Teil 2:
Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
  - Problemstellung
  - Algorithmus von Prim
  - Algorithmus von Kruskal
  - Vereinigung und Suchen von disjunkten Mengen (Disjoint Set Union-Find)
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Problemstellung (1)

Definition

Sei G = (V,E) ein ungerichteter, gewichteter Graph.

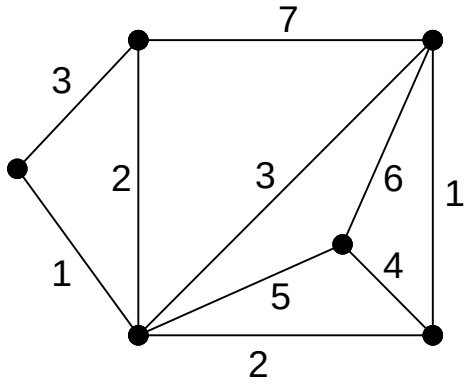
Dann heißt B ein minimal aufspannender Baum, falls B folgende Eigenschaften erfüllt:

(1) B = (V,E‘) mit E‘ ⊆ E;
d.h. B ist ein Teilgraph von G mit gleicher Knotenmenge

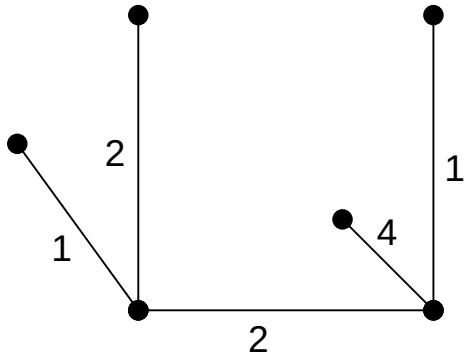
(2) B ist ein Baum;
d.h. ein azyklischer, ungerichteter Graph

(3) Die Summe der Kantengewichte von B ist minimal;
d.h. es gibt keinen anderen Baum, der Eigenschaften (1) und (2) erfüllt und eine kleinere Kantengewichtssumme hat.

Beispiel



Ungerichteter Graph G

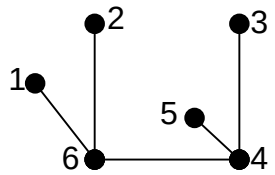


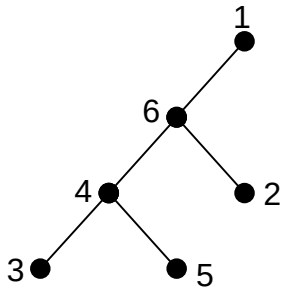
Minimal aufspannender Baum von G

Problemstellung (2)

Bemerkung

- Man beachte, dass ein azyklischer (d.h. kreisloser) ungerichteter Graph ein Baum ist: man nehme einfach einen beliebigen Knoten als Wurzel und „verschiebe die anderen Knoten nach unten“.





- Man mache sich die Bedeutung von aufspannend klar:
  Da ein minimal aufspannender Baum B zu einem Graphen G alle Knoten von G enthält und außerdem ein Baum zusammenhängend ist, spannt B den Graphen G auf.
- Der minimal aufspannende Baum muss nicht eindeutig sein.

Typische Anwendung

Zwischen n Orten soll ein Versorgungsnetz (Kommunikation, Strom, Wasser, etc.) aufgebaut werden, so dass je 2 Orte direkt oder indirekt miteinander verbunden sind. Die Verbindungskosten zwischen 2 Orte seien bekannt.

Gesucht ist ein Versorgungsnetz mit den geringsten Kosten.

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
  - Problemstellung
  - Algorithmus von Prim
  - Algorithmus von Kruskal
  - Vereinigung und Suchen von disjunkten Mengen (Disjoint Set Union-Find)
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Algorithmus von Prim (1)

Problem:

Eingabe: Gewichteter Graph

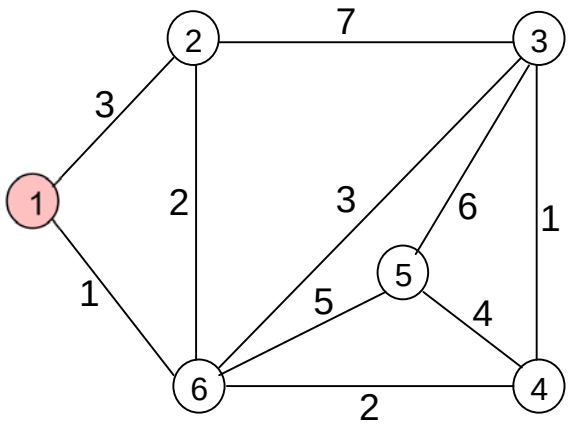
Ausgabe: Minimal aufspannender Baum

Idee

- Der Algorithmus von Prim ist eine leichte Modifikation des Algorithmus von Dijkstra.
- Zu jedem Zeitpunkt bilden die bereits besuchten Knoten einen minimal aufspannenden Baum. Anfangs ist noch kein Knoten besucht. Der bisher gefundene Baum wird wie bei den Algorithmen für kürzeste Wege in einem Vorgängerfeld gehalten.
- Alle Knoten, die als nächstes besucht werden können, werden in einer Kandidatenliste gehalten. Zu Anfang wird mit einem beliebigen Knoten als Kandidat begonnen.
- Von den Kandidaten wird derjenige Knoten als nächster besucht, der über die billigste Kante zu erreichen ist. Beachte: Beim Algorithmus von Djikstra wird immer der Knoten mit der kürzesten Distanz zum Startknoten s besucht.

Algorithmus von Prim (2)

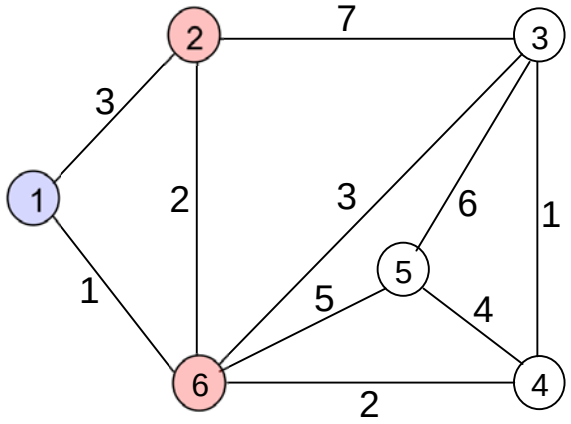
Beispiel:



Zu Beginn ist Knoten 1 einziger Kandidat.

Kandidatenknoten sind rot unterlegt.

Knoten 1 wird besucht



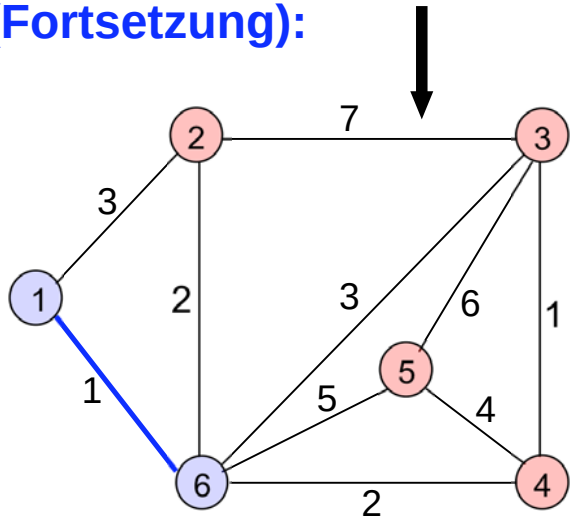
Alle noch nicht besuchten Nachbarn von Knoten 1 werden Kandidaten.

Bereits besuchte Knoten und ausgewählte Kanten sind blau unterlegt und bilden den bisher gefundenen minimal aufspannenden Baum.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	-	-	-	-	-

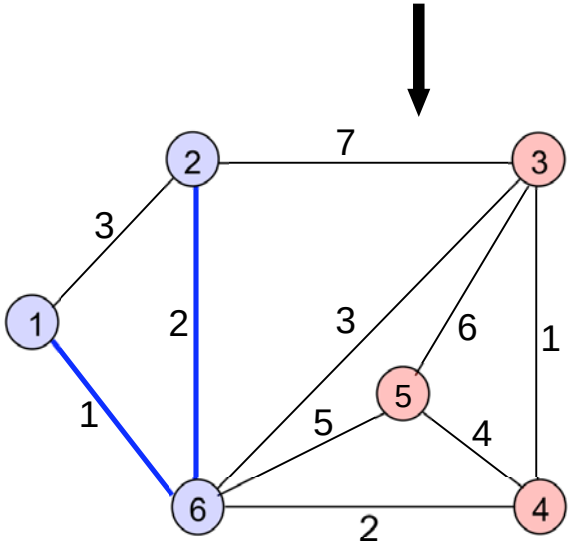
Algorithmus von Prim (3)

Beispiel (Fortsetzung):



Knoten 6 wird besucht, da Knoten 6 über die günstigste Kante zu erreichen.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	-	-	-	-	1



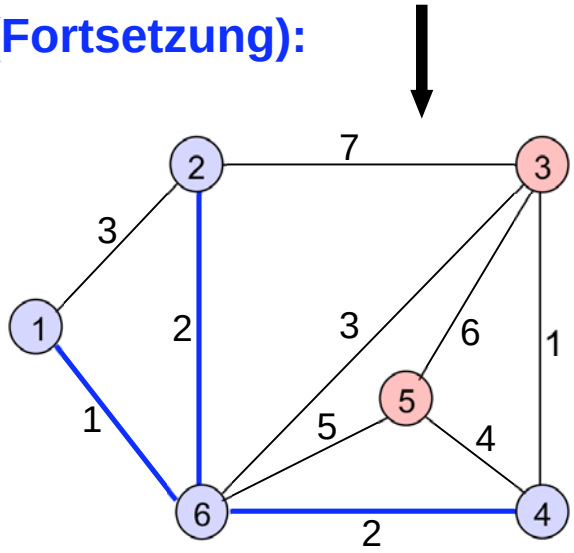
Knoten 2 wird besucht.

(Alternativ hätte man auch Knoten 4 besuchen können.)

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	-	-	-	1

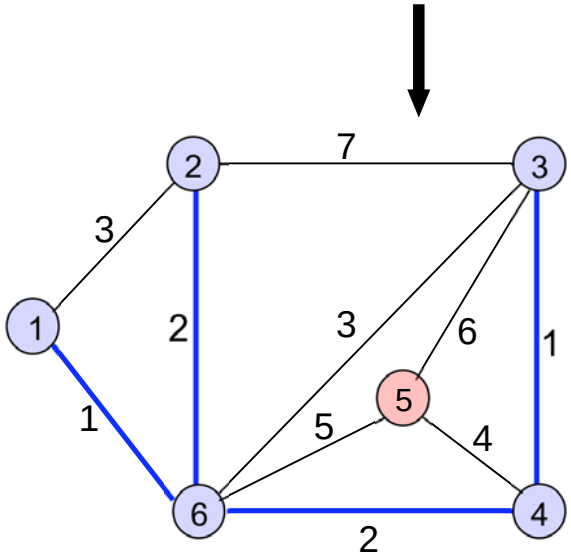
Algorithmus von Prim (4)

Beispiel (Fortsetzung):



Knoten 4 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	4	1

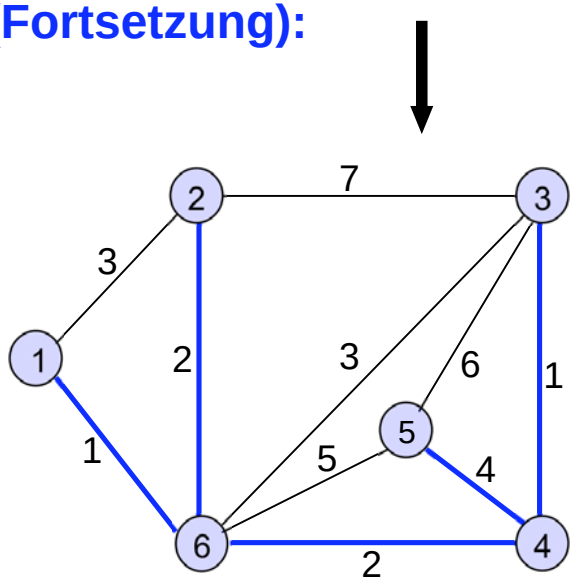


Knoten 3 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	-	1

Algorithmus von Prim (5)

Beispiel (Fortsetzung):



Knoten 5 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	4	1

Fertig, da alle Knoten besucht!

Algorithmus von Prim (6)

```
void minimumSpanningTree(Graph G, Vertex p[ ] )
{
        Set kl;   // Kandidatenliste

        for (jeden Knoten v) {
            d[v] = ∞;
            p[v] = undef;
            imBaum[v] = false;
        }
        s = 1; d[s] = 0;     // Startknoten
        kl.insert(s);

        while (! kl.empty() )
        {
            lösche Knoten v aus kl mit d[v] minimal;
            imBaum[v] = true;  // Knoten v wird im Baum eingebaut
            for (jeden NachbarKnoten w von v)
                if ( ! inBaum[w] )  {
                    if (d[w] == ∞)   // w noch nicht in Kandidatenliste
                        kl.insert(w);
                    if ( c(v,w) < d[w] ) {
                        p[w] = v;
                        d[w] = c(v,w);
                    }
        }
}
```

Eingabe: Graph G.

Ausgabe: Vorgängerfeld p, das den minimal aufspannenden Baum darstellt.

In der Kandidatenliste sind alle Knoten abgespeichert, die als nächstes besucht werden können.

d[v] gibt für jeden Kandidat das Gewicht der billigsten Kante an, mit der er von den bereits besuchten Knoten erreicht werden kann. Für die noch nicht besuchten Knoten, die noch keine Kandidaten sind, ist d[w] = ∞.

d[w] verbessert sich.

Falls nicht alle Knoten besucht wurden, existiert kein aufspannender Baum. Es müsste dann noch eine entsprechende Ausgabe erfolgen.

Algorithmus von Prim (7)

Analyse:

Mit der gleichen Begründung wie beim Algorithmus von Dijkstra gilt:

(a) Kandidatenliste als einfaches (unsortiertes) Feld

$T = O(|V|^2)$

(b) Kandidatenliste als Vorrangswarteschlange

$T = O(|E| \log|V|)$

Fazit:

Ist der Graph dicht besetzt, d.h. $|E| = O(|V|^2)$, dann ist die Variante (a) besser.

Ist der Graph dünn besetzt, d.h. $|E| = O(|V|)$, dann ist die Variante (b) besser.

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
  - Problemstellung
  - Algorithmus von Prim
  - Algorithmus von Kruskal
  - Union-Find-Strukturen (Vereinigung und Suchen von disjunkten Mengen)
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Algorithmus von Kruskal (1)

Problem:

Eingabe: Gewichteter Graph

Ausgabe: Minimal aufspannender Baum

Idee

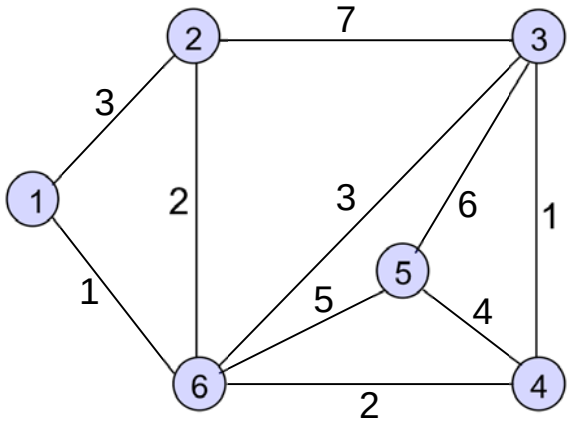
- Es wird ein Wald von minimal aufspannenden Bäumen verwaltet.
  (Wald = Menge von Bäumen)
- Start:
  Zu Anfang besteht der Wald aus allen Bäumen mit jeweils genau einem Knoten.
- Vereinigungsschritt:
  Suche die billigste Kante k, die zwei Bäume aus dem Wald verbindet. Vereinige die 2 Bäume mit k zu einem größeren Baum.
- Es werden solange Vereinigungsschritte durchgeführt, bis nur noch ein Baum übrig bleibt. Das ist dann der minimal aufspannende Baum.

Anmerkung:

Der Wald von Bäumen lässt sich besonders effizient mit einer so genannten Union-Find-Struktur darstellen (siehe nächster Abschnitt).

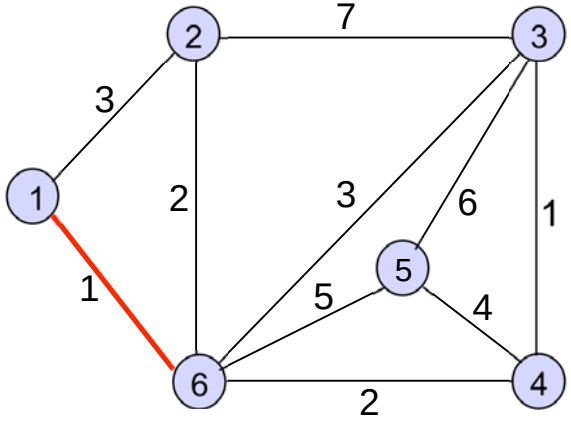
Algorithmus von Kruskal (2)

Beispiel:



Am Anfang besteht der
Wald aus 6 Bäumen
mit jeweils genau einem Knoten.

Kante k = (1,6) mit Gewicht 1 wird gewählt

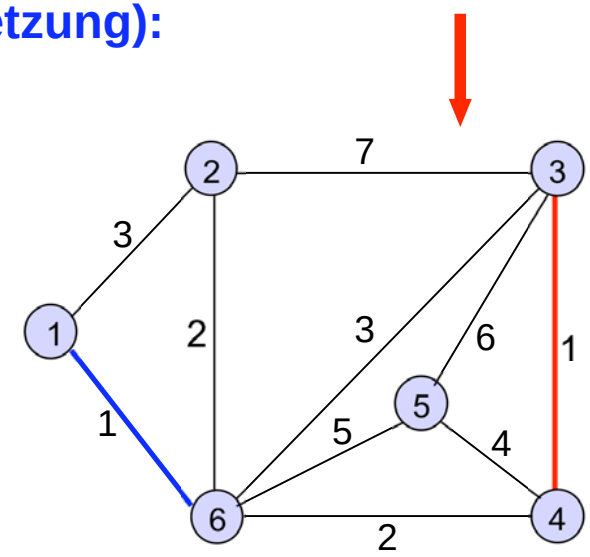


Die beiden Bäumen, die aus dem Knoten 1 bzw. 6 bestehen, werden zu einem Baum vereinigt.

Der **Wald** besteht damit nun aus
5 Bäumen.

Algorithmus von Kruskal (3)

Beispiel (Fortsetzung):

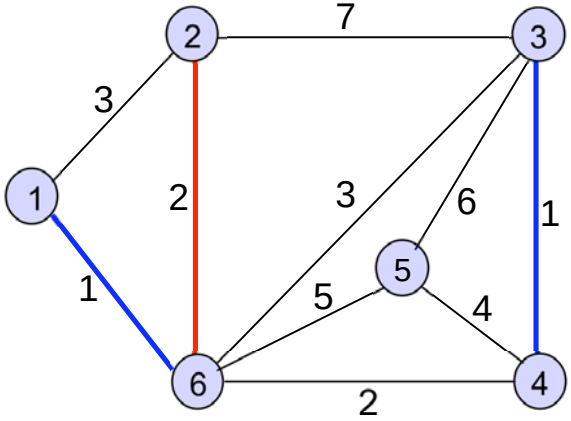


Kante k = (3,4) mit Gewicht 1 wird gewählt

Die beiden Bäumen, die aus dem Knoten 3 bzw. 4 bestehen, werden zu einem Baum vereinigt.

Der **Wald** besteht damit nun aus **4 Bäumen**.

Kante k = (2,6) mit Gewicht 2 wird gewählt

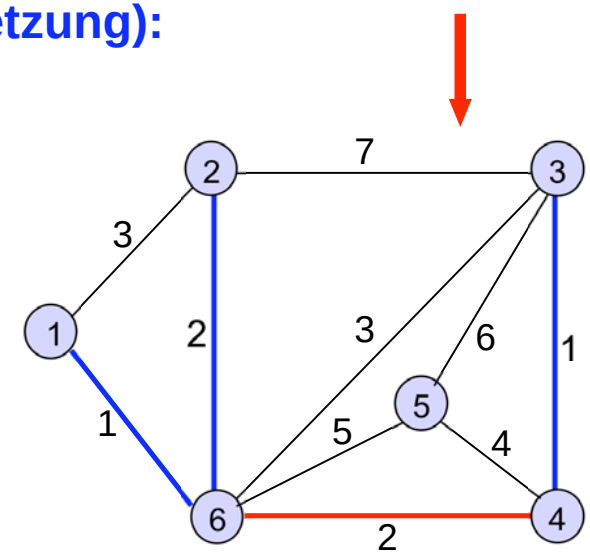


Der Baum mit den Knoten {1, 6} und der Baum mit dem Knoten 2 werden zu einem Baum vereinigt.

Der **Wald** besteht damit nun aus **3 Bäumen**.

Algorithmus von Kruskal (4)

Beispiel (Fortsetzung):

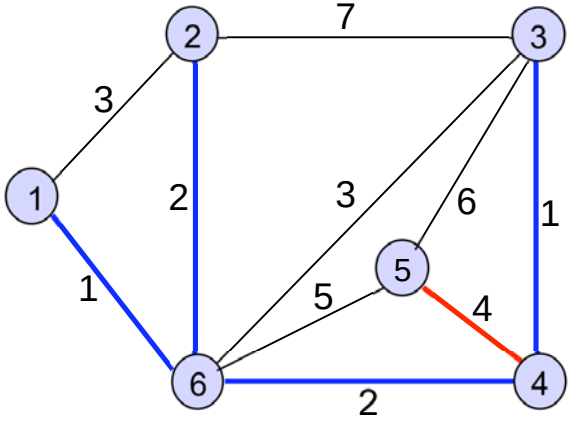


Kante k = (4,6) mit Gewicht 2 wird gewählt

Der Baum mit den Knoten {1,2,6} und der Baum mit den Knoten {3,4} werden zu einem Baum vereinigt.

Der **Wald** besteht damit nun aus **2 Bäumen**.

Kante k = (4,5) mit Gewicht 4 wird gewählt



Der letzte Vereinigungsschritt führt zu einem Wald mit genau einem Baum.

⇒ **Minimal aufspannender Baum.**

Algorithmus von Kruskal (5)

Rückgabe: Minimal aufspannender Baum

Eingabe: Graph G = (V,E).

```
Tree minimumSpanningTree(Graph G)
{
      Sets forest(V);

(1)   PriorityQueue edges(E);

      Tree minSpanTree = ∅;

      while ( forest.number() != 1 && edges != empty ) {
(2)         (v,w) = edges.delMin();
(3)         Set t1 = forest.find(v);
(4)         Set t2 = forest.find(w);
            if (t1 != t2) {
(5)            forest.union(t1,t2);
(6)            minSpanTree.add(v,w);
            }
       }

      if (edges == empty && forest.number() != 1)
            return "es existiert kein aufspannender
Baum";
      else
            return minSpanTree;
}
```

forest ist eine **Menge von Bäumen**, die anfangs aus jeweils genau einem Knoten aus V bestehen.

Alle Kanten werden mit ihren Gewichten in einer **Vorrangswarteschlange** gespeichert, so dass effizient auf die Kante mit dem kleinsten Gewicht zugegriffen werden kann.

Solange der Wald noch mehr als ein Baum enthält und es noch Kanten gibt.

Wähle Kante mit kleinstem Gewicht, die 2 Bäume aus dem Wald verbindet.

forest.find(v) liefert denjenigen Baum zurück, in dem v enthalten ist.

Vereinige die beiden Bäume zu einem Baum.

Algorithmus von Kruskal (6)

Bemerkung:

- Mit dem Datentyp **Sets** lässt sich eine Menge (Kollektion; hier: Wald) von disjunkten Mengen (hier Bäume) effizient verwalten mit den Operationen find und union (siehe nächster Abschnitt über Union-Find-Strukturen)

  Damit:

  - find: O(log n)
  - union: O(1)

- Eine **Vorrangswarteschlange ProrityQueue** läßt sich mit einer Heap-Struktur implementieren. (siehe STL-Container priority_queue oder HeapSort aus Prog 2)

  Damit:

  - Aufbau einer ProrityQueue (Zeile (1)): O(n)
  - delMin (Zeile (2)): O(log n)

- Der **minimal aufpannende Baum minSpanTree** lässt sich durch eine einfache Kanten-Liste realisieren. Damit ist (6) in O(1) durchführbar.

Insgesamt:

Im schlechtesten Fall wird delMin (Zeile (2)) gefolgt von Zeile (3), (4) und evtl. (5) und (6) |E|-mal durchgeführt.

Damit: $T(n) = O(|E|(\log|E| + \log|V|)) = O(|E| \log|V|)$.

Teil 2: Graphenalgorithmen

- Anwendungen
- Definitionen
- Datenstrukturen für Graphen
- Elementare Algorithmen
- Topologisches Sortieren
- Kürzeste Wege
- Minimal aufspannende Bäume
  - Problemstellung
  - Algorithmus von Prim
  - Algorithmus von Kruskal
  - Union-Find-Strukturen (Vereinigung und Suchen von disjunkten Mengen)
- Flüsse in Netzwerken
- Zusammenhangskomponenten

Union-Find-Strukturen (1)

Problemstellung

Effiziente Verwaltung einer **Kollektion (Menge) von disjunkten Mengen**

$col = \{S_0, S_1, ..., S_{n-1}\}, \quad S_i \cap S_j = \emptyset$ für $i \neq j$,

wobei folgende Operationen unterstützt werden:

- **Find:**
  S = col.find(e), liefert diejenige Menge S zurück, in der e enthalten ist.
- **Union:**
  col.union(S1,S2) vereinigt die beiden Mengen S1 und S2 aus der Kollektion zu einer neuen Menge.

Da es sowohl bei Find als auch bei Union gleichgültig ist, wie die Mengen benannt werden, wird als Name der Menge einfach irgendein Element (Repräsentant der Menge) gewählt.

Typische Operation:

Vereinige die Menge, zu der x gehört, mit der Menge, zu der y gehört:

```
col.union( col.find(x), col.find(y))
```

Union-Find-Strukturen (2)

Grundlegende Datenstruktur

Verwalte Mengen als Bäume, wobei Bäume mit einem Vorgänger- oder besser Elternfeld p dargestellt werden.

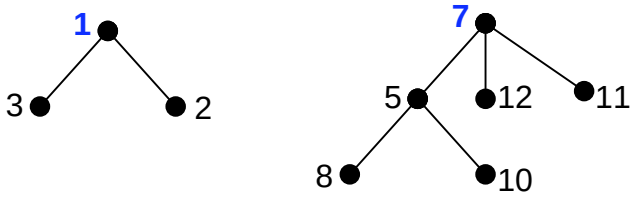
Der Repräsentant der Menge ist die Wurzel.

Beispiel

- **Kollektion von Mengen:**

  col = { {1,2,3}, {5,7,8,10,11,12}, {4}, {6,9} }

- **Bäume:**



Namen (Repräsentant) der jeweiligen Menge

- **Elternfeld:**

e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	**-**	**1**	**1**	**-**	**7**	**-**	**-**	**5**	**6**	**5**	**7**	**7**

p[e] gibt den Elternknoten (Vorgängerknoten) zu p an.

Beachte, dass Wurzeln keinen Elternknoten haben.

Union-Find-Strukturen (3)

Find-Algorithmus

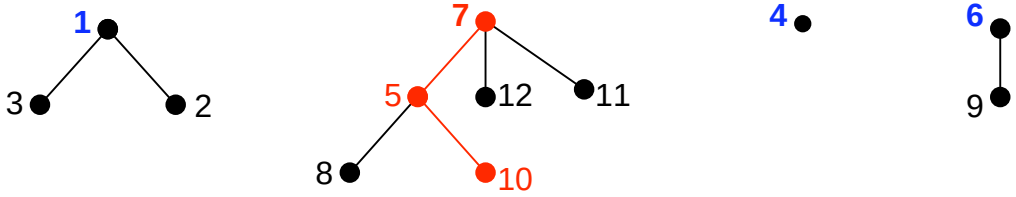
```
int find(int e)
{
        if (p[e] == undef)
                return e;
        else
                return find(p[e]);
}
```

Beispiel

- **Kollektion von Mengen:**

  col = { {1,2,3}, {5,7,8,10,11,12}, {4}, {6,9} }

- **Bäume:**



find(10)
liefert 7 zurück.

- **Elternfeld:**

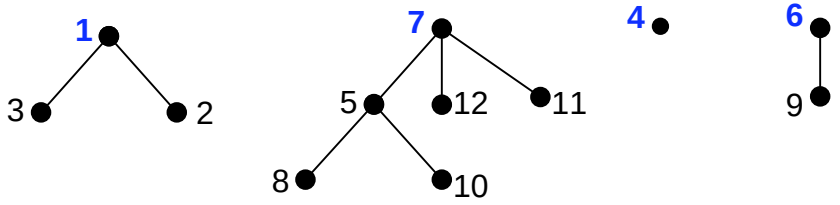
e	1	2	3	4	**5**	6	7	8	9	**10**	11	12
p[e]	-	1	1	-	**7**	-	-	5	6	**5**	7	7

Union-Find-Strukturen (4)

Union-Algorithmus

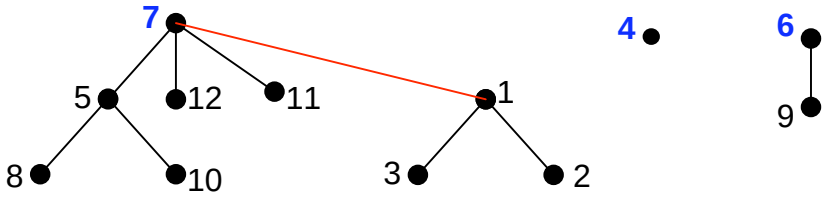
Werden zwei Mengen vereinigt, dann wird der kleinere Baum als Kind der Wurzel des größeren Baumes eingehängt.

Beispiel:



e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	-	1	1	-	7	-	-	5	6	5	7	7

union(1,7) vereinigt die Menge {1,2,3} und die Menge {5,7,8,10,11,12}:



e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	7	1	1	-	7	-	-	5	6	5	7	7

Union-Find-Strukturen (5)

Union-Algorithmus (Fortsetzung)

Es gibt zwei Möglichkeiten festzulegen, welcher Baum kleiner als der andere ist:

- Baum B1 ist kleiner als Baum B2, falls Höhe von B1 kleiner als Höhe von B2
  ⇒ unionByHeight
- Baum B1 ist kleiner als Baum B2, falls Anzahl der Knoten von B1 kleiner als Anzahl der Knoten von B2
  ⇒ unionBySize

Die Höhen- bzw. Größeninformation für ein Baum lässt sich bei p[e], wobei e die Wurzel des Baums ist, als negative Zahl abspeichern.

```
void unionByHeight (int s1, int s2)
{
      if ( -p[s1] < -p[s2] )
            p[s1] = s2;
      else
      {
            if ( -p[s1] == -p[s2] )
                  p[s1]--;
            p[s2] = s1;
      }
}
```

Höhe von s2

Höhe von s1

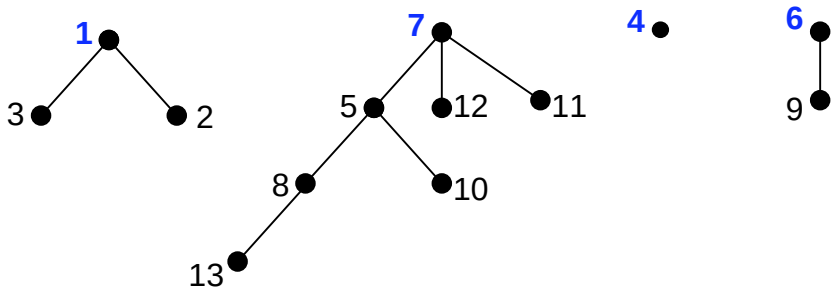
```
void unionBySize (int s1, int s2)
{
      if (-p[s1] < -p[s2] )
      {
            p[s2] -= p[s1];
            p[s1] = s2;
      }
      else
      {
            p[s1] -= p[s2];
            p[s2] = s1;
      }
}
```

Größe von s2

Größe von s1

Union-Find-Strukturen (6)

Beispiel für unionByHeight

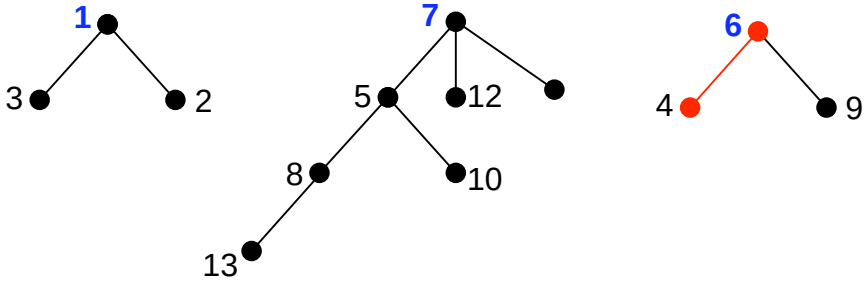


e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	-1	1	1	0	7	-1	-3	5	6	5	7	7

Wurzeln

Höheninformation

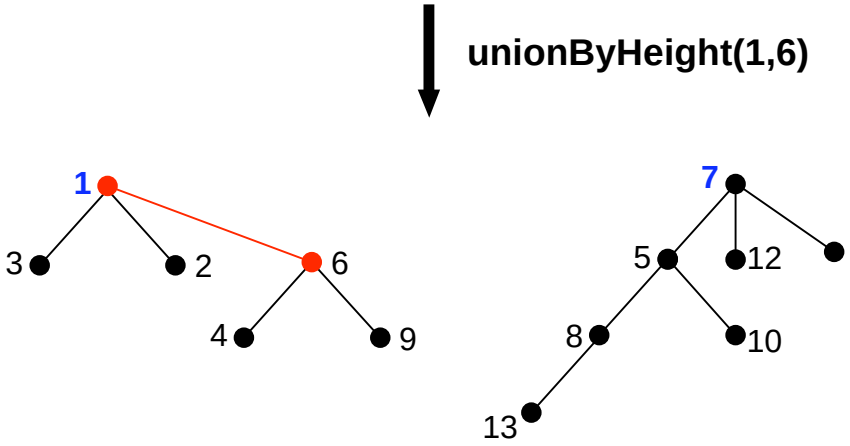
unionByHeight(4,6)



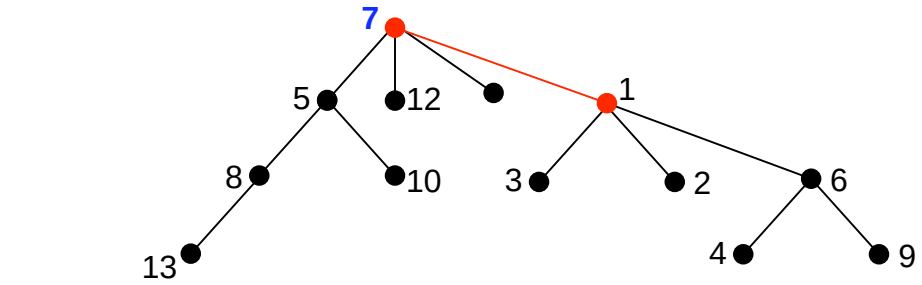
e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	-1	1	1	6	7	-1	-3	5	6	5	7	7

Union-Find-Strukturen (7)

Beispiel für unionByHeight (Fortsetzung)



e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	-2	1	1	6	7	1	-3	5	6	5	7	7



e	1	2	3	4	5	6	7	8	9	10	11	12
p[e]	7	1	1	6	7	1	-3	5	6	5	7	7

Union-Find-Strukturen (8)

Analyse:

Union: T(n) = O(1)

Find: T(n) = O(log n).