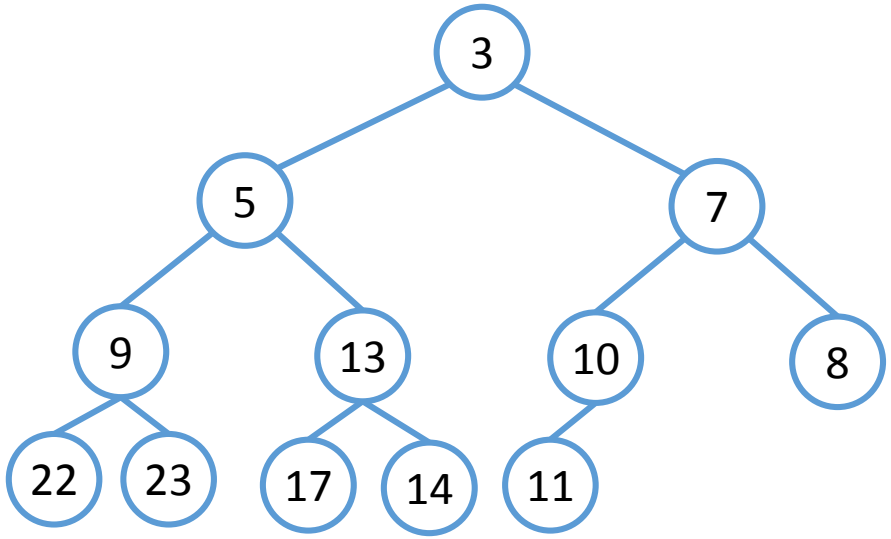
Heaps

Heaps

Ein (Min-)Heap ist ein Binärbaum, welcher

- die (Min-)Heap-Eigenschaft hat: Schlüssel eines Kindes ist immer grösser als der des Vaters.
  [Max-Heap: Kinder-Schlüssel immer kleiner als Vater-Schlüssel]
- bis auf die letzte Ebene vollständig ist
- höchstens eine Lücke in der letzten Ebene hat, welche rechts liegen muss



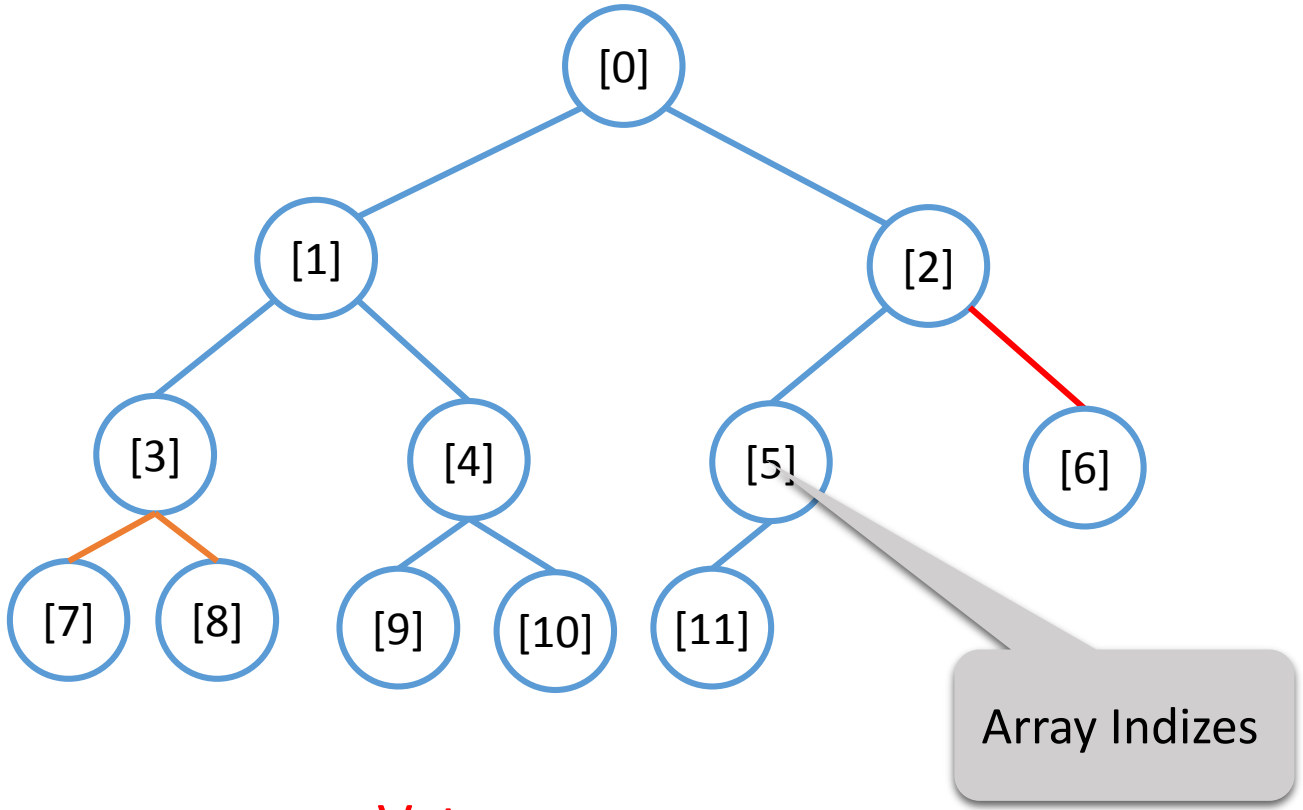
Heaps und Arrays

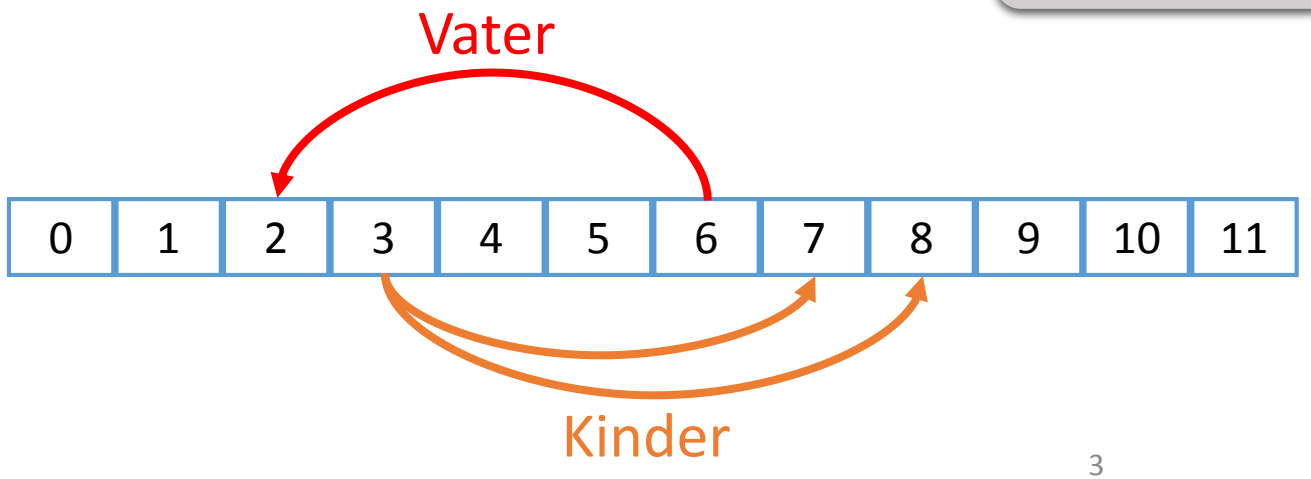
Ein Heap lässt sich sehr gut in einem Array speichern:

Es gilt

$$\text{Vater}(i) = \lfloor (i-1)/2 \rfloor,$$

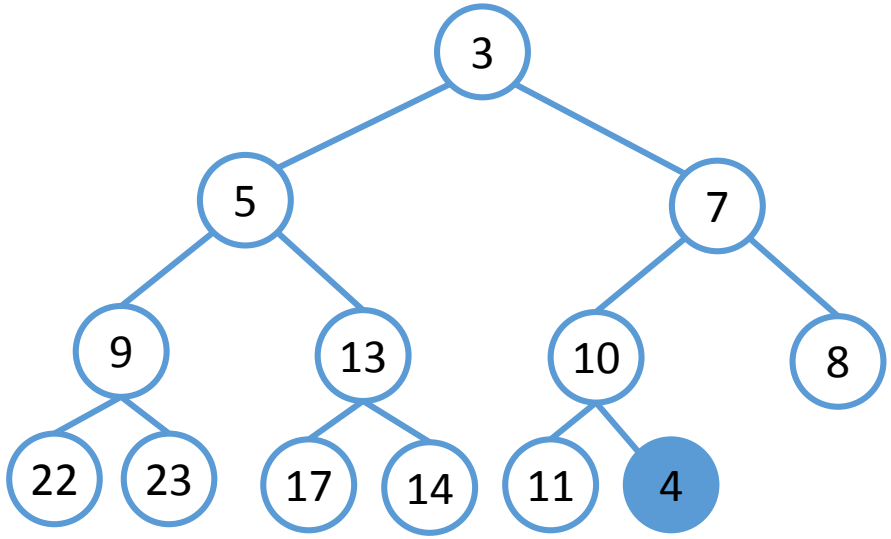
$$\text{Kinder}(i) = \{2i+1, 2i+2\}$$





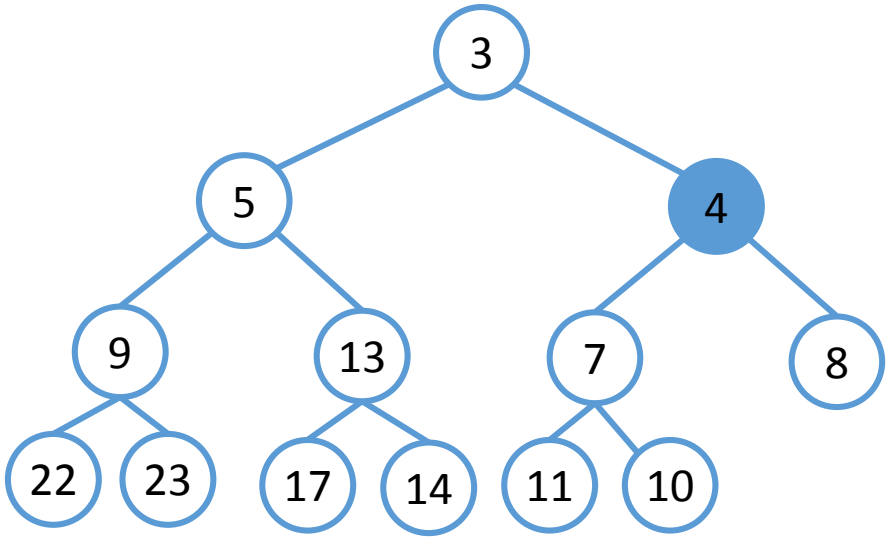
Einfügen

Füge ein neues Element k an der ersten freien Stelle ein. Verletzt Heap-Eigenschaft potentiell.



Stelle Heap-Eigenschaft wieder her durch sukzessives Aufsteigen von k.

Worst-Case Komplexität $\boldsymbol{O(log\, n)}$



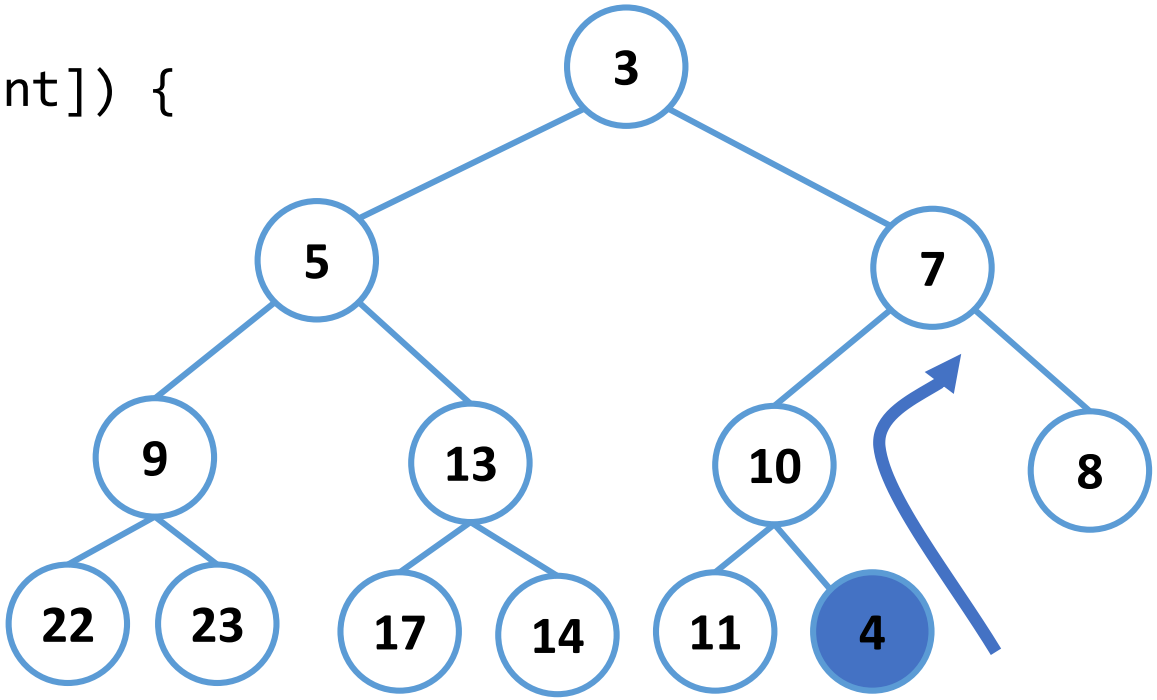
Datenstruktur ArrayHeap

```
public class ArrayHeap {

   float[] data;   // Array zum Speichern der Daten
   int used;       // Anzahl belegte Knoten

   ArrayHeap () {
      data = new float[16];
      used = 0;
   }

   void Grow(){ … } // Binäres Vergrössern von data, wenn nötig
   ...
}
```

Insert

```
public void Insert(double value){
   if (used == data.length)
      Grow();
   int current = used;
   int parent = (current-1)/2;
   while (current > 0 && value < data[parent]) {
      data[current] = data[parent];
      current = parent;
      parent = (current-1)/2;
   }
   data[current] = value;
   used++;
   Check(0); // Debugging check
}
```

Vater von current



GetMin

Das kleinste Element ist immer an der Wurzel im Baum. Somit kann es sehr schnell ausgelesen werden ($O(1)$).

Wie verhält es sich aber mit Auslesen *und Entfernen?*

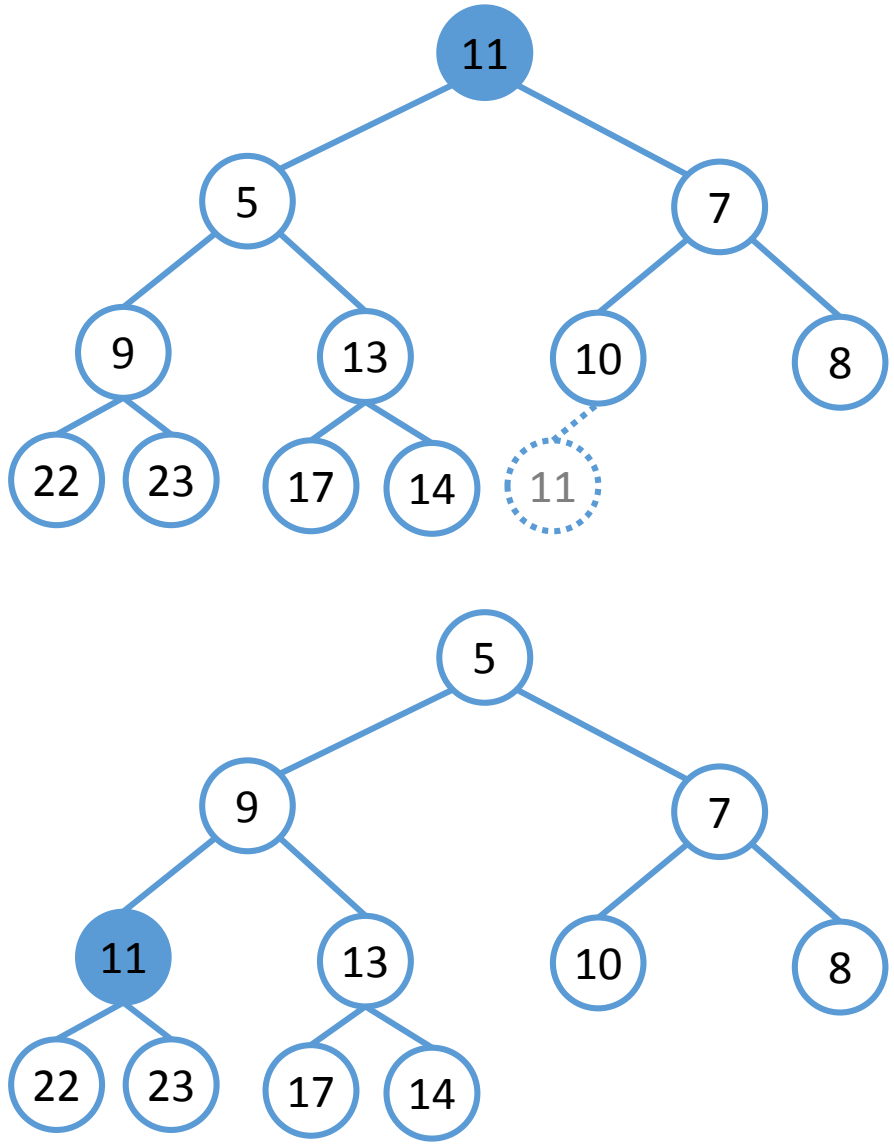
Wiederholtes Entfernen der Wurzel ergibt Schlüssel in aufsteigender Reihenfolge: das kann z.B. auch zum *Sortieren* verwendet werden (Heap-Sort Algorithmus).

Minimum entfernen

Ersetze die Wurzel durch den letzten Knoten

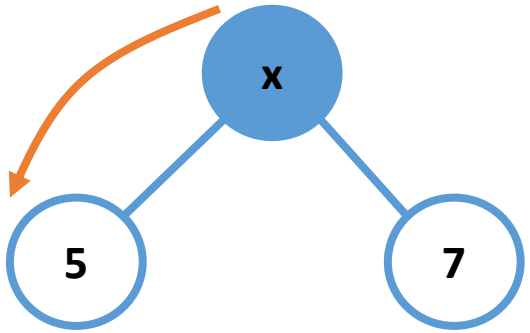
Lasse die Wurzel nach unten sinken, um die Invariante wiederherzustellen

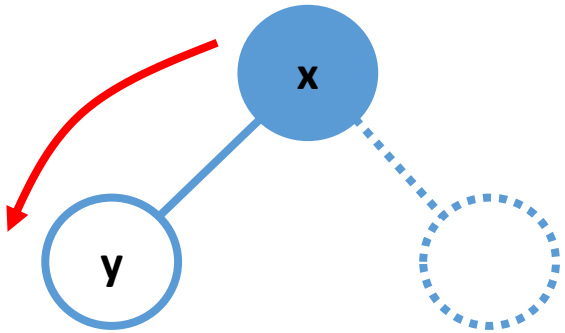
Worst-Case Komplexität
$\boldsymbol{O(log\, n)}$

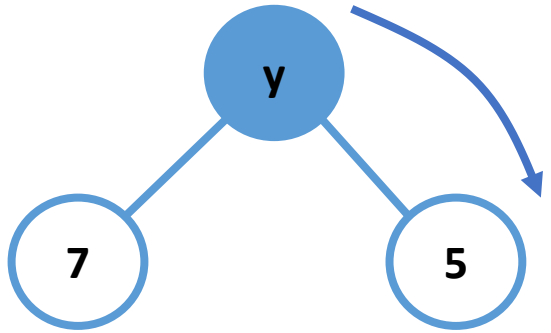


Absinken: Welche Richtung?

```
// return child with smaller value.
// If larger child index (2*current+2) is not in range, return smaller index
private int BestChild(int current) {
    if (2*current+2 >= used || data[2*current+1] < data[2*current+2])
        return 2*current+1;    // take left branch
    else
        return 2*current+2;    // right branch
}
```

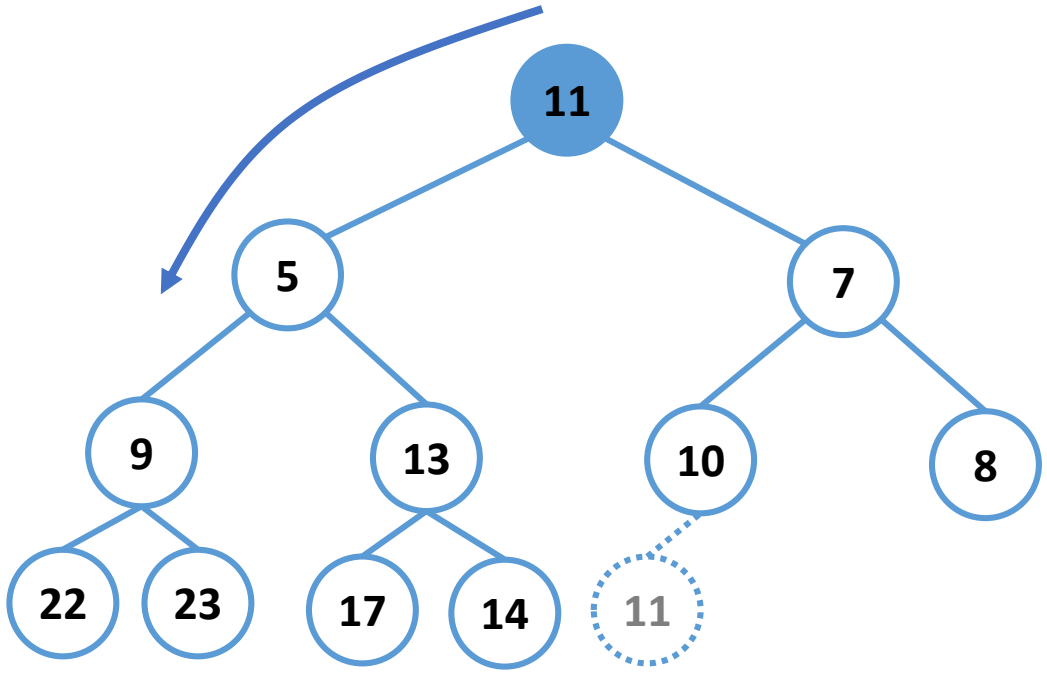






Wurzel Extrahieren

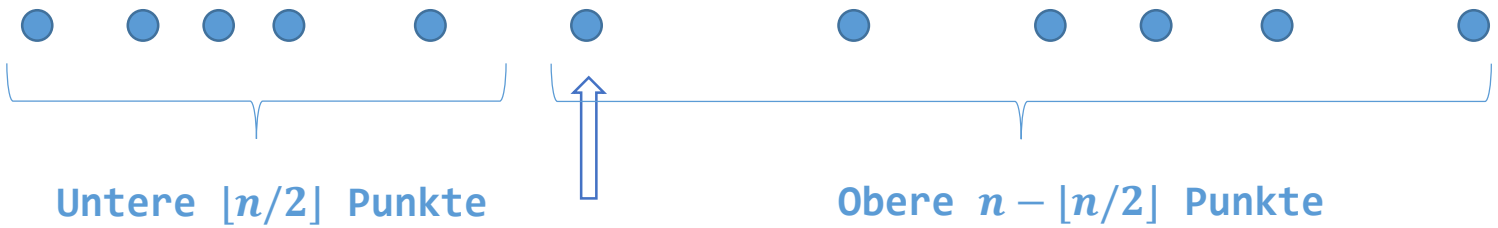
```
public double ExtractRoot() {
    double min = data[0];
    double value = data[used-1];
    int current = 0;
    int next = BestChild(current);
    while(next < used && !(value < data[next])) {
        data[current] = data[next];
        current=next;
        next = BestChild(current);
    }
    data[current] = value;
    used--;
    return min;
}
```



Verwendungsbeispiel Heap

Können wir das schnelle Auslesen, Extrahieren und Einfügen von Minimum (bzw. Maximum) im Heap für einen online-Algorithums des Median nutzen?

Beobachtung: Der Median bildet sich aus Minimum der oberen Hälfte der Daten und / oder Maximum der unteren Hälfte der Daten.



Online Median

Verwende Max-Heap H_{max} und Min-Heap H_{min}.
Bezeichne Anzahl Elemente jeweils mit $|H_{max}|$ und $|H_{min}|$

Einfügen neuen Wertes v in

- $H\_\max$, wenn $v \leq \max(H_{\max})$
- $H\_\min$, sonst

Rebalancieren der beiden Heaps

- Falls $|H\_\max| > \lfloor n/2 \rfloor$, dann extrahiere Wurzel von $H\_\max$ und füge den Wert bei $H\_\min$ ein.
- Falls $|H\_\max| < \lfloor n/2 \rfloor$, dann extrahiere Wurzel von H_min und füge den Wert bei H_max ein.

Gesamt worst-case Komplexität $O(\log n)$

Berechnung Median

Berechnung Median

Wenn n ungerade, dann

$$\text{median} = \min(H\_\text{min})$$

Wenn n gerade, dann

$$\text{median} = \frac{\max(H_{\max}) + \min(H_{\min})}{2}$$

➔ worst-case Komplexität $O(1)$

Hash-Tabellen

möglichst einfach

Datensatz

```
class Friend{
        String name;
        String telephone;

        Friend(String Name, String Tel) {
                name = Name;
                telephone = Telephone;
        }
}
```

Telefonbuch

```
class MyFriends{
    Friend[] data;
    int friends;

    MyFriends(int maxFriends) {
        data = new Friend[maxFriends]; friends = 0;
    }

    void Add(String name, String telephone){
        data[friends] = new Friend(name, telephone);
        friends ++;
    }

    Friend GetFriend(String name){
        for (int i = 0; i<friends; ++i)
            if (data[i].name.Equals(name))
                return data[i];
        return null;
    }
}
```

Schlecht: oft lange Suchzeiten O(data.length)

Besseres Telefonbuch

```
class MyFriends{
    Friend[] data;

    MyFriends(int maxFriends) {
        data = new Friend[maxFriends];
    }

    void Add(String name, String telephone){
        int index = Hash(name);
        while(data[index] != null)
            index = (index + 1) % data.length;
    }

    Friend GetTelephone(String name){
        int index = Hash(name);
        while(data[index] != null & !data[index].name.equals(name))
            index = (index + 1) % data.length;
        return data[index];
    }
}
```

Magie

Anwendungsbeispiel:

```
Friends friends =  new Friends(10);
friends.Add("Hannes", "01/007");
friends.Add("Niklas", "01/008");
friends.Add("Katrin", "01/009");
friends.Add("Tobias", "02/800");
friends.Add("Florian", "02/801");
...
System.out.println(friends.GetFriend("Tobias").telephone);
```

"02/800"

Magie enttarnt!

```
int Hash(String name){
        int value = 0;
        for (int i = 0; i < name.length(); ++i){
                char c = name.getCharAt(i);
                value = value * 31 + (int)c;
        }
        if (value<0)
                value = -value;
        return value % data.length;
}
```

c	(int) c	value
H	72	72
a	97	2329
n	110	72309
n	110	2241689
e	101	69492460
s	115	-2140700921

Warum funktioniert die Hastabelle?

Das Resultat von Hash(name) ist für sehr viele Namen unterschiedlich.

name	value	value % 100	value % 10
Hannes	2140700921	21	1
Niklas	1961629266	66	**6**
Katrin	2054630759	59	9
Tobias	1784584236	36	**6**
Florian	898707565	65	5

Die Einschränkung auf die Array-Grösse macht sog. Kollisionen wahrscheinlicher. Diese werden aber behandelt, indem noch lokal nachgeprüft (*sondiert*) wird.

Sondieren

```
class MyFriends{
    Friend[] data;

    MyFriends(int maxFriends) {
        data = new Friend[maxFriends];
    }

    void Add(String name, String telephone){
        int index = Hash(name);
        while(data[index] != null)
            index = (index + 1) % data.length;
    }

    Friend GetTelephone(String name){
        int index = Hash(name);
        while(data[index] != null && !data[index].name.equals(name))
            index = (index + 1) % data.length;
        return data[index];
    }
}
```

name	value	value % 10
Hannes	2140700921	1
Niklas	1961629266	**6**
Katrin	2054630759	9
Tobias	1784584236	**6**
Florian	898707565	5

index	data
0	null
1	(Hannes)
2	null
3	null
4	null
5	(Florian)
6	**(Niklas)**
7	**(Tobias)**
8	null
9	Katrin

Lineares Sondieren

Lineares Sondieren

Zusammengefasst

Hash-Tabellen funktionieren, wenn eine gute Hash-Funktion vorhanden ist, welche verschiedene Werte möglichst auf verschiedene Array-Indizes abbildet.

Darüber hinaus muss der Indexbereich (das Array) gross genug sein, damit Kollisionen unwahrscheinlich werden.

Sind Kollisionen hinreichend unwahrscheilich, so findet man Elemente in der Hashtabelle in O(1)!

Es lässt sich noch viel sagen über: Löschen von Einträgen, Andere Sondiermethoden, dynamische Hash-Tabellen, Hash-Funktionen, Kollisionswahrscheinlichkeiten etc., aber wir wollen hier nur das Prinzip verstehen.

Generische Hashtabellen in Java

Wir kennen bereits java.util.Vector: Klasse für dynamisches, selbstwachsendes Array.

Neu: Klasse für HashTabelle java.util.HashMap

```
HashMap<String, Integer> map; // Abbildung String → Integer


map.put("abc",3);
map.put("xyz",100);
int i = map.get("abc"); // i = 3 !
int j = map.get("xyz"); // i = 100 !
```

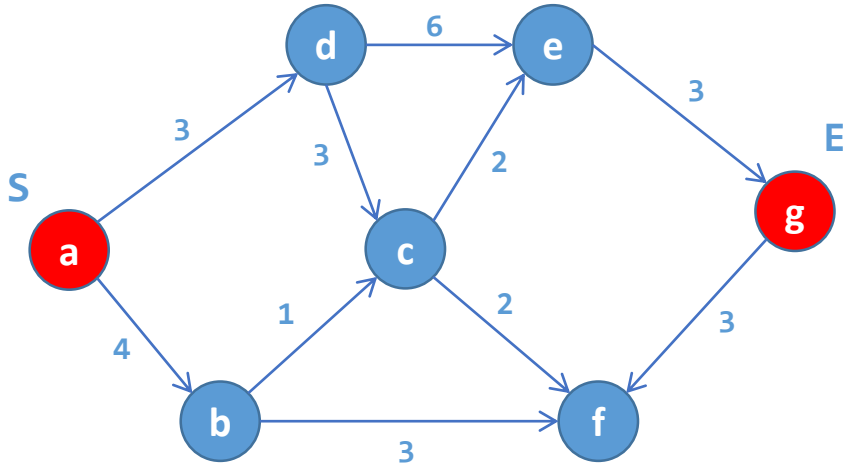
Kann zwischen allen Objekte abbilden!
Also, z.B. auch Node → Integer

Shortest Paths

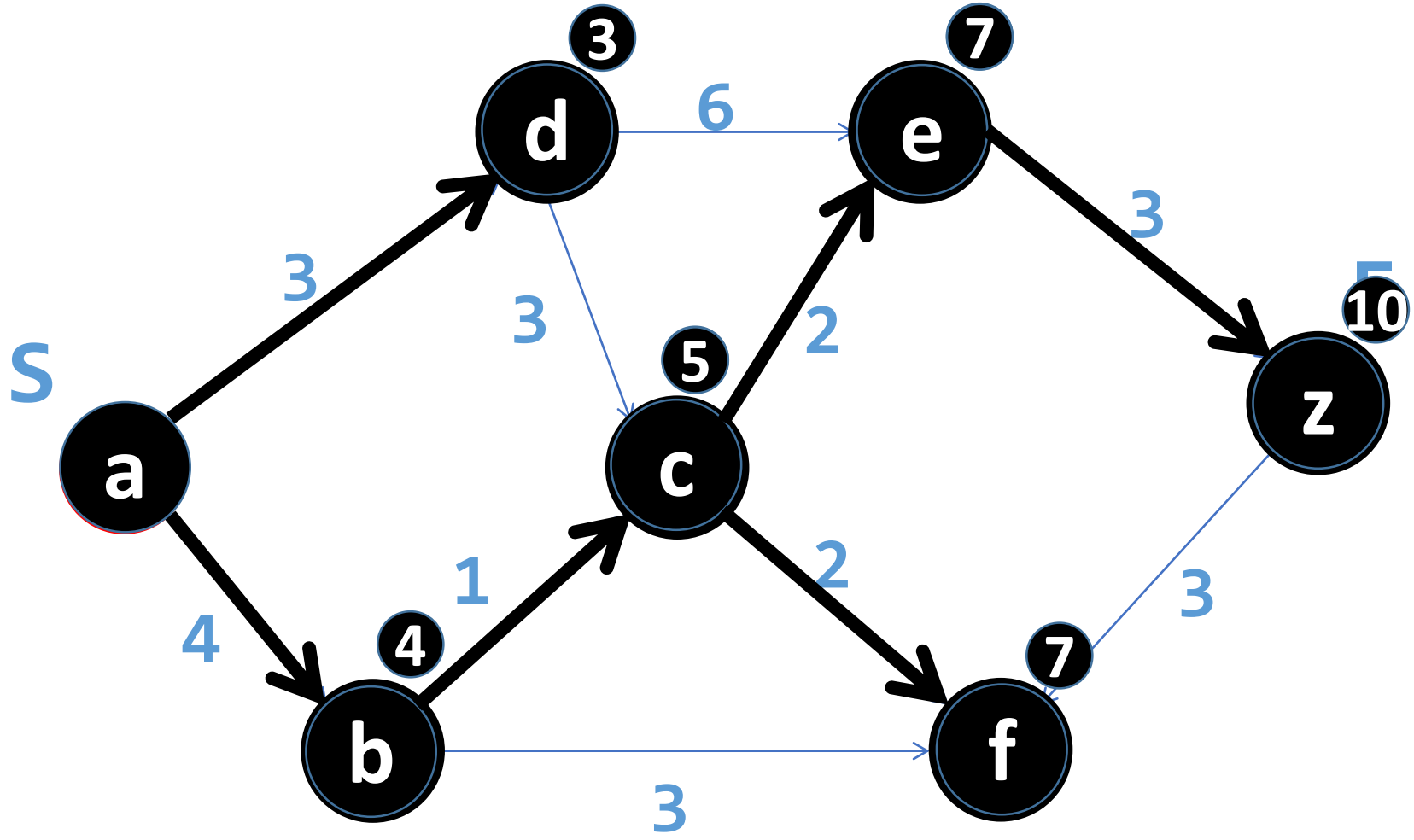
Fallstudie Dijkstra's Shortest Path

Gegeben: Gerichteter *Graph* (V, E) mit Knotenmenge V und Kantenmenge E, bei dem jeder Kante $e \in E$ eine Länge $l(e) \geq 0$ zugeordnet ist.

Problem: Finde zu Startpunkt $S \in V$ und Endpunkt $E \in V$ den kürzesten Pfad entlang der Kanten im Graph.



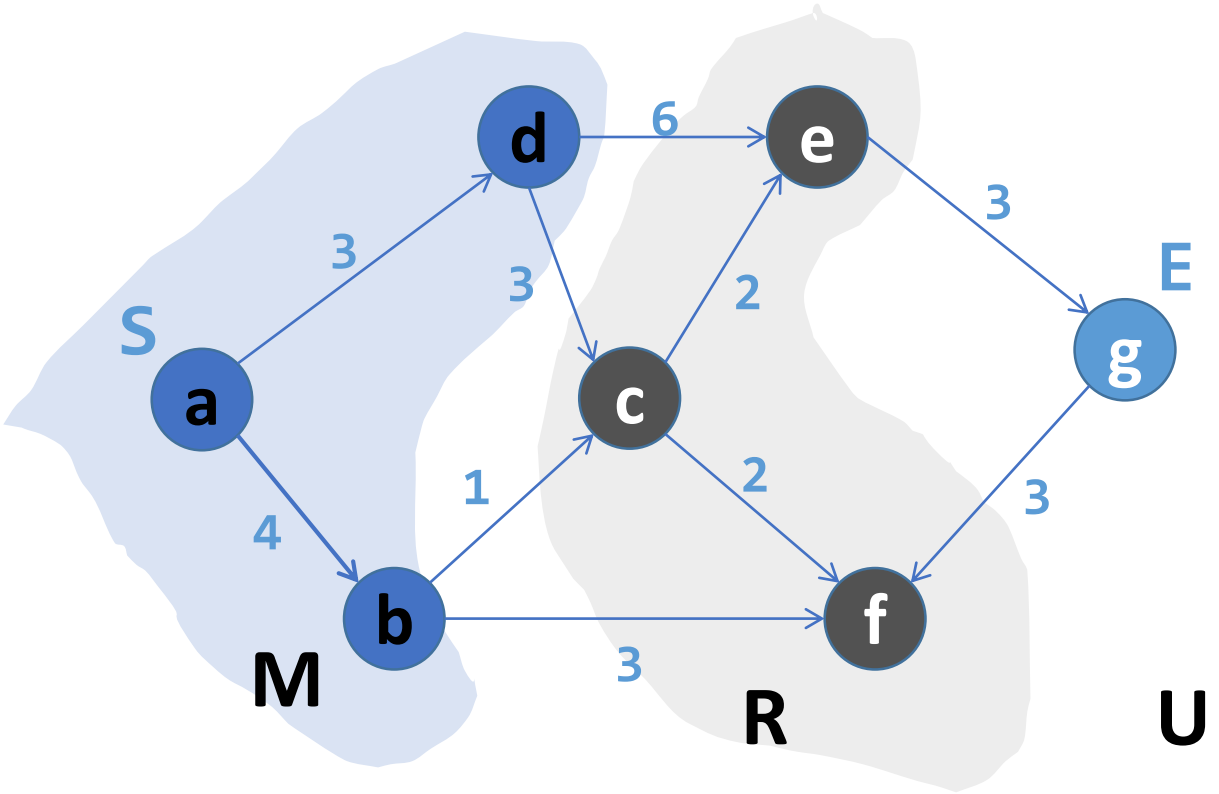
Shortest Path: Animation



Implementation

Grundsätzlich wird die Menge der Knoten unterteilt in

a. Knoten, die schon als Teil eines minimalen Pfades erkannt wurden (M)
b. Knoten, die nicht in M enthalten sind, jedoch von M aus direkt (über eine Kante) erreichbar sind (R) und
c. Knoten die noch nie berücksichtigt wurden ($U := V \setminus (M \cup R)$)



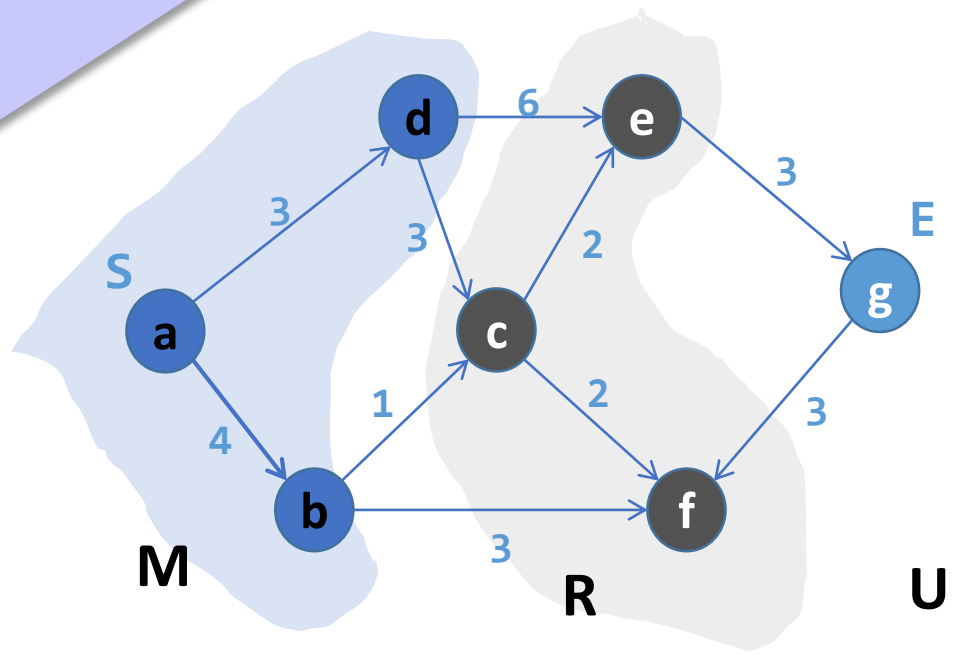
Algorithmus

Ein Knoten K aus R mit minimaler Pfadlänge in R kann nicht mit kürzerer Pfadlänge über einen anderen Knoten in R erreicht werden.

Daher kann er zu M hinzugenommen werden.

Dabei vergrössert sich R potentiell um die Nachbarschaft von K und die Pfadlänge aller von K aus direkt erreichbaren Knoten muss angepasst werden.

Daher bietet sich die Datenstruktur Heap für R an!



(a,0), (a-d,3), (a-b,4) +(b-c,5)

Beim Anpassen der Nachbarn von K sind potentiell auch Elemente von R betroffen.
Nie jedoch Elemente aus M oder gar U

Algorithmus

[Initial gilt: Pfadlänge $(K) = \infty$ für alle Knoten im Graph]

Setze $\text{Pfadlänge}(S) = 0$;

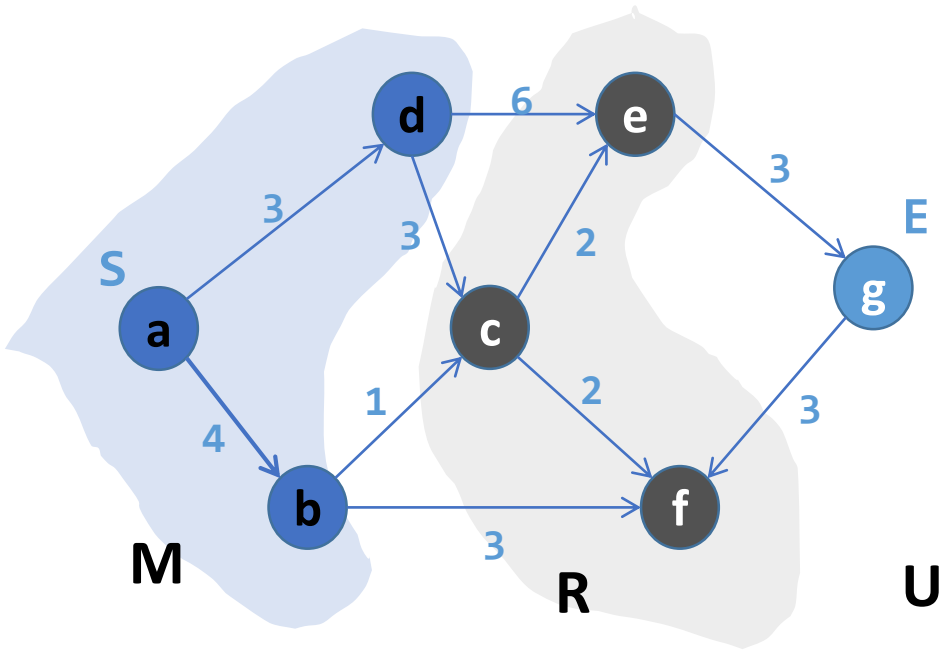
Starte mit $M = \{S\}$; Setze $K = S$;

Solange ein neuer Knoten K hinzukommt und dieser nicht der Zielknoten ist

- Für jeden Nachbarknoten N von K
  - Berechne die Pfadlänge x nach N über K.
  - Ist $\text{Pfadlänge}(N) = \infty$, so nimm N zu R hinzu.
  - Ist $x < \text{Pfadlänge}(N) < \infty$, so setze $\text{Pfadlänge}(N) = x$ und passe R an den neuen Wert an.
- Wähle als neuen Knoten K den Knoten mit kleinster Pfadlänge in R

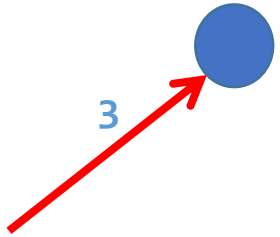
Modellierung

- Graph = Menge von Knoten
- Knoten mit ausgehenden Kanten und zuletzt ermittelter Pfadlänge (für den Algorithmus)
- Kanten mit Länge und Zielknoten
- MinHeap R mit schnellem Zugriff auf kürzesten Pfad

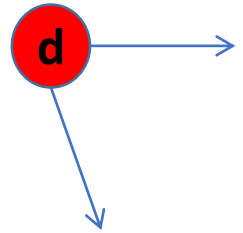


Kanten und Knoten

```
public class Edge {
    private Node to;
    private int length;

    Edge (Node t, int l) {
        to = t; length = l;
    }

    // Getters and Setters omitted for brevity
}
```




```
public class Node {
    private Vector<Edge> out;
    private String name;
    private int pathLen;
    private Node pathParent;

    Node(String n) {
        out = new Vector<Edge>();
        pathParent = null;
        pathLen = Integer.MAX_VALUE;
        name = n;
    }
    // Getters and Setters omitted for brevity
}
```



Graph

```
public class Graph {
        private Vector<Node> nodes;
        private HashMap<String,Node> nodeByName;
        Graph(){
                nodes = new Vector<Node>();
                nodeByName = new HashMap<String,Node>();
        }
        public void AddNode(String s){}
        public Node FindNode(String s){}
        public void AddEdge(String from, String to, int length) {}

        Vector<Node> ShortestPath(Node S, Node E) {};
}
```

Min-Heap R

```
public class Heap {
        ...
        // Vergleich zweier Knoten = Vergleich der aktuellen Pfadlängen
        private boolean Smaller(Node l, Node r) {
                return l.GetPathLen() < r.GetPathLen();
        }
        public void Insert(Node n) { ... }
        public Node ExtractRoot() { ... }
}
```

Was ist damit: "Ist x $<$ Pfadlänge$(N) < \infty$, so setze Pfadlänge$(N) = x$ und passe R an den neuen Wert an." ?

→ neue Methode DecreaseKey !

Schnelles Finden von Nodes im Heap?

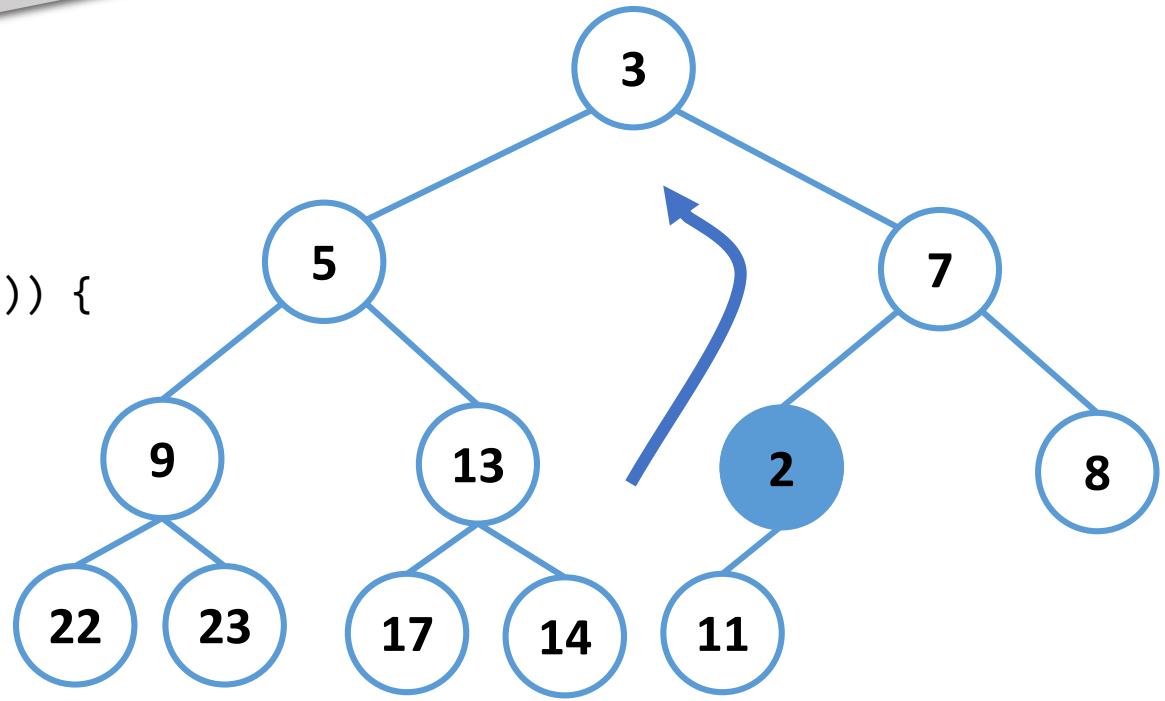
```
public class Heap {
   ...
   HashMap <Node, Integer> map;

   // Damit die Hashtabelle immer konsistent bleibt, muss nun
   // an allen Stellen im Code, wo vorher data[x] = y stand,
   // Set(x,y) stehen:
   private void Set(int index, Node value){
      data[index] = value;
      map.put(value, index);
   }

}
```

neu:
Hash-Tabelle Node → Index !

Heap: DecreaseKey

```
public class Heap {
    ...
    public void DecreaseKey(Node n)
    {
        int current = map.get(n);
        int parent = (current-1)/2;
        // aufsteigen
        while (current > 0 && Smaller(n, data[parent])) {
            Set(current, data[parent]);
            current = parent;
            parent = (current-1)/2;
        }
        Set(current, n);
    }
}
```

Hier brauchen wir die Hashtabelle!

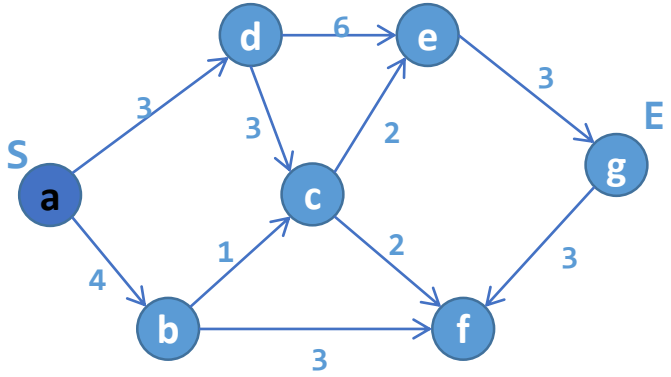


Alle Bausteine sind da.
Implementation des Shortest Path Algorithmus in der Übung.

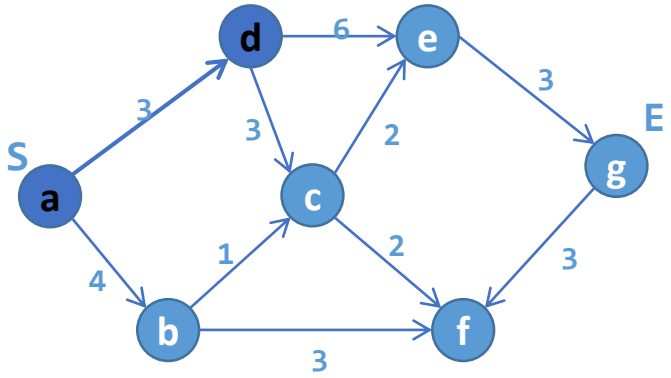
Appendix: Animation auf Folie 26 expandiert

Durchprobieren aller Pfade zu ineffizient

Dijkstra's Idee: Aufbau der kürzesten Pfade bis Ziel gefunden

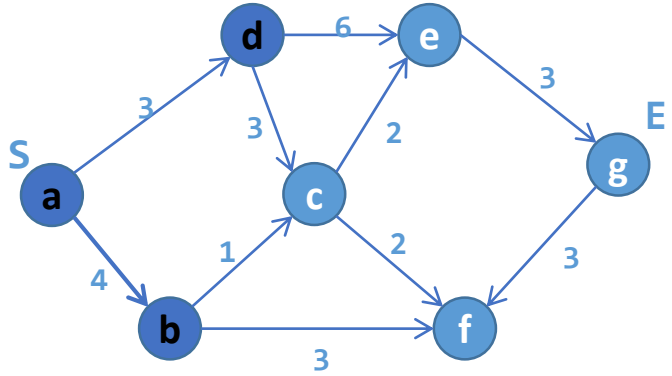


Starte bei S,
(Knoten, Pfadlänge) : (a,0)

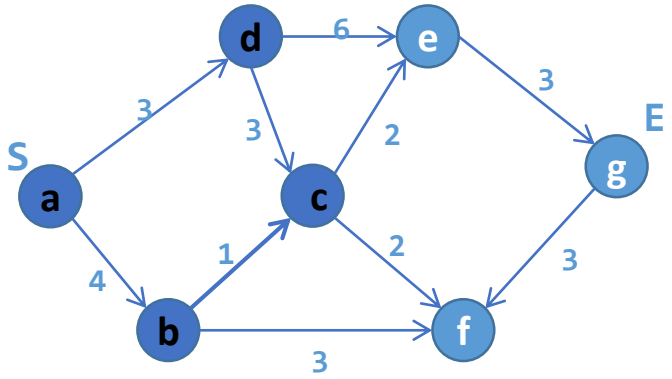


Kürzester Weg von (a,0)
Zusätzliche Kante
+(a-d,3)

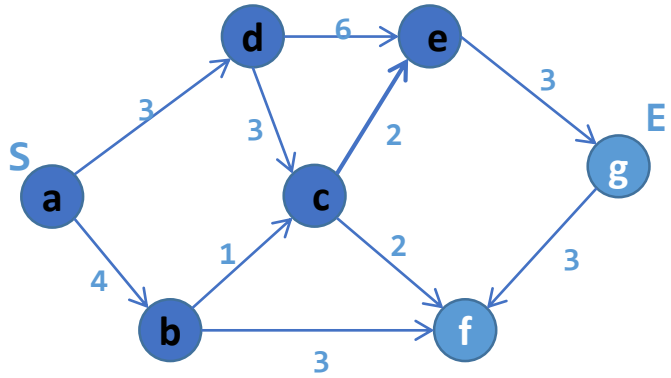
Algorithmus Idee



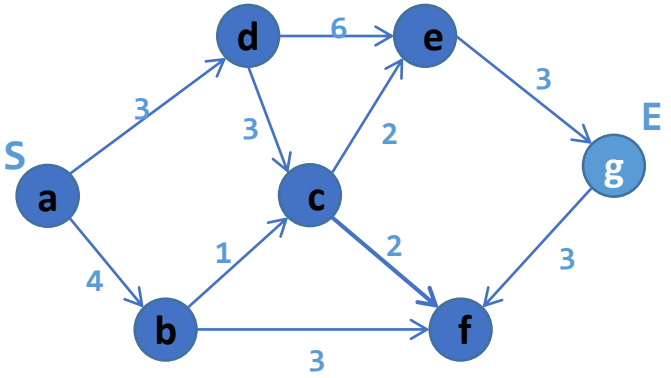
(a,0), (a-d,3) +(a-b,4)



(a,0), (a-d,3), (a-b,4) +(b-c,5)



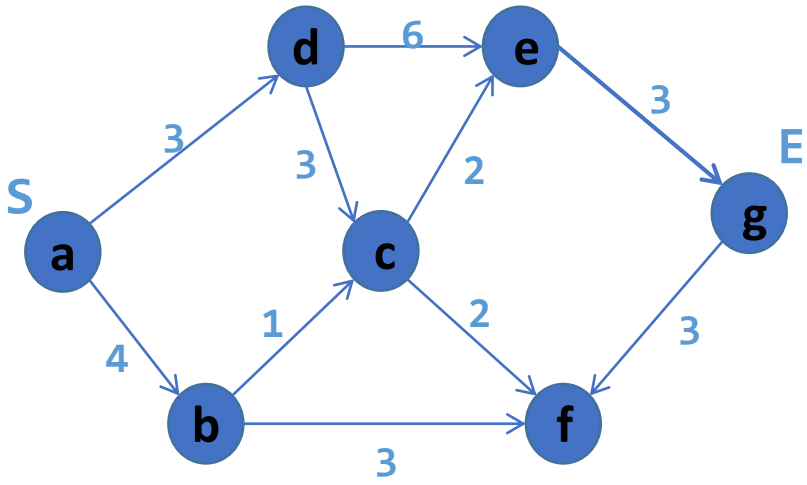
(a,0), (a-d,3), (a-b,4), (b-c,5) +(c-e,7)



(a,0), (a-d,3), (a-b,4), (b-c,5), (c-e,7), + (c-f,7)

Algorithmus Idee

Algorithmus terminiert, wenn Ziel erreicht



(a,0), (a-d,3), (a-b,4), (b-c,5), (c-e,7), (c-f,7) + (e-g,10)

Weg finden: über Vorgänger zurücklaufen