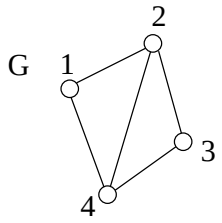
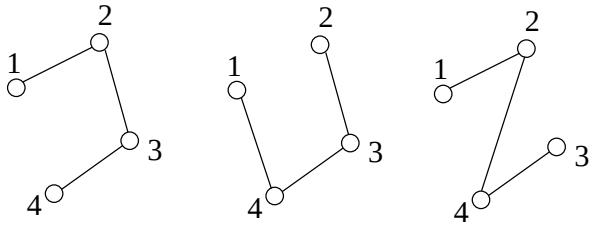
7 Minimaler Spannbaum und Datenstrukturen

Hier betrachten wir als Ausgangspunkt stets zusammenhängende, ungerichtete Graphen.



So sind



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 $f \in FH \setminus \{f\} = (V, F \setminus \{f\})$ nicht mehr zusammenhängend ist (H ist maximal zusammenhängend).*

Folgerung 7.1: *Sei H zusammenhängend, Teilgraph von G.*
H Spannbaum von $G \iff H$ 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 $\neg B \wedge \neg A$)
„$\Leftarrow$ “ 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 $\neg A \wedge \neg B$) □

Damit hat jeder Spannbaum genau $n-1$ Kanten, wenn $|V| = n$ ist. Vergleiche Seite 34. Die Baumkanten einer Suche 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 $\binom{m}{n-1} = \frac{m!}{(n-1)! \cdot \underbrace{(m-1+1)!}_{\geq 0 \text{ für } n-1 \leq m}}$

Wieviel ist das? $m = 2 \cdot n$, dann
$\frac{(2n)!}{(n-1)!(n+1)!} = \frac{2n(2n-1) \cdot (2n-2) \cdot \ldots \cdot n}{(n+1)n(n-1)\ldots 1}$

$\geq \underbrace{\frac{2n-2}{n-1}}_{=2} \cdot \underbrace{\frac{2n-3}{n-1}}_{>2} \cdot \ldots \cdot \underbrace{\frac{n}{1}}_{>2}$

$\geq \underbrace{2 \quad 2 \ldots 2}_{n-1-\text{mal}} = 2^{n-1}$

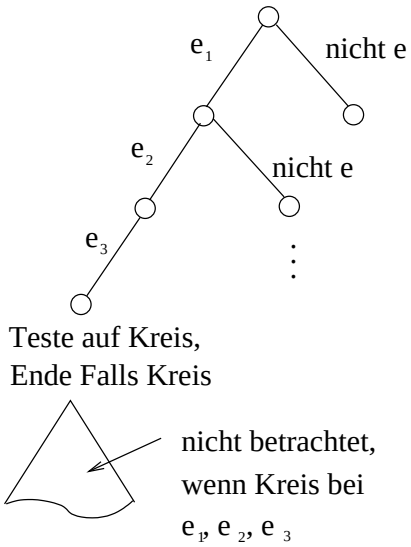
Also Zeit $\Omega(2^n)$.

Regel: $c \geq 0$
$\frac{a}{b} \leq \frac{a-c}{b-c}$
$\Leftrightarrow b \leq a.$
Das c im Nenner hat mehr Gewicht.

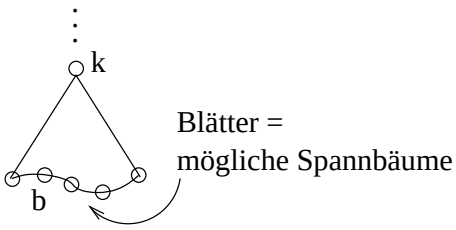
Definition 7.3(Ω-Notation): *Ist $f, g : \mathbb{N} \to \mathbb{R}$, so ist $f(u) = \Omega(g(u))$ genau dann, wenn es eine Konstante $C > 0$ gibt, mit $|f(n)| \geq C - g(n)$ für alle hinreichend großen n. (vergleiche O-Notation)*

O-Notation: Obere Schranke
Ω-Notation: Untere Schranke.

Verbesserung: Backtracking, frühzeitiges Erkennen von Kreisen: 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 $k = \sum_{f \text{ in k} \atop \text{gewählt}} K(f)$.



Es ist für Blätter wie b unter k.
Kosten* von $k \leq$ Kosten von b.
*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.

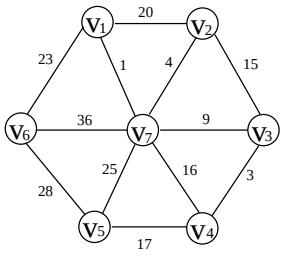


Abbildung 3: ungerichteter Graph mit Kosten an den Kanten

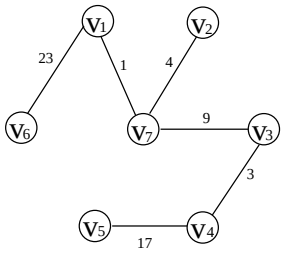
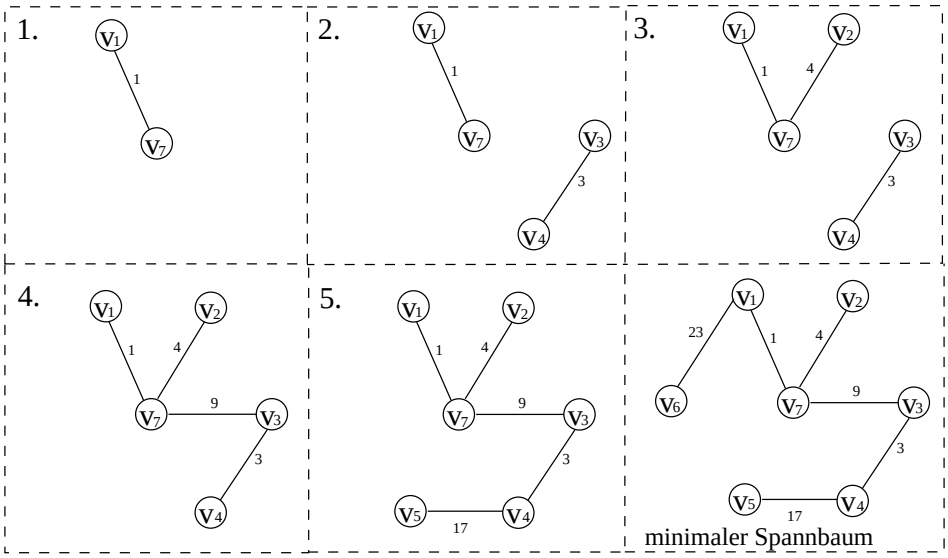


Abbildung 4: Spannbaum mit minimalen Kosten

Aufbau eines minimalen Spannbaums siehe Abbildung 5



Nimm die günstigste Kante, die keinen Kreis ergibt, also nicht

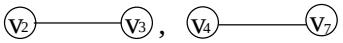


Abbildung 5: schrittweiser Aufbau eines Spannbaumes

Implementierung durch Partition von V

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

Problem: Wie Partition darstellen?

7.1 Algorithmus Minimaler Spannbaum

(Kruskal 1956) Eingabe:

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

Ausgabe:

F = Mege von Kanten eines minimalen Spannbaumes.

```
1. F = ∅, P = {{1}, ..., {n}} // P ist die Partition
2. E nach Kosten sortieren    // geht in O(E log |E|)=O(|E| log|V|),
                              // E≤V², log|E|=Olog|V|
3. while(|P|≥1){              // solange P≠{{1, ..., n}}
4.    {v, w} = kleinstes (erstes) Element von E
      {v, w} aus E löschen
5.    Testen, ob F ∪ {v, w} einen Kreis hat
6.    if({v, w} induziert keinen Kreis){
         W_v = die Menge mit v aus P;
         W_w = die Menge mit w aus P;
         W_v und W_w in P vereinigen
         F = F ∪ {{v, w}}
      }
   }
```

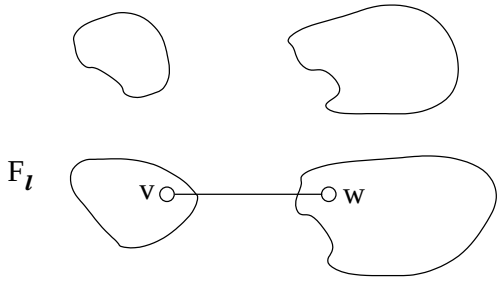
□

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:
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.
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. 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 ghört z 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:

1. $O(n)$
2. Kanten sortieren: $O(|E| \cdot log|E|)$ (wird später behandelt), also $O(|E| \cdot log|V|)$!

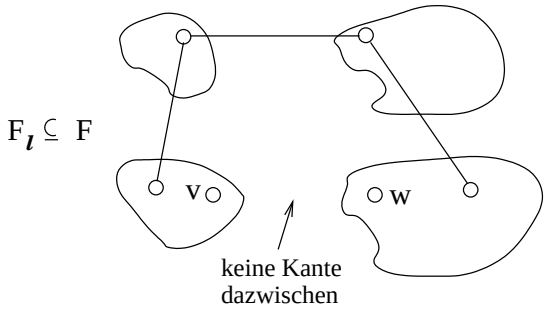
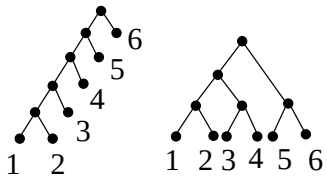


Abbildung 6: ein Beispiel für F

3. 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.

Zeit hängt an der Darstellung von P.
Einfache Darstellung: $arrayP[1,\ldots,n]$, 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 ... 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:

```
1. a = P[v]; b = P[w]          // a+b, wenn W_u ≠ W_v
2. for i = 1, ..., n{
      if (P[i] = a){
         P[i] = b
      }
   }
```

Damit Laufzeit der Schleife:
7. insgesamt $n \cdot O(n) = O(n^2)$ (Es wird $n-1$-mal vereinigt.)
4. und 5. insgesamt $O(|E|)$
3. $O(n)$ insgesamt, wenn $|P|$ mitgeführt.
Also $O(n^2)$, wenn E bereits sortiert ist.
Darstellung von P durch eine Union-Find-Struktur:

- Darstellung einer Partition P einer Grundmenge (hier klein, d.h. als Indexmenge verwendbar)
- Für $U, W \in P$

  $$\text{Union}(U, W)$$

  soll $U, W \in P$ durch $U \cup W \in P$ ersetzen
- Für u aus der Grundmenge

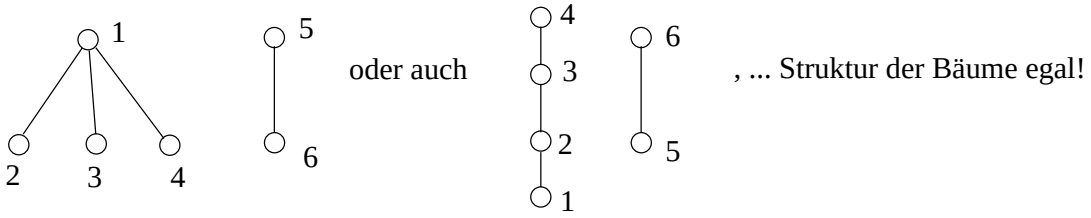
  $$\text{Find(u)}$$

  soll Namen der Menge $U \in P$ mit $u \in U$ geben. (Also es geht nicht ums Finden von u!)

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

7.2 Datenstruktur Union-Find-Struktur

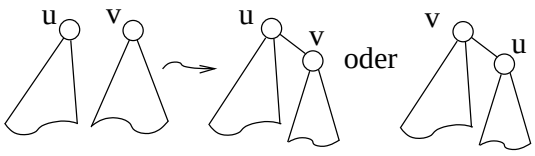
(a) Jede Menge von P als ein Baum mit Wurzel.
$P = \{\{1, 2, 3, 4\}, \{5, 6\}\}$, dann etwa Wurzel = Name der Menge.



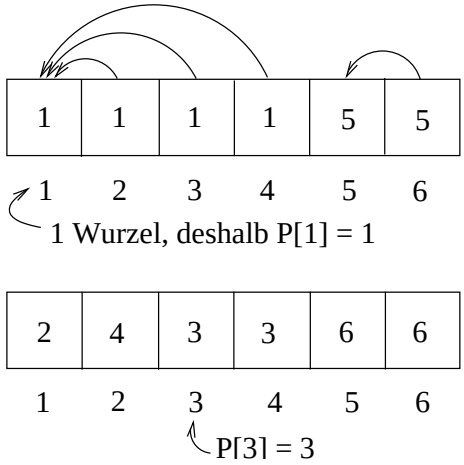
(b)

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)



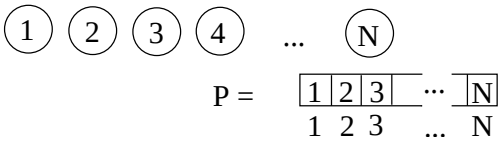
(c) Bäume in array(Vaterarray): $P[1,\ldots,n]$ of $1\ldots n$
$P[v]$ = Vater von v.
Aus (a) weiter:



Wieder wichtig: Indices = Elemente.
Vaterarray für beliebige Bäume

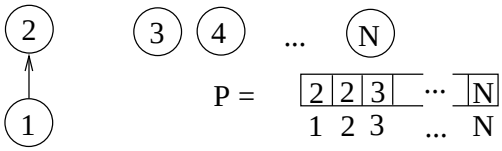
```
Union (u,v)                          //u, v Wurzeln
  P[u]=v;                            // u hängt unter v
Find(v)
  while (P[v]≠v){                    // O(n) im worst case
    v=P[v];
  }
  Ausgabe von v.
```

1.



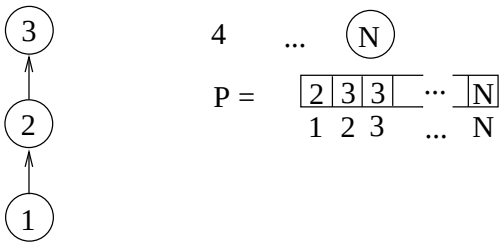
↷ Union(1,2) in O(1)

2.



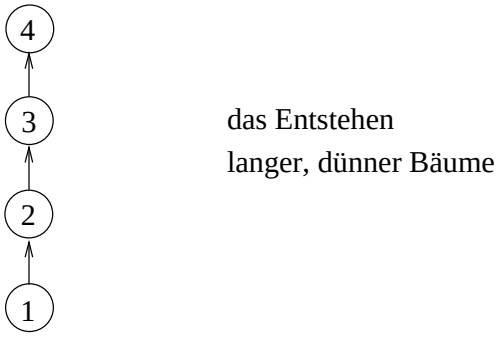
↷ Union(2,3) in O(1)

3.



↷ Union(3,4) in O(1)

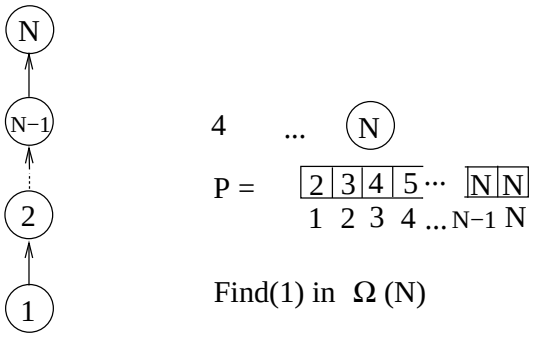
4.



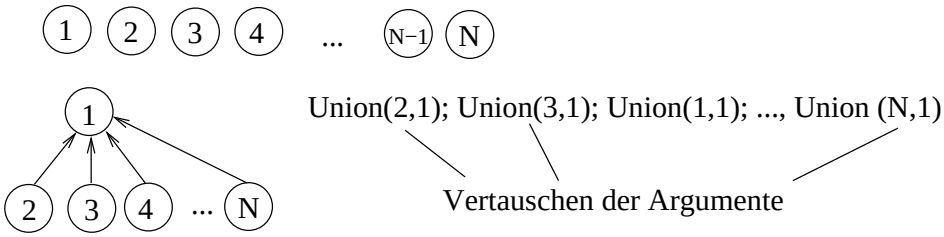
das Entstehen
langer, dünner Bäume

⋮

N.



Find(1) in Ω (N)



Find(2) in O(1).

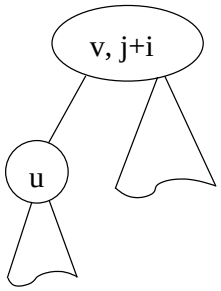
⇒ Union by Size; kleinere Anzahl nach unten.
⇒ maximale Tiefe (längster Weg von Wurzel zu Blatt) = $log_2 N$

Beachte:

$log_2 N$ zu $N = 2^{log_2 N}$ wie N zu 2^N.

7.3 Algorithmus Union-by-Size

```
Union((u,i), (v,j))          // i, j # Elemente in den Mengen.
                             // Muss mitgeführt werden
if (i≤j){
```




```
}
else {„umgekehrt\}
```

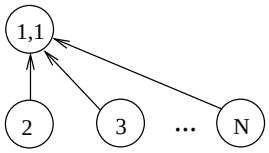
Union by Size:

↷ Union(2,1)

↷ Union(1,3)

⋮

↷ Union(1,N)



Tiefe Bäume durch Union-by-Size?

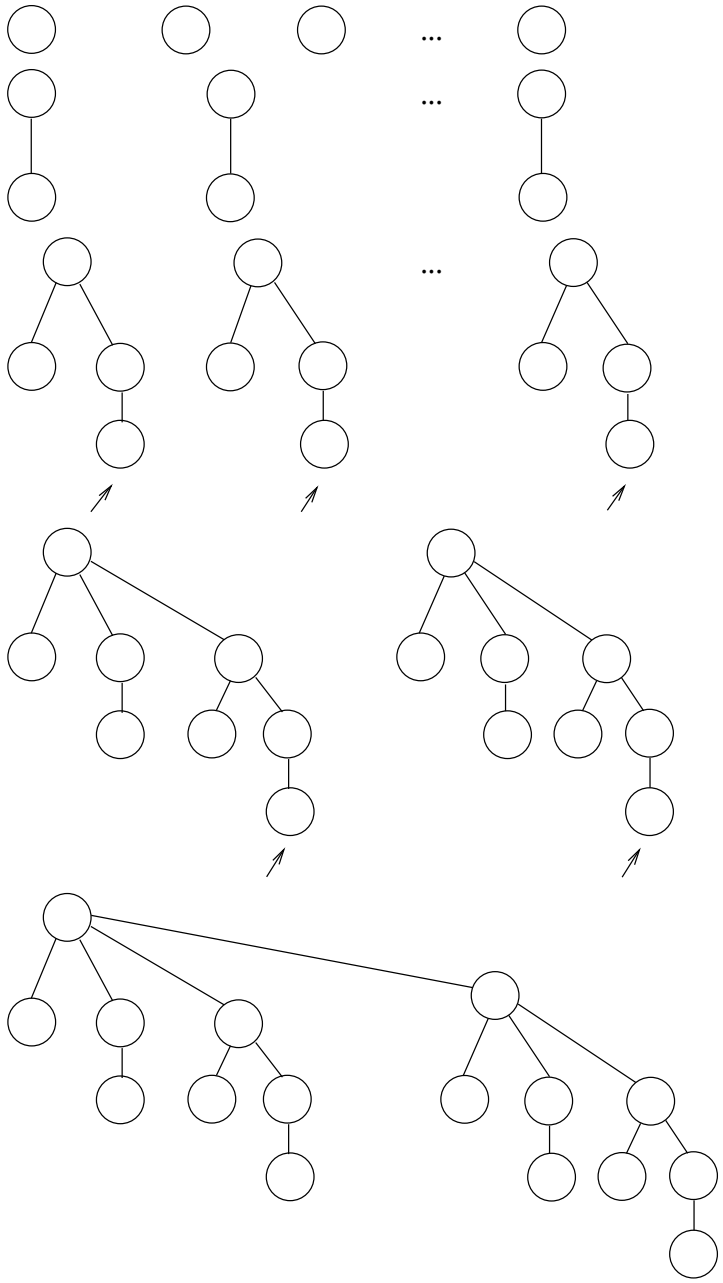
1.

2.

3.

4.

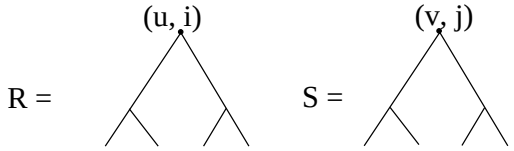
5.



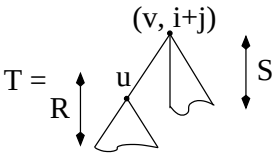
Vermutung: N Elemente ⇒ Tiefe $\leq log_2 N$.

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 # 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:

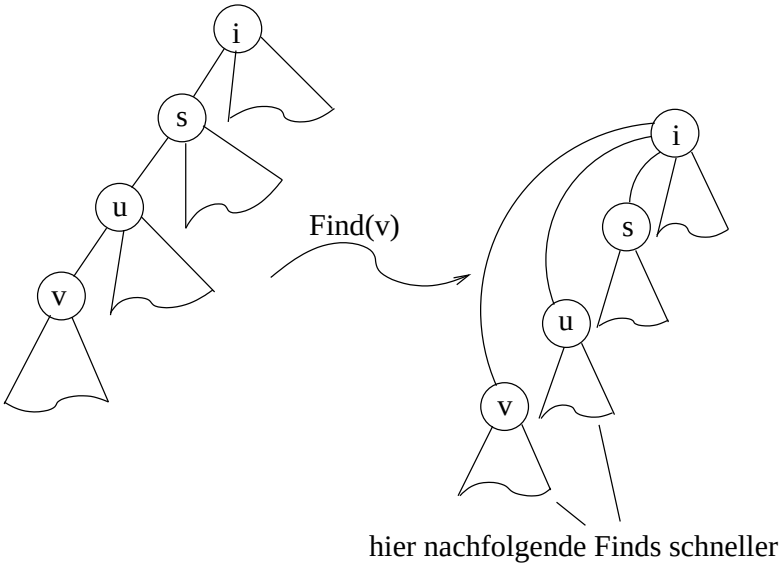


T durch Union (u,v) und

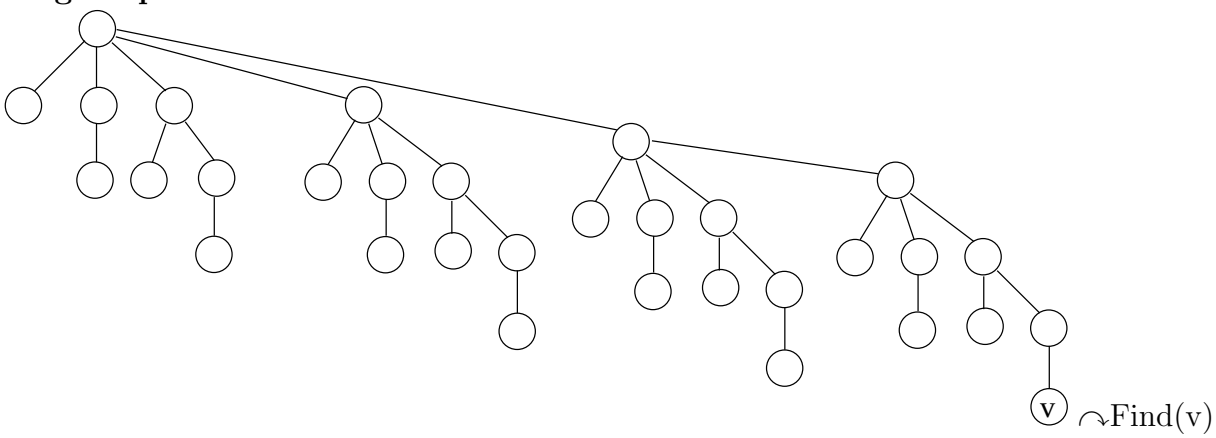


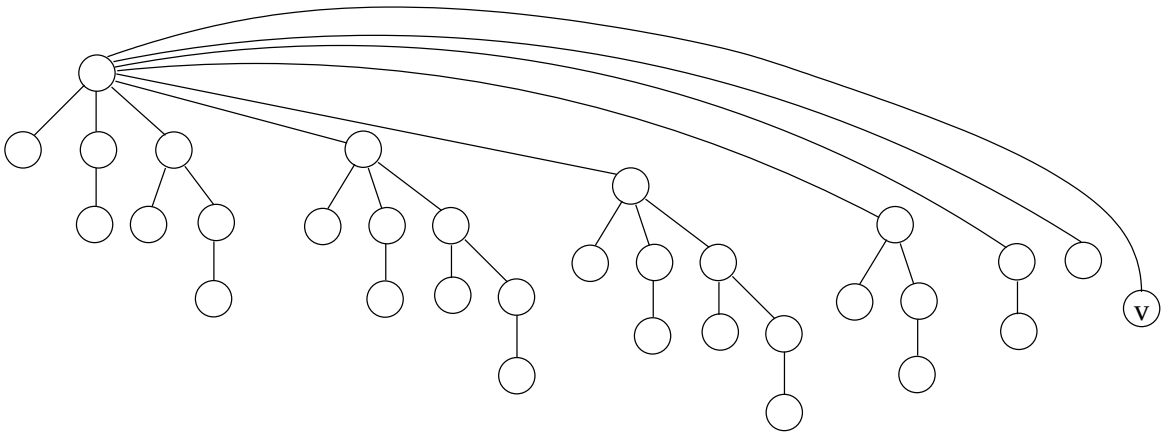
Sei $i \leq j$, dann
1. Fall: keine größere Tiefe als vorher. Die Behauptung gilt nach Induktionsvoraussetzung, da T mehr Elemente hat erst recht. 2. Fall: Tiefe vergrößert sich echt. Dann aber $Tiefe(T) = Tiefe(R) + 1 \leq (log_2|R|) + 1 = log_2(2|R|) \leq |T|$
Union by Size. ($2^x = |R| \Rightarrow 2 \cdot 2^x = 2^{x+1} = 2|R|$)!
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): $O(|E|log|V|)$ ($log|V|$ ... 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*):



Wegkompression





Im Allgemeinen $\Omega(logN)$ und $O(logN)$

7.4 Algorithmus Wegkompression

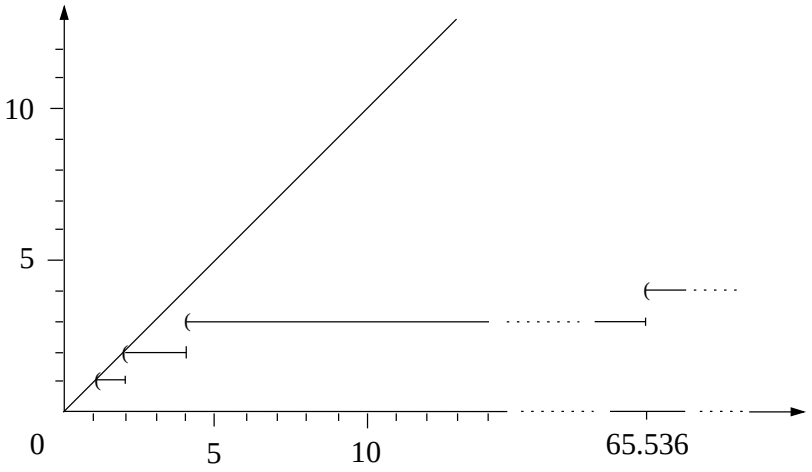
```
Find(v)
1. v auf Keller tun                  // etwa als array implemntieren
2. while (P[v]≠v){                   (wie Schlange)
3.    v = P[v];
      v auf Keller tun
   }
4. v ausgeben;
5. for (w auf dem Keller){
      P[w] = v;                      //Können auch Schlange, Liste oder
   }                                 //sonstwas nehmen, da Reihenfolge egal
```

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^*



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

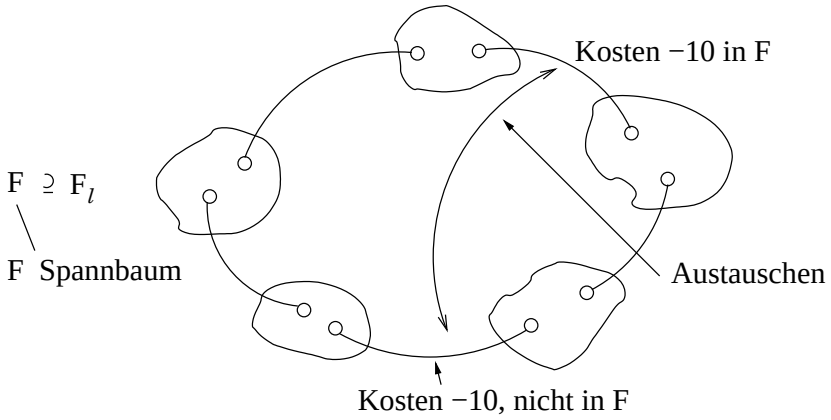
$log^*n - Min\{s|log^{(s)}(n) \leq 1\}$

$log^*2^{(2^{(2^{(2^{(2^2)})})})} = 7$

$log^{(s)}(n) = \underbrace{log(log(\ldots(log}_{s-\text{ mal}}(n))\ldots))$

Kruskal bekommt dann eine Zeit von $O(|V| + |E| \cdot log^*|V|)$
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.

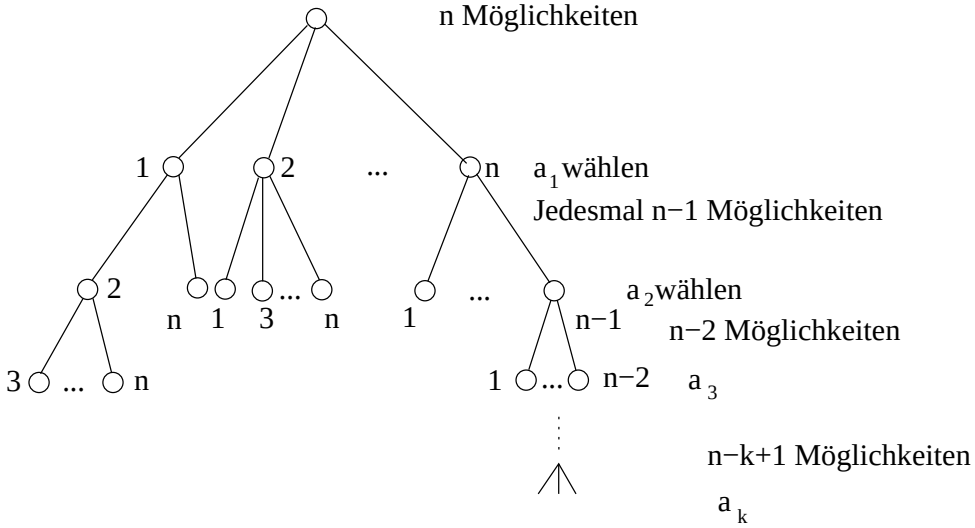


Alle Kanten, die zwei Komponenten verbinden, haben Kosten $\geq$ -10!
Beachte: Negative Kosten sind bei Kruskal kein Problem, nur bei branch-and-bound (vorher vergrößern)!
Binomialkoeffizient $\binom{n}{k}$ für $n \geq k \geq 0$.

$$\binom{n}{k} = \frac{n!}{(n \cdot k) \cdot k!} = \frac{\overbrace{n(n-1) \cdot \ldots \cdot (n-k+1)}^{\text{oberste k Faktoren von n!}}}{k!}$$

$0! = 1! = 1$.
$\binom{n}{k}$ = # Teilmengen von $\{1, \ldots, n\}$ mit genau k Elementen
$\binom{n}{1} = n, \binom{n}{2} = \frac{n(n-1)}{2}, \binom{n}{k} \leq n^k$.

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



Der Baum hat $\underbrace{n(n-1)\cdot\ldots\cdot\overbrace{(n-k+1)}^{n-(k-1)}}_{\text{k Faktoren (nicht k-1, da 0, 1, \ldots, k+1 genau k Zahlen)}} = \frac{n!}{(n-k)!}$ Jedes Blatt $\Longleftrightarrow$ genau eine Folge $(a_1, \ldots, a_k)$
Jede Menge aus k Elementen kommt $k!$-mal vor: $\{a_1, \ldots, a_k\}$ ungeordnet, geordnet als $(a_1, \ldots, a_k), (a_2, a_1, \ldots, a_k), \ldots (a_k, a_{k-1}, \ldots, a_2, a_1) k!$ Permutationen.
Also $\frac{n!}{(n-k)!k!} = \binom{n}{k}$ Mengen mit k verschiedenen Elementen. □

7.5 Algorithmus Minimaler Spannbaum nach Prim 1963

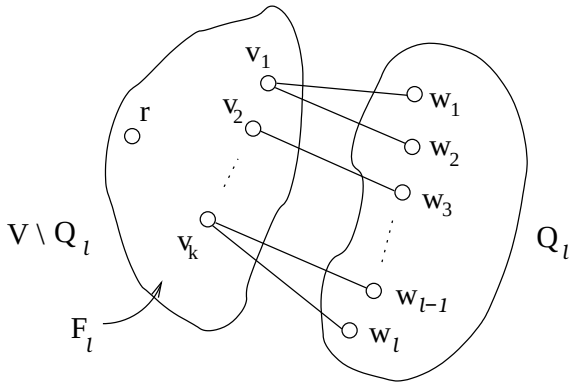
Eingabe: Wie bei Kruskal
Ausgabe: Menge von Kanten F eines minimalen Spannbaumes

```
1. Wähle irgendeinen Startknoten r
   Q = V\{r}                          // Q enthält immer die Knoten,
   F = ∅                              // die noch bearbeitet werden müssen.
2. while (Q ≠ ∅ {
3.    M = {{v, w}| v ∈ V\ Q, w ∈ Q}
                                      // M = Menge der Kanten mit
                                      // genau einem Knoten in Q
4.    {v, w} = eine Kante von minimalen Kosten in M
      F = F ∪{{v, w}};
      Q = Q ohne den einen Knoten aus {v, w}, der auch zu Q gehört
```

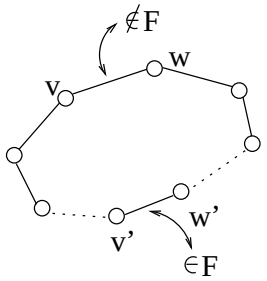
Korrektheit mit der Invariante:

- Es gibt einen minimalen Spannbaum $F \supseteq F_l$
- $Q_l =$
  $l = 0$ ✓.

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 $K(\{v, w\} = K(\{v', w'\})$.
Können in F die Kante $\{v', w'\}$ durch $\{v, w\}$ ersetzen und haben immernoch einen minimalen Spannbaum. Damit Invariante für l+1.
Öaufzeit von Prim:

- $n - 1$ Läufe durch 2.
- 3. und 4. einmal $O(|E|)$ reicht sicherlich, da $|E| \geq |V| - 1$.

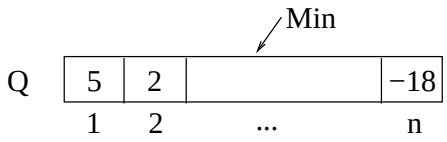
Also $O(|V| \cdot |E|)$, bis $O(n^3)$ bei $|V| = n$.
Bessere Laufzeit durch bessere Verwaltung von Q.

- Maßgeblich dafür, ob $w \in Q$ in den Baum aufgenommen wird, sind die minimalen Kosten einer Kante(!) $\{v, w\}$ für $v \notin Q$.
- Also array key[1 ... n] of real mit der Intention: Für alle $w \in Q$ ist key[w] = min. Kosten einer Kante $\{v, w\}$, wobei $v \notin Q$. (key[w] = ∞, gdw. keine solche Kante existiert.)
- Außerdem array ka[1, ..., n] of {1, ..., n} mit: Für alle $w \in Q$
  ka[w] = v $\iff$ $\{v, w\}$ ist eine Kante minimaler Kosten mit $v \notin Q$

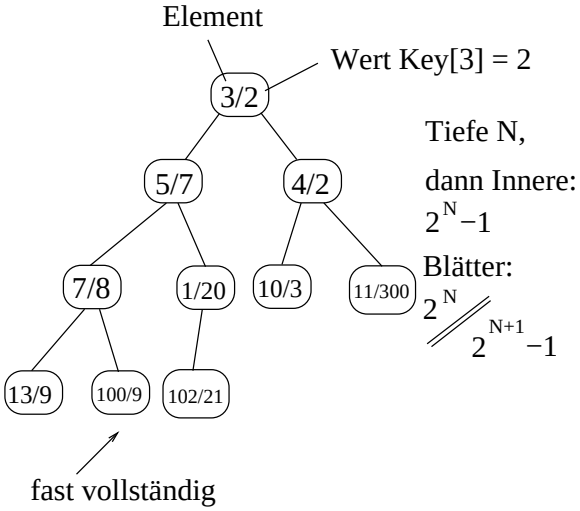
Wir werden Q mit einer Datenstruktur für die *priority queue* (Vorrangwartesschlange) implementieren:

- 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

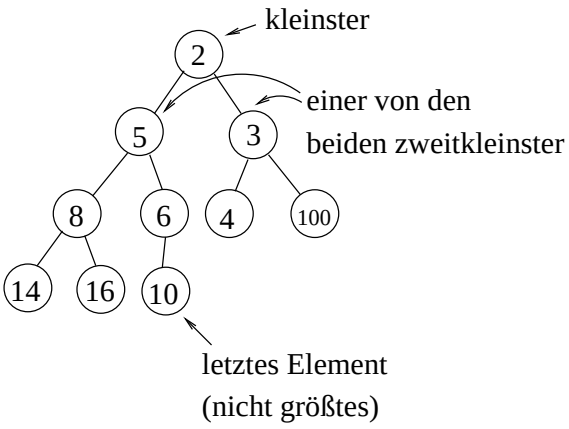


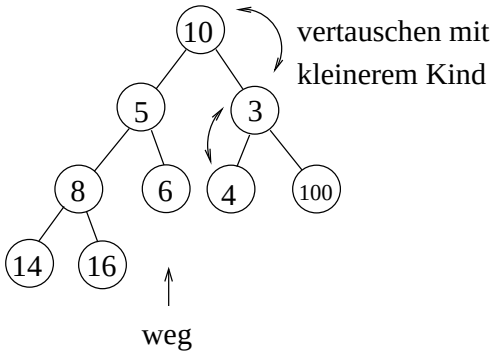
Indices = Elemente,
Einträge in Q = Schlüsselwerte, Sonderwert (etwa -∞) bei „nicht vorhanden“.
Zeiten:
Insert(v,s) in $O(1)$ (sofern keine gleichen Elemente mehrfach auftreten)
Min in $O(1)$
DeleteMin - $O(1)$ fürs Finden des zu löschenden Elementes, dann aber $O(n)$, um ein neues Minimum zu ermitteln.
2. Möglichkeit:
Darstellung als Heap.
Heap = „fast vollständiger“ binärer Baum, dessen Elemente (= Knoten) bezüglich Funktionswerten key[j] nach oben hin kleiner werden.

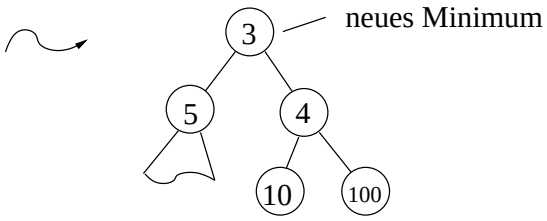


Heapeigenschaft: für alle u, v, u Vorfahr von $v \Longrightarrow key[u] \leq key[v]$

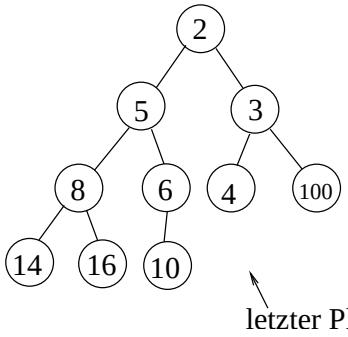
Beispiel Minimum löschen:
Nur Werte: Key[I]:

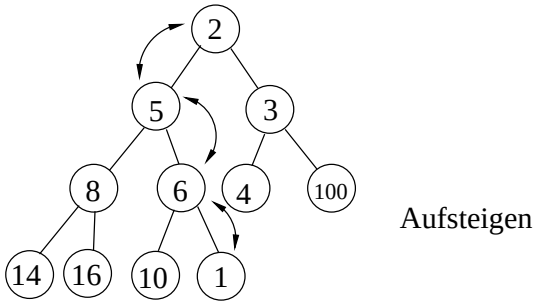


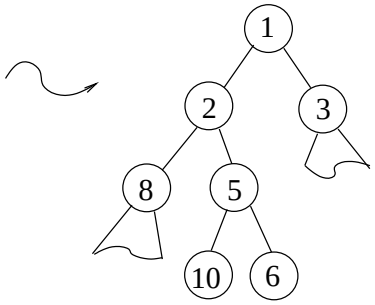




Beispiel Einfügen
Element mit Wert 1 einfügen



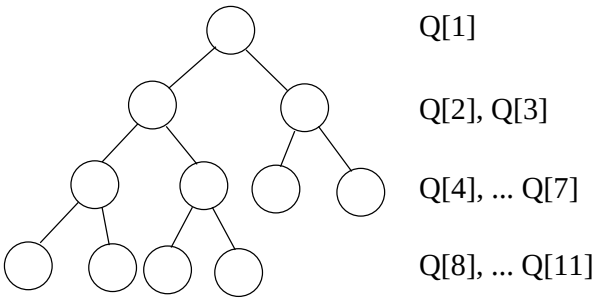


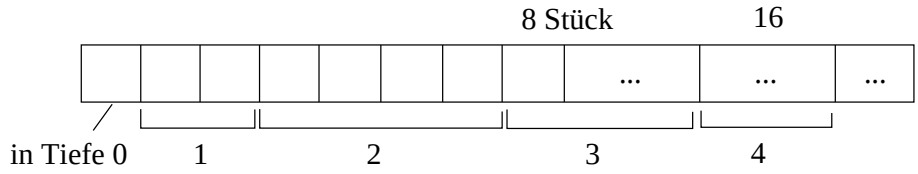


Priority Queue mit Heap, sagen wir Q.
Min {
1. Gib Element der Wurzel aus
} O(1)
Deletemin{
1. Nimm „letztes“ Element, setze es auf Wurzel x:=Wurzel(-Element)
2. While Key[x]≥Key[linker Sohn von x] oder rechter Sohn
{tausche x mit kleinerem Sohn}
}
N Elemente, $O(logN)$, da maximal $O(logN)$ Durchläufe
Insert(v,s){//Key[v]=s.
1. Tue v als letztes Element in den Heap.
2. While Key[Vater von v]¿Key[v]
{vertausche v mit seinem Vater}
}
N Elemente $O(logN)$
Vergleich der Möglichkeiten der Priority Qeue

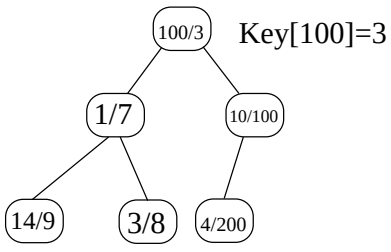
	Min	Deletemin	Insert	Element finden
Heap	1	log N	logN	N
Array o. Liste	1	N(!)	1	1(Array), N(Liste)

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





Vater I{// I Index aus 1, ..., N
if (I+1){
Gebe $\lfloor \frac{1}{2} \rfloor$ aus}
Linker Sohn I 2I
Rechter Sohn 2I+1
Abschließendes Beispiel



Array Q
Q[1] = 100 Key[100] = 3
Q[2] = 1 Key[2] = 7
Q[3] = 10 Key[10 = 100
Q[4] = 14
Q[5] = 3
Q[6] = 4
$\vdots$
wenn Elemente nur einmal

Suchen nach Element oder Schlüssel wird nicht (!) unterstützt.

7.6 Algorithmus (Prim mit Q in heap)

```
1. Wähle Startknoten r.
   for each v∈ Adj[r]{
     key[v]=k({r,v});
     kante[v]=r}
   Für alle übrigen v∈V{
     key[v]=∞; kante[v]=∞}
   Füge V\{r} mit Schlüsselwerten key[v] in heap Q ein.
2. while Q≠ ∅{
3.    w = Min; DeleteMin_Q;
      F = F∪{{kante[w],w}}
4. for each u∈ Adj[w]∩ Q{
      if K({u,w})<key[u]{
        key[u] = k({u,w});
        kante[u] = w;
        Q anpassen}
      }
   }
```

Korrektheit mit zusätzlicher Invariante:

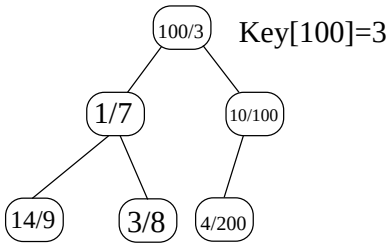
Für alle $w \in Q_l$

key$_l$[w] = minimale Kosten einer Kante {v, w} für $v \notin Q_l$}
key$_l$[w] = ∞, wenn eine solche Kante nicht existiert.

Laufzeit

1. $O(n) + O(n \cdot logn)$ für das Füllen von Q.
(Füllen von Q in $O(n)$ - interessante Übungsaufgabe)
2. $n - 1$ Läufe
3. Einmal $O(logn)$, insgesamt $O(n \cdot logn)$
4. Insgesamt $O(|E|) + |E|$-mal Anpassen von Q.

Anpassen von Heap Q
Operation Decreasekey(v,s)
$s < key[v]$, neuer Schlüssel



Decreasekey(v,s)

1. Finde Element v in Q //nicht von Q unterstützt!
2. while key[Vater von v]¿s tausche v mit Vater

Laufzeit Deckey(v,s):
2. $O(logN)$ 1. $O(N)$(!!).
zum Finden: „Index drüberlegen“
Direkte Adressen:
Ind[1] = 3, Ind[3] = 5, Ind[4]= 6, Ind[100] = 1
Finden in $O(1)$
Falls Grundmenge zu groß, dann Suchbaum: Finden $O(logN)$
Falls direkte Adressen dabei:
Einmaliges Anpassen von Q (und Index) $O(log|V|)$
Dann Prim insgesamt $O(|E|log|V|)$ (Beachte vorher $O(|E| \cdot |V|)$)
(wie Kruskal mit Union by Size.)