Informatik II

Vorlesung am D-BAUG der ETH Zürich

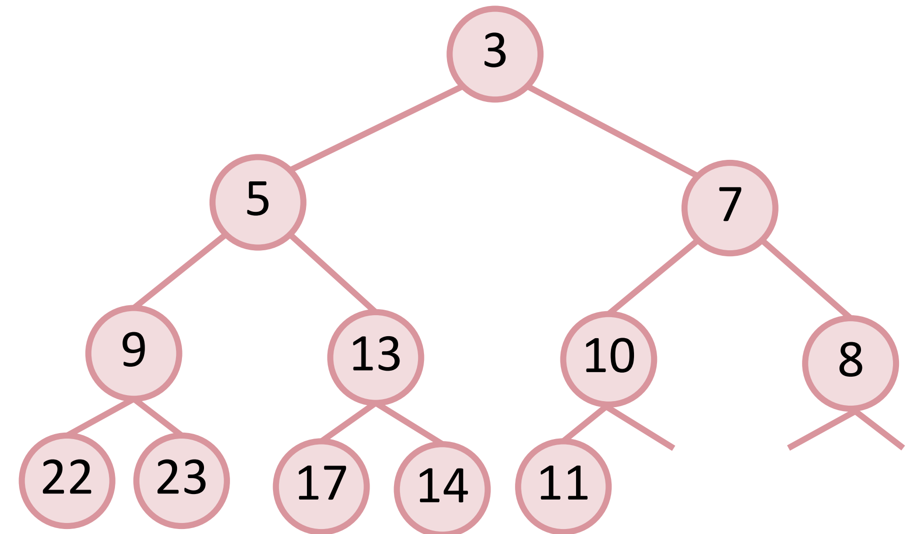
Vorlesung 10, 9.5.2016

Heaps und Shortest Paths

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 Lücken in der letzten Ebene hat, welche alle rechts liegen müssen



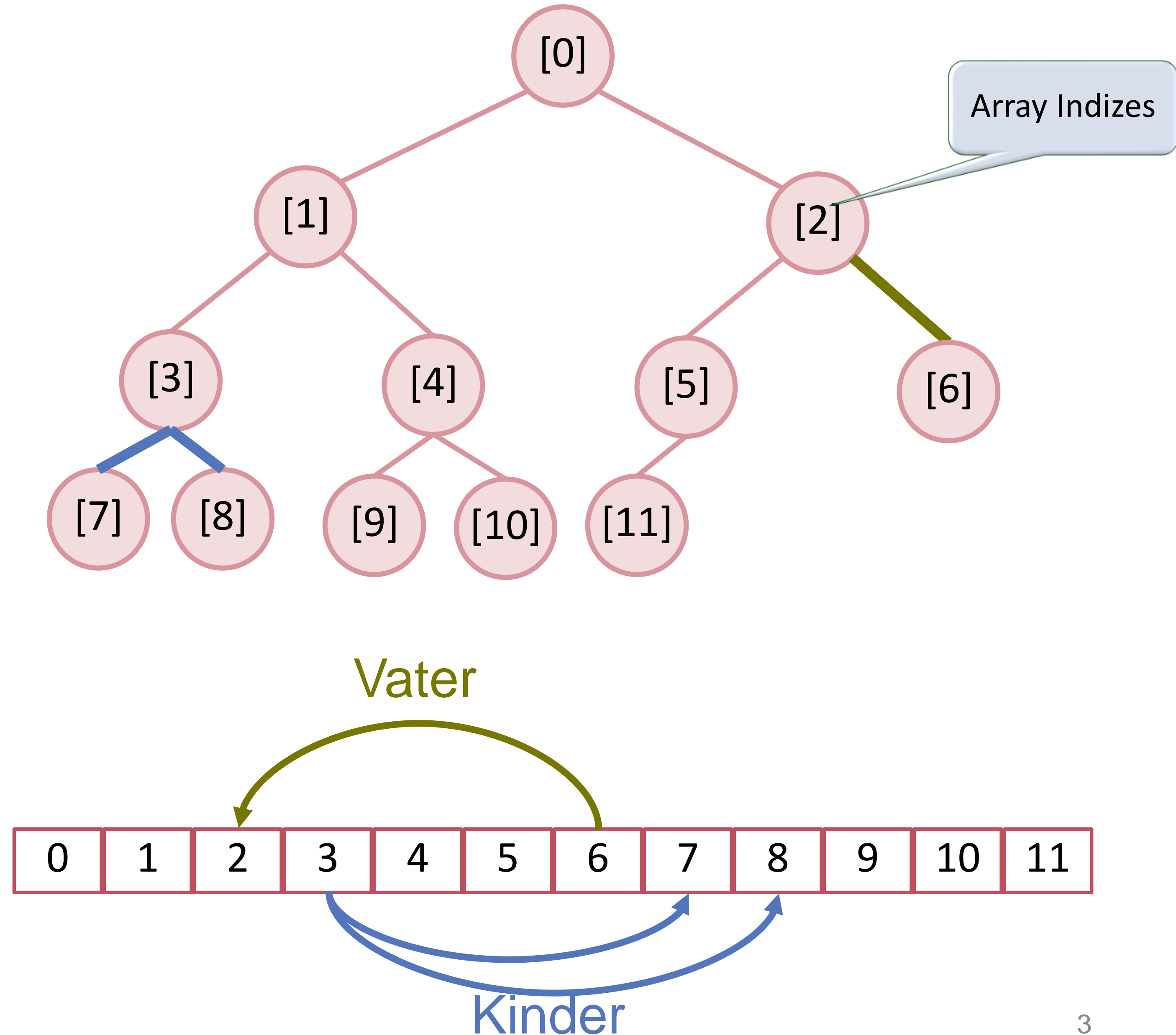
Ein Heap ist kein Suchbaum

Heaps und Arrays

Ein Heap lässt sich sehr gut in einem Array speichern:

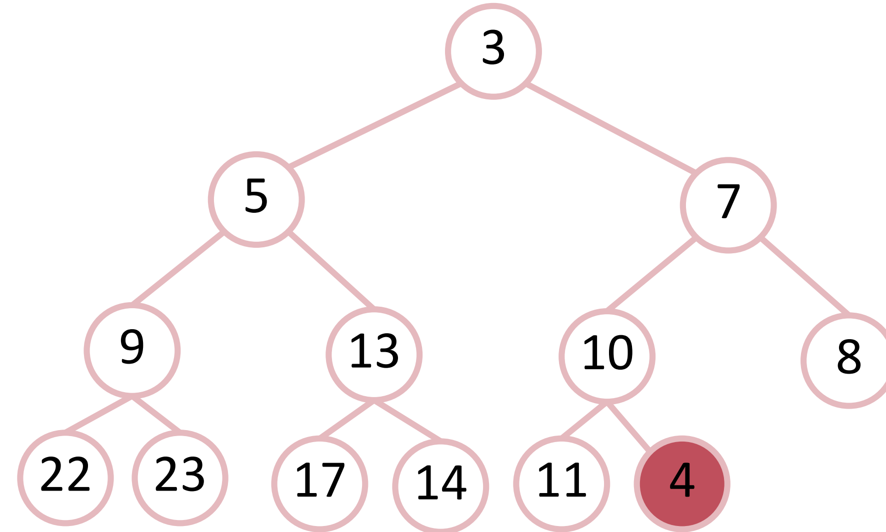
Es gilt

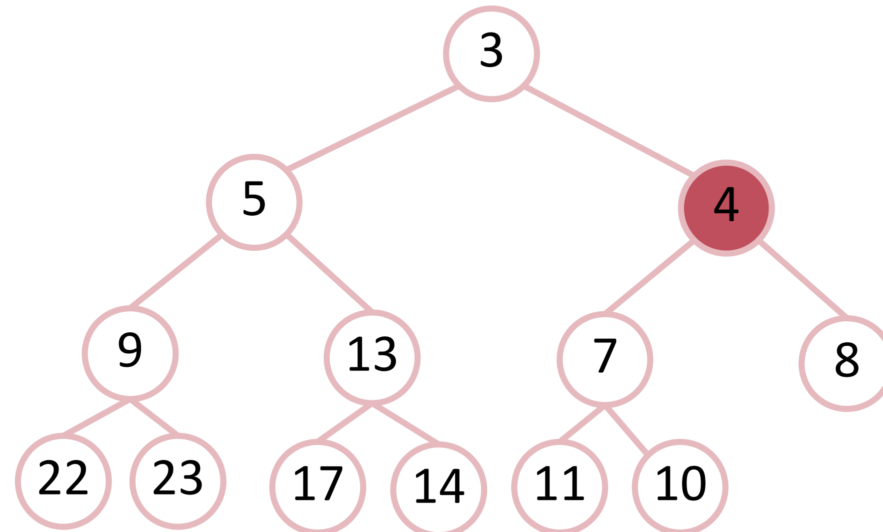
- $\text{Kinder}(i) = \{2i + 1, 2i + 2\}$
- $\text{Vater}(i) = \lfloor (i - 1)/2 \rfloor$



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 $\boldsymbol{O(log\, n)}$



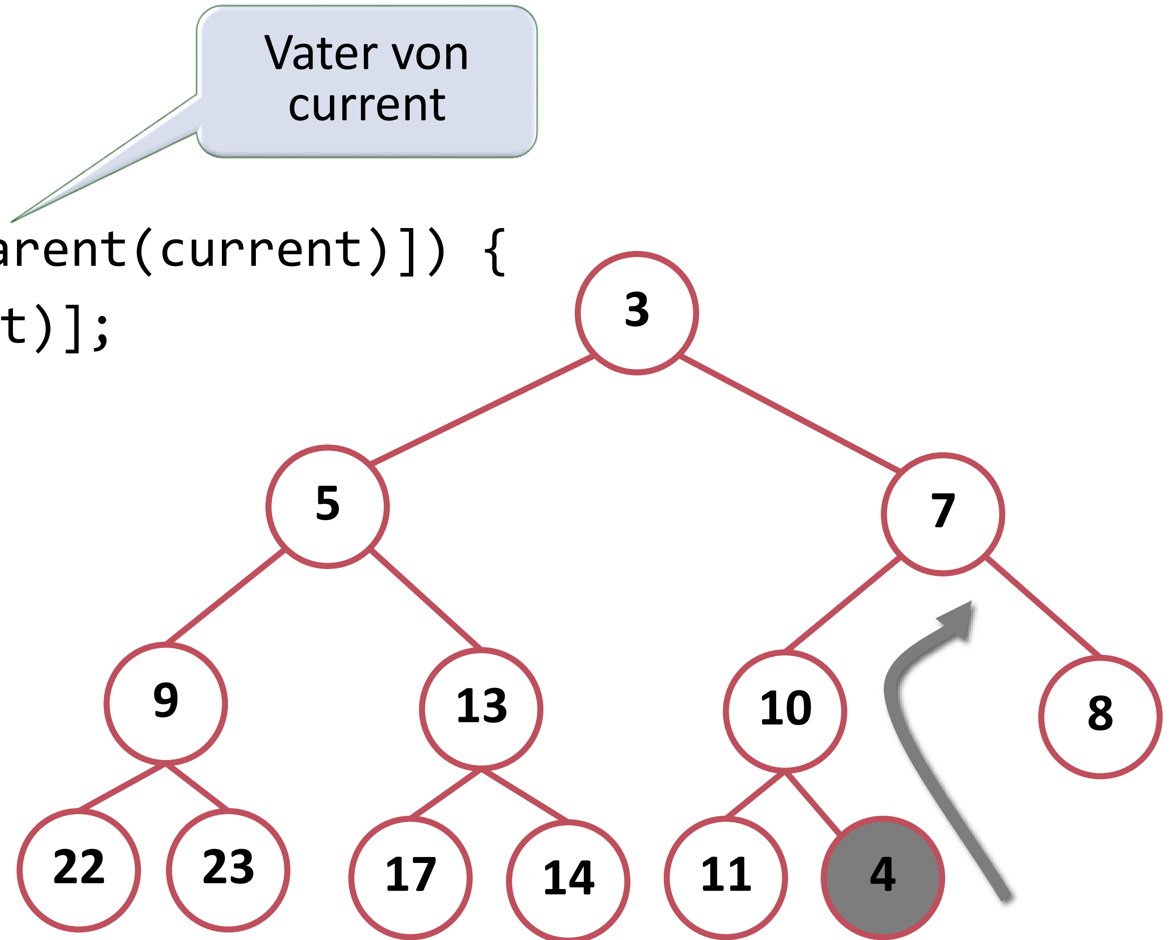


Datenstruktur ArrayHeap

```
public class ArrayHeap {

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

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

   int Parent(int of){
      return (of-1)/2:
   }

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

Insert

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



GetMin

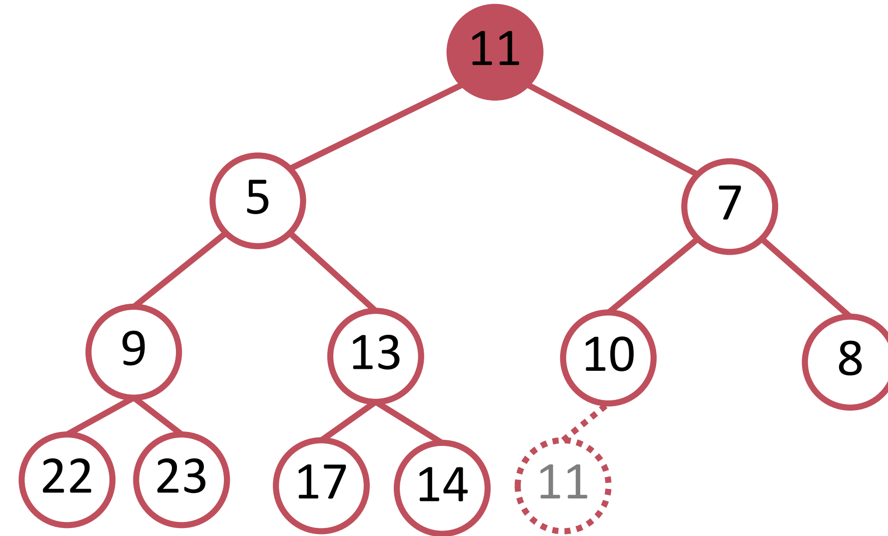
Das kleinste Element ist immer an der Wurzel im Baum. Somit kann es sehr schnell ausgelesen werden ($O(1)$).

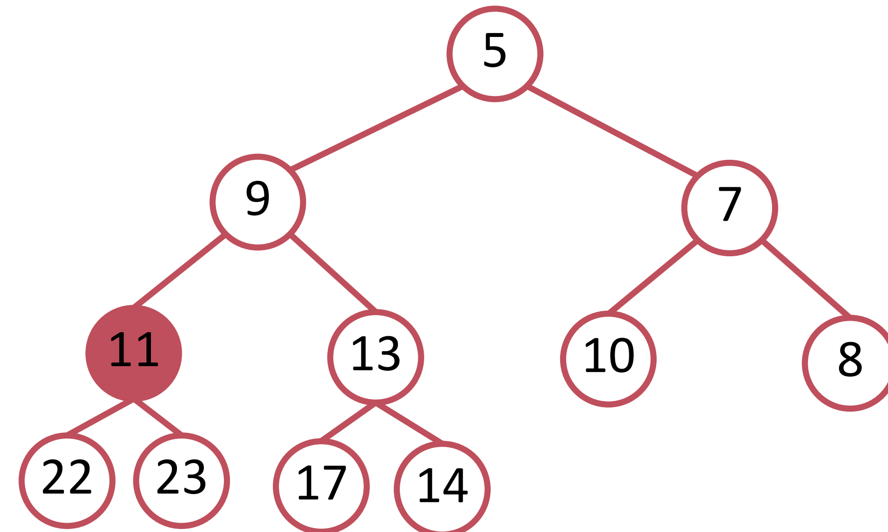
Wie verhält es sich aber mit Auslesen ***und Entfernen***?

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

Minimum entfernen

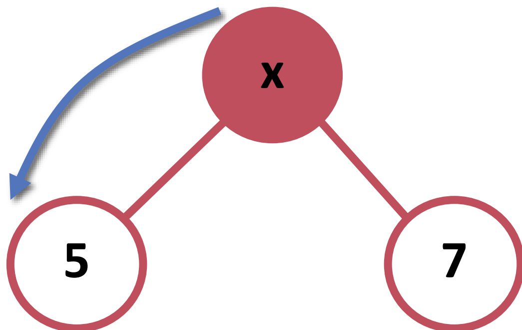
- Ersetze die Wurzel durch den letzten Knoten
- Lasse die Wurzel nach unten sinken, um die Invariante wiederherzustellen
- Worst-Case Komplexität $\boldsymbol{O(\log n)}$

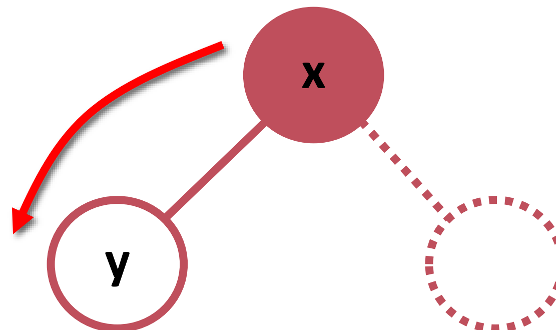


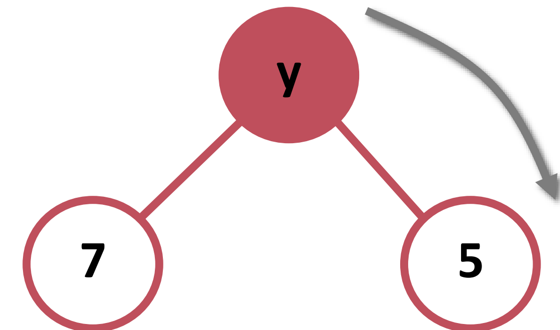


Absinken: Welche Richtung?

```
// 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
}
```

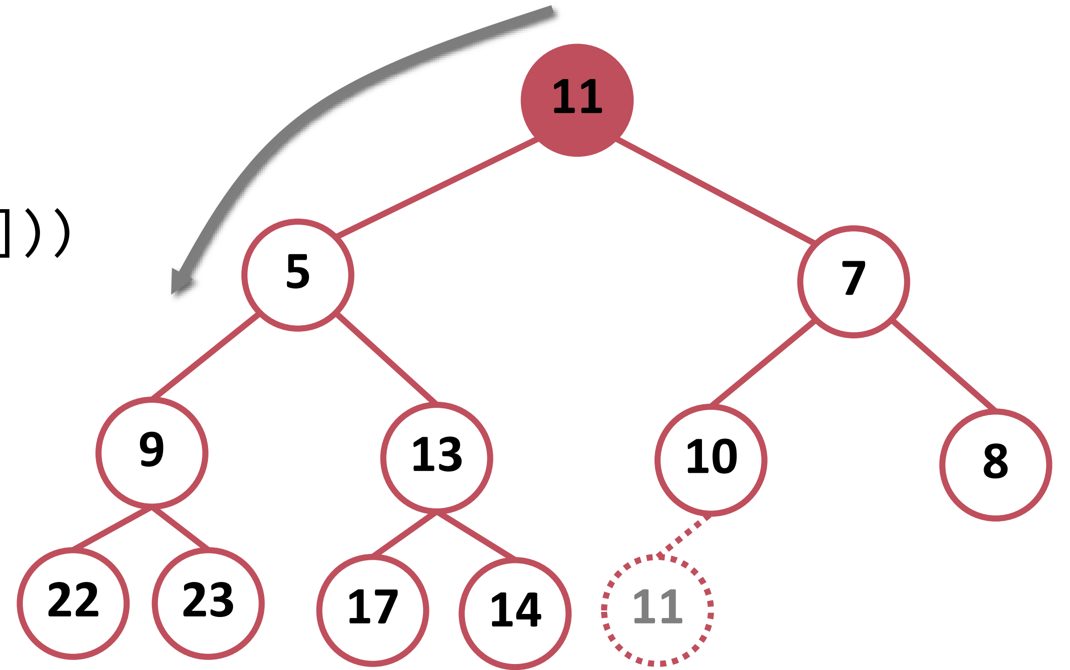






Wurzel Extrahieren

```
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

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

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 $|H_{max}| = 0$ *oder* $v \leq \max(H_{\max})$

  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 des Einfügens: $O(\log n)$

Berechnung Median

Berechnung Median

- Wenn n ungerade, dann

$$\text{median} = \min(H_{\min})$$

- Wenn n gerade, dann

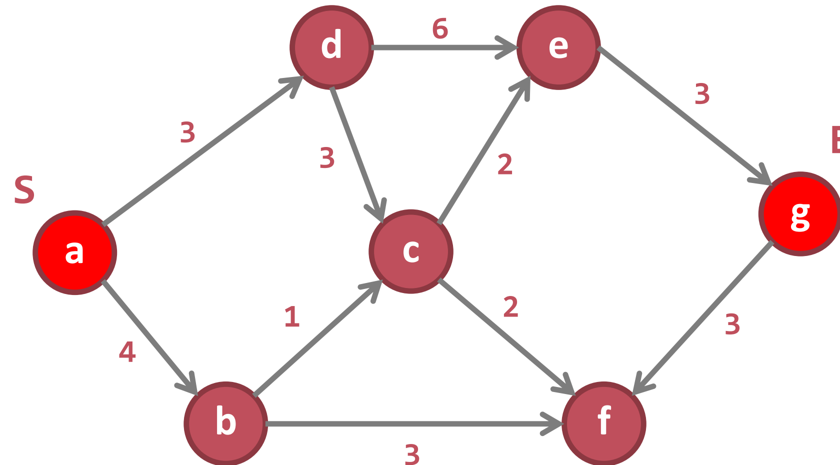
$$\text{median} = \frac{\max(H_{\max}) + \min(H_{\min})}{2}$$

➔ worst-case Komplexität $O(1)$

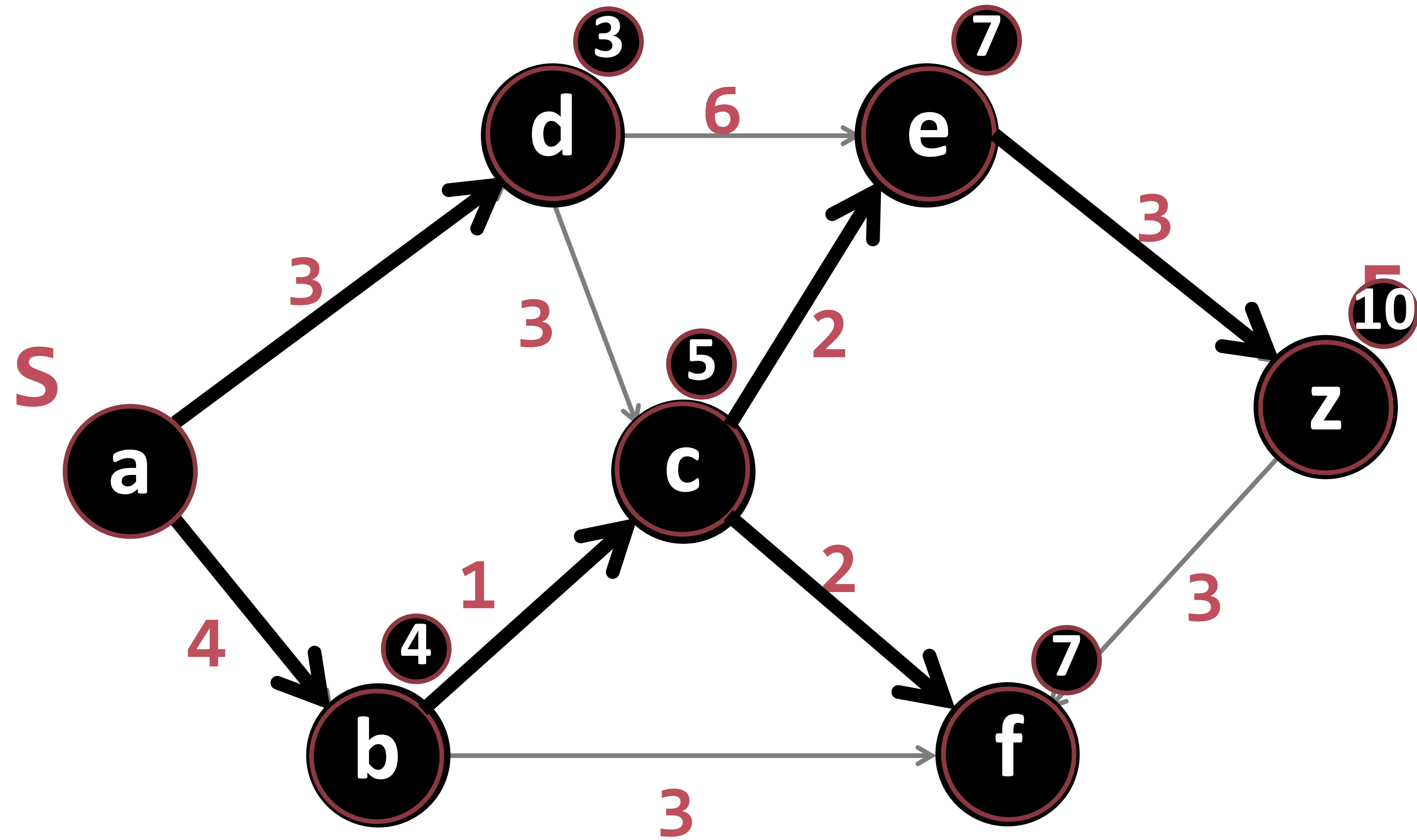
DIJKSTRA'S SHORTEST PATH ALGORITHMUS

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.

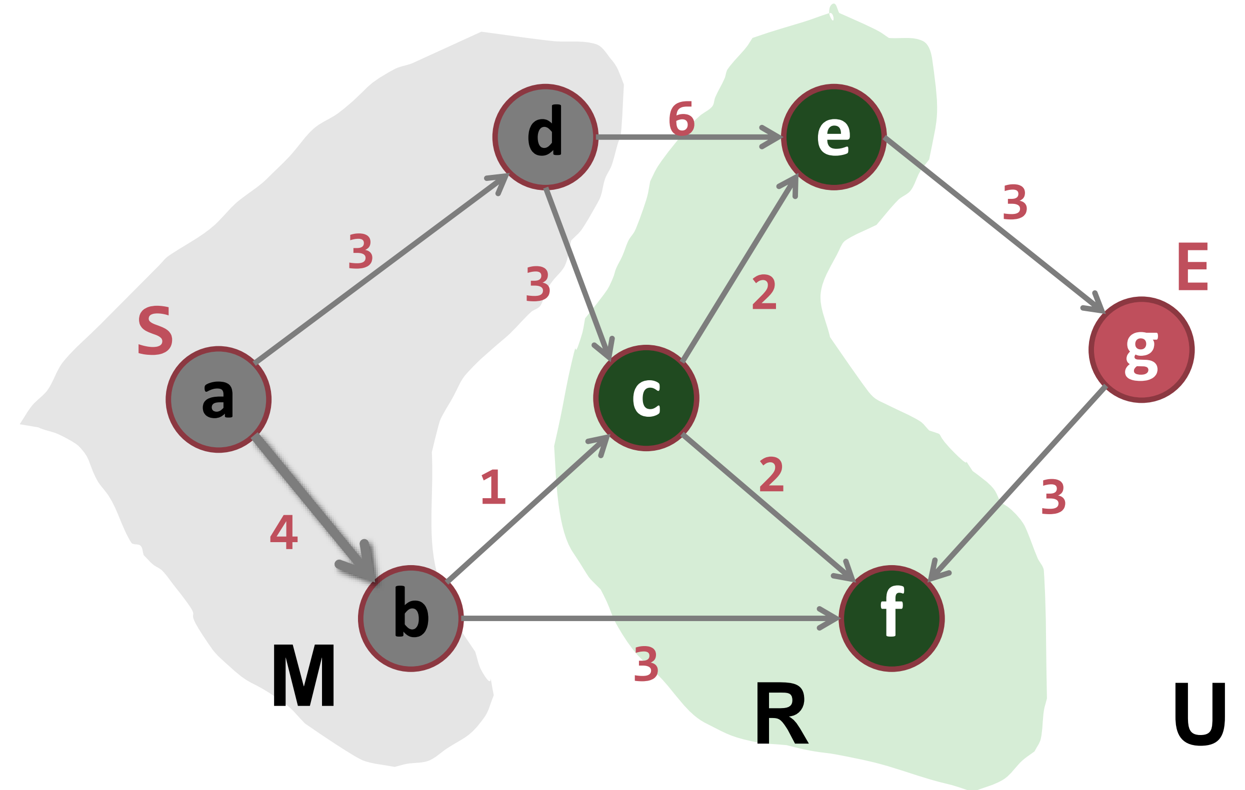


Shortest Path: Animation



Implementation

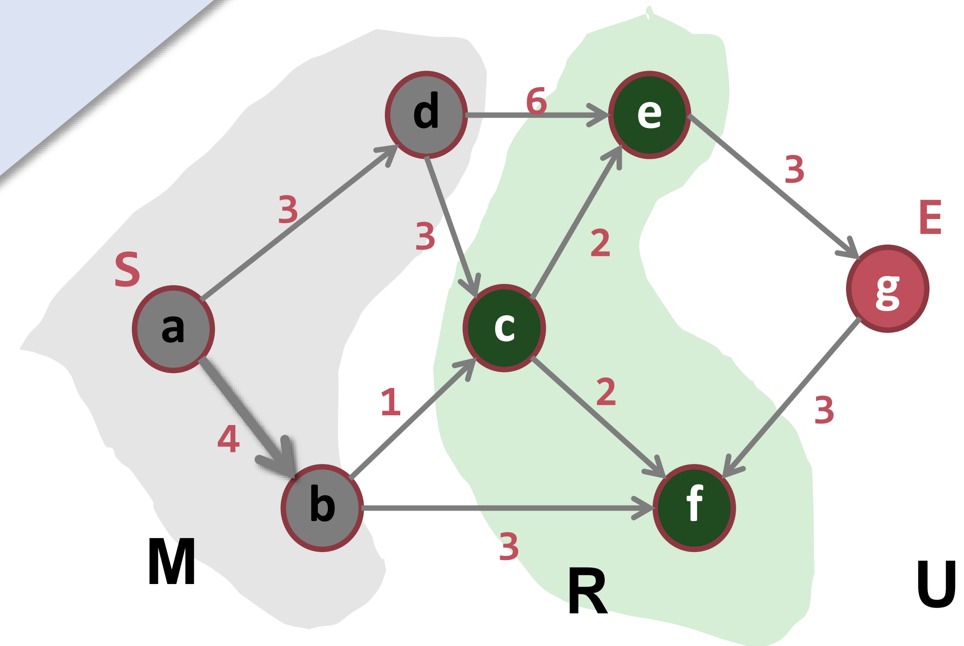
- Grundsätzlich wird die Menge der Knoten unterteilt in
  a. Knoten, die schon als Teil eines minimalen Pfades erkannt wurden (M)
  b. Knoten, die nicht in M enthalten sind, jedoch von M aus direkt (über eine Kante) erreichbar sind (R) und
  c. Knoten die noch nie berücksichtigt wurden ($U := V \setminus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



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: $\text{Pfadlänge}\ (K)\ =\ \infty$ für alle Knoten im Graph]

$\text{Setze Pfadlänge}(S) = 0;$

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 $\text{x} < \text{Pfadlänge}(N) < \infty$, 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

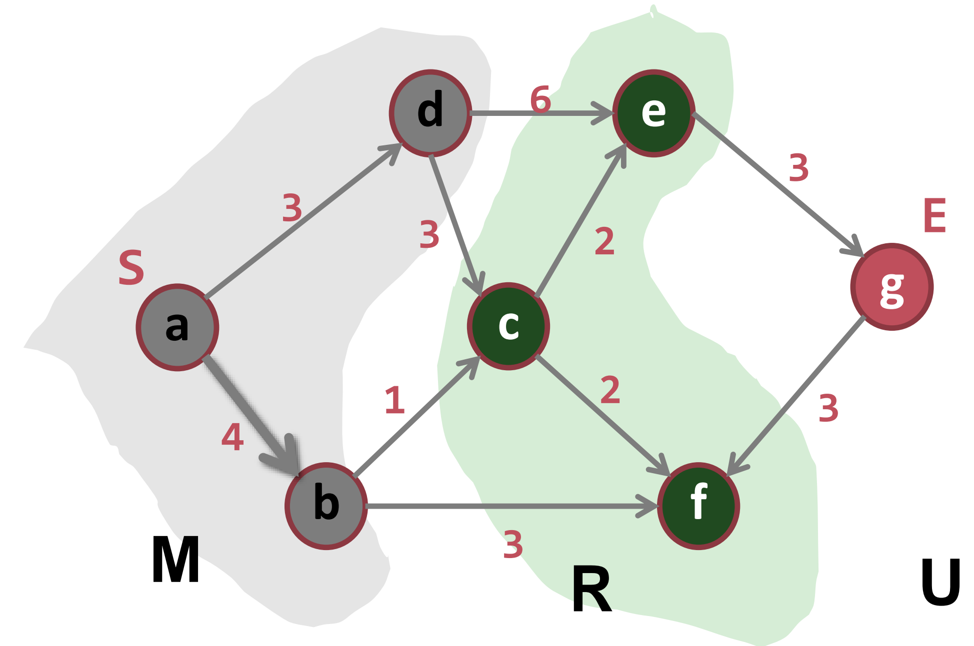
Modellierung

Graph = Menge von Knoten

Knoten mit ausgehenden Kanten und zuletzt ermittelter Pfadlänge (für den Algorithmus)

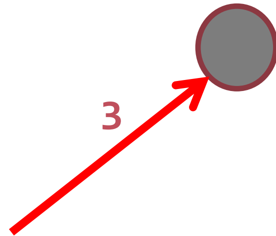
Kanten mit Länge und Zielknoten

MinHeap R mit schnellem Zugriff auf kürzesten Pfad

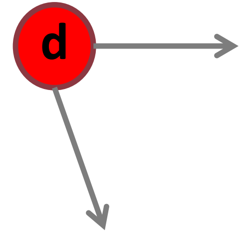


Kanten und Knoten

```
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 $x < \text{Pfadlänge}(N) < \infty$, so setze $\text{Pfadlänge}(N) = x$ und passe R an den neuen Wert an." ?

→ neue Methode DecreaseKey !

Schnelles Finden von Nodes im Heap?

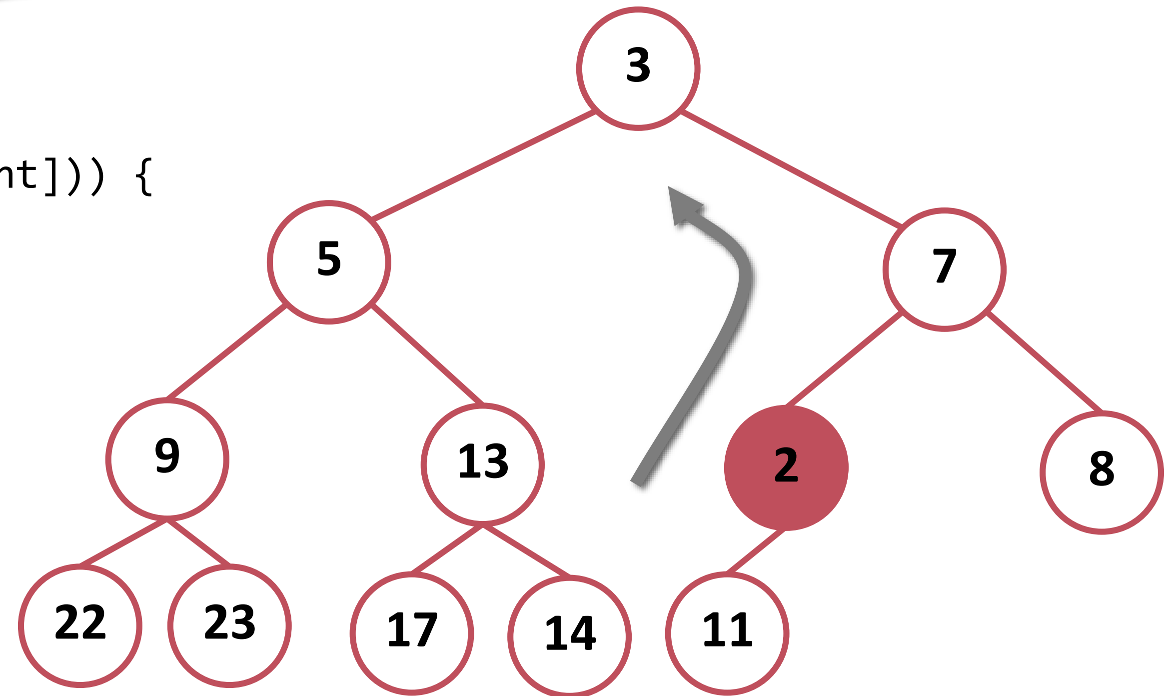
```
public class Heap {
   ...
   HashMap <Node, Integer> map;

   // 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

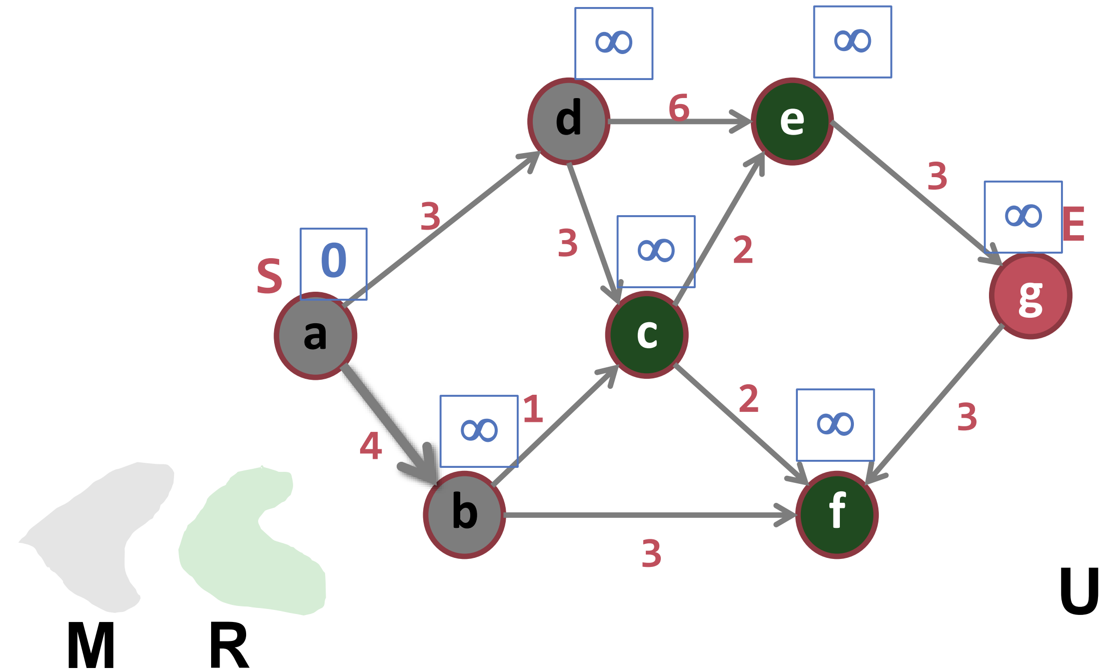
```
public class Heap {
    ...
    public void DecreaseKey(Node n)
    {
        int current = map.get(n);
        int parent = (current-1)/2;
        // aufsteigen
        while (current > 0 && Smaller(n, data[parent])) {
            Set(current, data[parent]);
            current = parent;
      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

Hier brauchen wir die Hashtabelle!



Implementation Algorithmus

```
LinkedList<Node> ShortestPath(Node S, Node E)
{
   // Initialisierung
   LinkedList<Node> path = new LinkedList<Node>();

   Heap R=new Heap();
   for (Node node: nodes)
   {
      node.SetPathLen(Integer.MAX_VALUE);
      node.SetPathParent(null);
   }
   S.SetPathLen(0);
   Node newNode = S;

...
```



Implementation Algorithmus

```
...
// Kernstück des Algorithmus
while (newNode != null && newNode != E){
  for (Edge edge: newNode.out){
    int newLength = newNode.GetPathLen() + edge.length;
    Node dest = edge.to;
    int prevLength = dest.GetPathLen();
    if (newLength < prevLength){
      dest.SetPathLen(newLength);
      dest.SetPathParent(newNode);
      if (prevLength == Integer.MAX_VALUE)
        R.Insert(dest);
      else
        R.DecreaseKey(dest);
    }
  }
newNode = R.ExtractRoot();
}
...
```

Solange noch neue Knoten hinzukommen

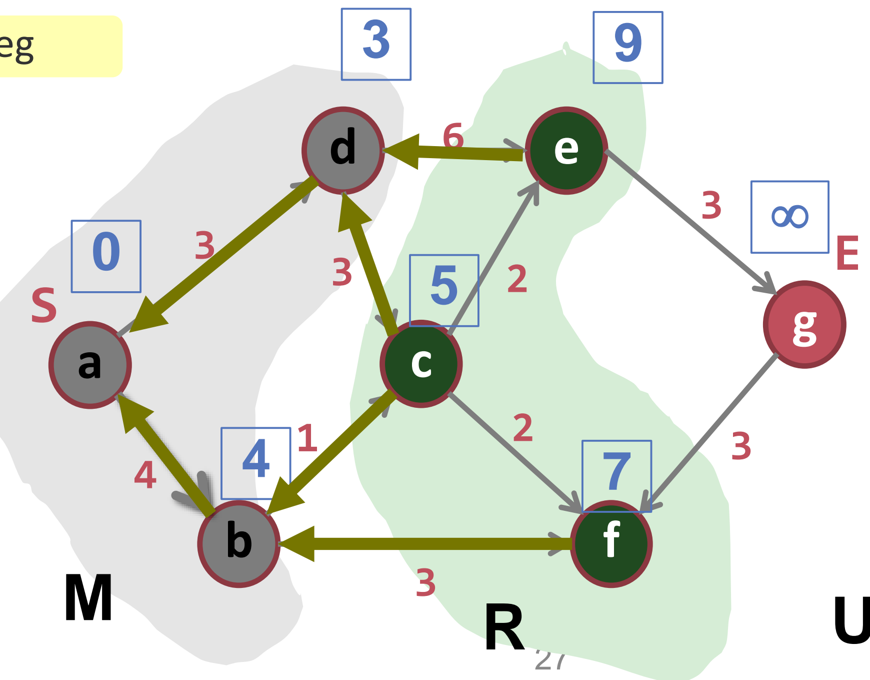
Für jede ausgehende Kante von newNode
- bestimme die Länge

Wenn der Pfad kürzer wird,
passe den aktuellen Wert an

und sorge für den Rückweg

passe ggfs. den Heap an

der nächste Knoten, welcher
hinzukommt ist der mit der kürzesten
Länge in R



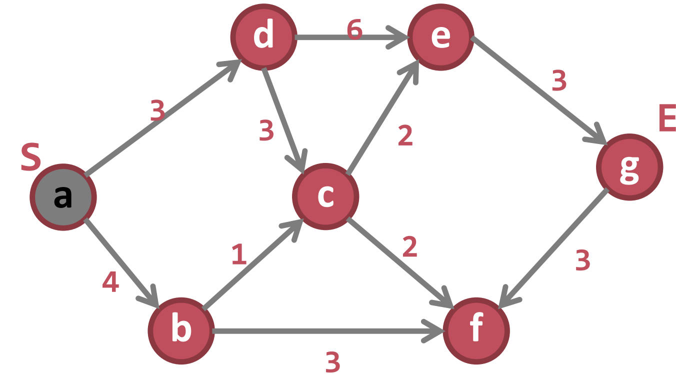
Implementation Algorithmus

```
...
  // Rückwärtstraversieren
  while (newNode != null) {
   path.push(newNode);
   newNode = newNode.GetPathParent();
  }

  return path;
 }
}
```

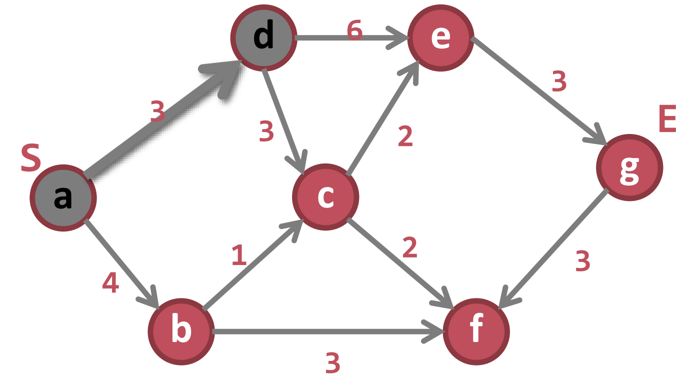
Appendix: Animation expandiert

Durchprobieren aller Pfade zu ineffizient

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

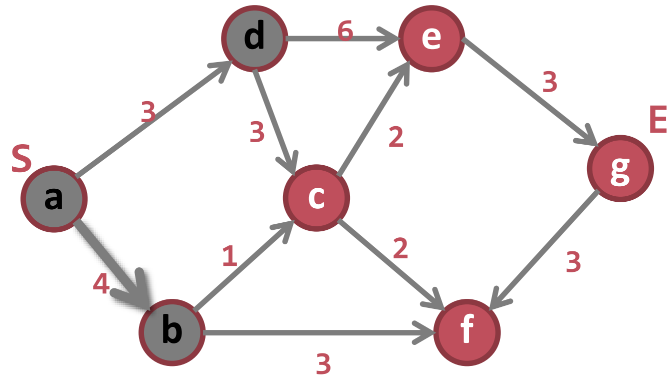


Starte bei S,
(Knoten, Pfadlänge) : (a,0)

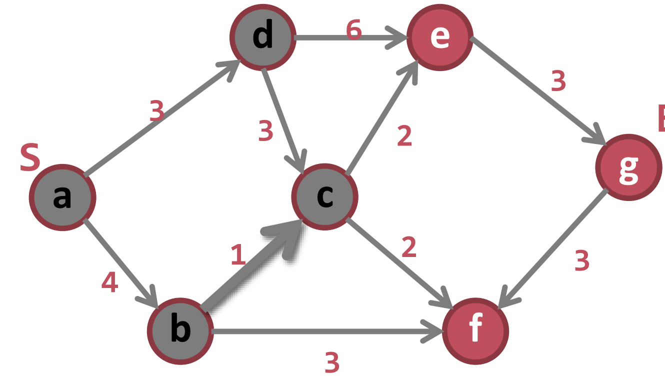


Kürzester Weg von (a,0)
Zusätzliche Kante
+(a-d,3)

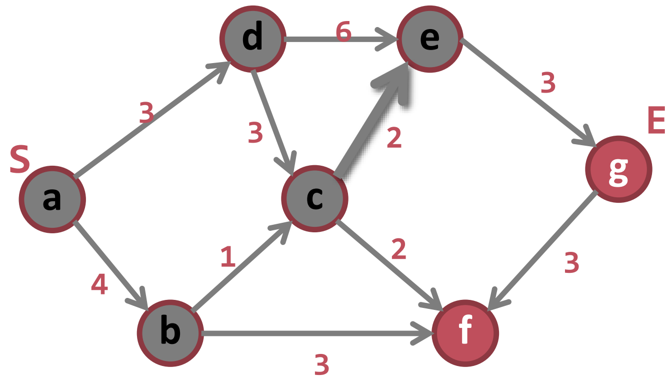
Appendix: Algorithmus Idee



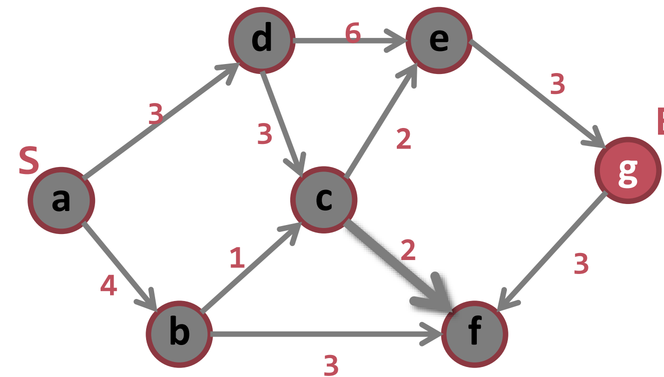
(a,0), (a-d,3) +(a-b,4)



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



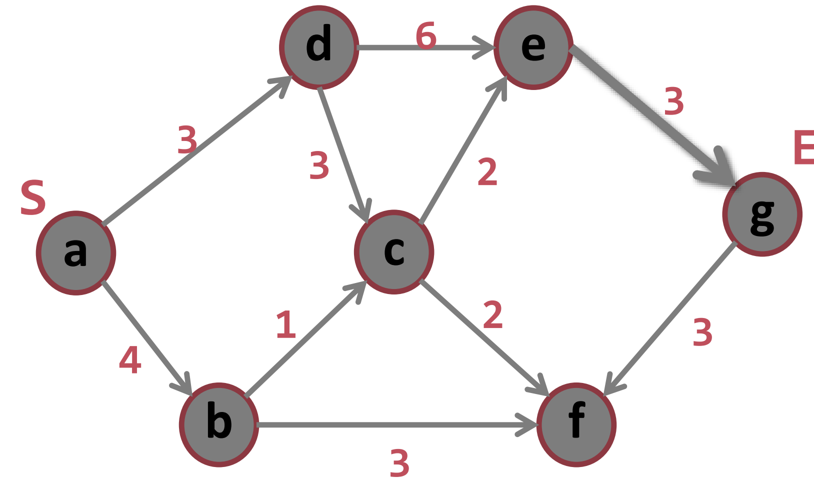
(a,0), (a-d,3), (a-b,4), (b-c,5)
+(c-e,7)



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

Appendix: 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