ key[rechter Sohn] do
    Tausche x mit kleinerem Sohn;
end
/* Laufzeit: O(log n), da bei n Elementen maximal log n
   Durchläufe. */
```

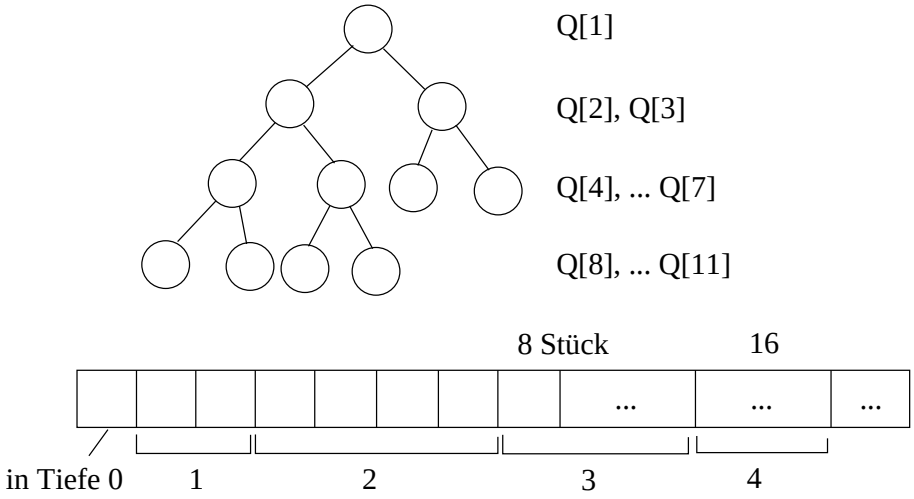
Prozedur Insert(Q, v, s)

```
key[v] = s;
Setze v als letztes Element in Q;
while key[Vater von v] > key[v] do
    Vertausche v mit Vater;
end
/* Laufzeit: O(log n) bei n Elementen. */
```

Vergleich der Möglichkeiten der Priority Queue:

	Min	DeleteMin	Insert	Element finden
Heap	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n)$
Array oder Liste	$O(1)$	$O(n)$	$O(1)$	$O(1)$ (Array), $O(n)$ (Liste)

Heap als Array $Q[1, \ldots, n]$



Prozedur Vater(Q, i)

```
   /* i ist Index aus 1,...,n */
1  if i ≠ 1 then
2  |  return ⌊i/2⌋;
3  else
4  |  return i;
5  end
```

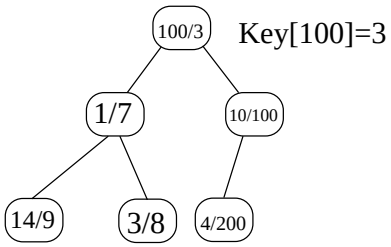
Prozedur linker Sohn(Q, i)

```
1  return 2i;
```

Prozedur rechter Sohn(Q, i)

```
1  return 2i + 1;
```

Beispiel 7.7: *Ein abschließendes Beispiel.*



$Q[1] = 100$ $key[100] = 3$
$Q[2] = 1$ $key[2] = 7$
$Q[3] = 10$ $key[10] = 100$
$Q[4] = 14$ $key[14] = 9$
$Q[5] = 3$ $key[3] = 3$
$Q[6] = 4$ $key[4] = 200$

. . . wenn die Elemente nur einmal auftreten, kann man key auch direkt als Array implementieren.

Suchen nach Element oder Schlüssel wird nicht unterstützt.

7.6 Algorithmus (Prim mit Q in Heap)

Algorithmus 12: Prim mit Heap

Input : $G = (V, E)$ zusammenhängend, $V = \{1, \ldots, n\}$, Kostenfunktion $K : E \to \mathbb{R}$

Output : F = Menge von Kanten eines minimalen Spannbaumes.

```
1  F = ∅;
2  Wähle einen beliebigen Startknoten s ∈ V;
3  foreach v ∈ Adj[s] do
4  |   key[v] = K({s, v});
5  |   kante[v] = s;
6  end
7  foreach v ∈ V \ ({s} ∪ Adj[s]) do
8  |   key[v] = ∞;
9  |   kante[v] = nil;
10 end
11 Füge alle Knoten v ∈ V \ {s} mit Schlüsselwerten key[v] in Heap Q ein;
12 while Q ≠ ∅ do
13 |   w = Min(Q);
14 |   DeleteMin(Q);
15 |   F = F ∪ {{kante[w], w}};
16 |   foreach u ∈ (Adj[w] ∩ Q) do
17 |   |   if K({u, w}) < key[u] then
18 |   |   |   key[u] = K({u, w});
19 |   |   |   kante[u] = w;
20 |   |   |   Q anpassen;
21 |   |   end
22 |   end
23 end
```

Korrektheit mit zusätzlicher Invariante. Für alle $w \in Q_l$ gilt:

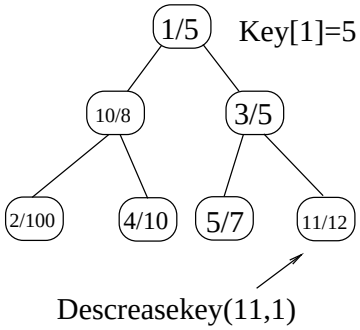
$$\begin{aligned} key_l[w] &= \text{minimale Kosten einer Kante } \{v, w\} \text{ mit } v \notin Q_l \\ key_l[w] &= \infty, \text{ wenn eine solche Kante nicht existiert.} \end{aligned}$$

Laufzeit:

1. - 11.	$O(n) + O(n \cdot logn)$ für das Füllen von Q. (Füllen von Q in $O(n)$ – interessante Übungsaufgabe)
12. - 22.	$n - 1$ Läufe
13. - 15.	Einmal $O(logn)$, insgesamt $O(n \cdot \log n)$
16. - 22.	Insgesamt $O(\|E\|) + \|E\|$-mal Anpassen von Q.

Anpassen von Heap Q

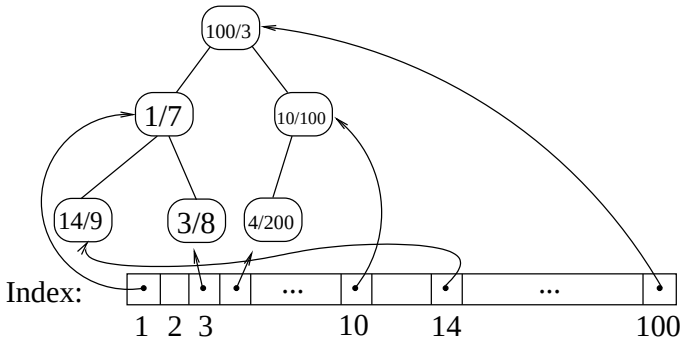
Beispiel 7.8(Operation DecreaseKey(v,s)): $s < key[v]$, *neuer Schlüssel*



Prozedur DecreaseKey(Q, v, s)

```
1 Finde Element v in Q;        /* Wird von Heap nicht unterstützt! */
2 key[v] = s;
3 while key[Vater von v] > key[v] do
4 |   Tausche v mit Vater;
5 end
  /* Laufzeit: O(n) + O(log n) */
```

zum Finden: „Index drüberlegen“



Direkte Adressen:

$$Index[1] = 2, Index[3] = 5, Index[4] = 6, \ldots, Index[100] = 1$$

Mit direkten Adressen geht finden in $O(1)$. Das $Index$-Array muss beim Vertauschen mit aktualisiert werden. Das geht auch in $O(1)$ für jeden Tauschvorgang.

Falls die Grundmenge zu groß ist, kann man einen Suchbaum verwenden. Damit geht das Finden dann in $O(\log n)$.

Falls direkte Adressen dabei: Einmaliges Anpassen von Q (und Index) $O(\log |V|)$. Dann hat Prim insgesamt die gleiche Laufzeit, $O(|E| \log |V|)$, wie Kruskal mit Union-by-Size. Beachte vorher $O(|E| \cdot |V|)$.