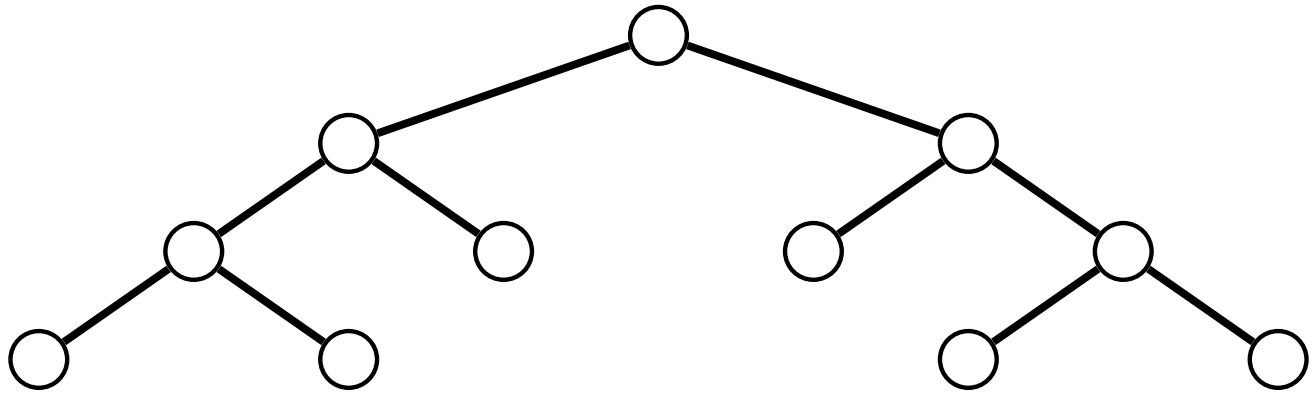
Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2018

Mai Bui

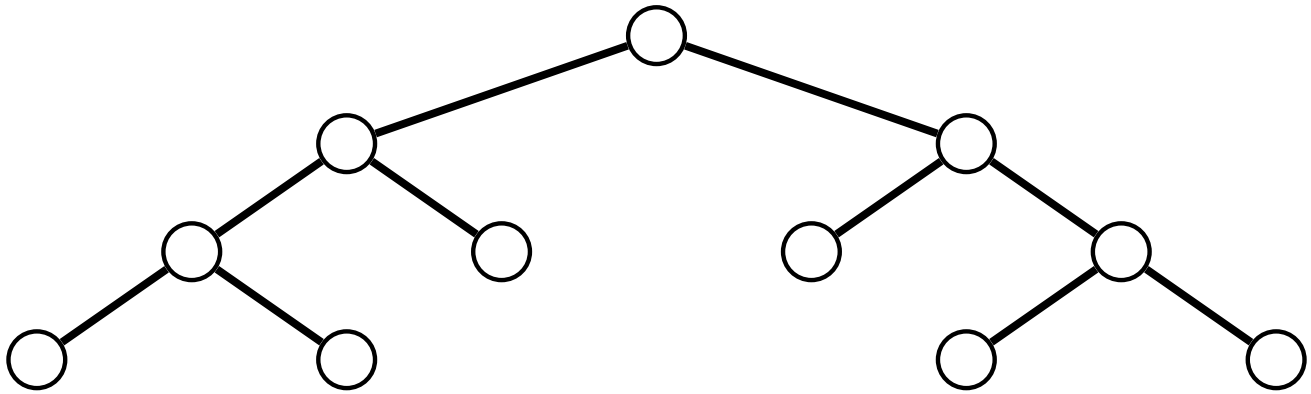
Computer Aided Medical Procedures
Technische Universität München

Bäume



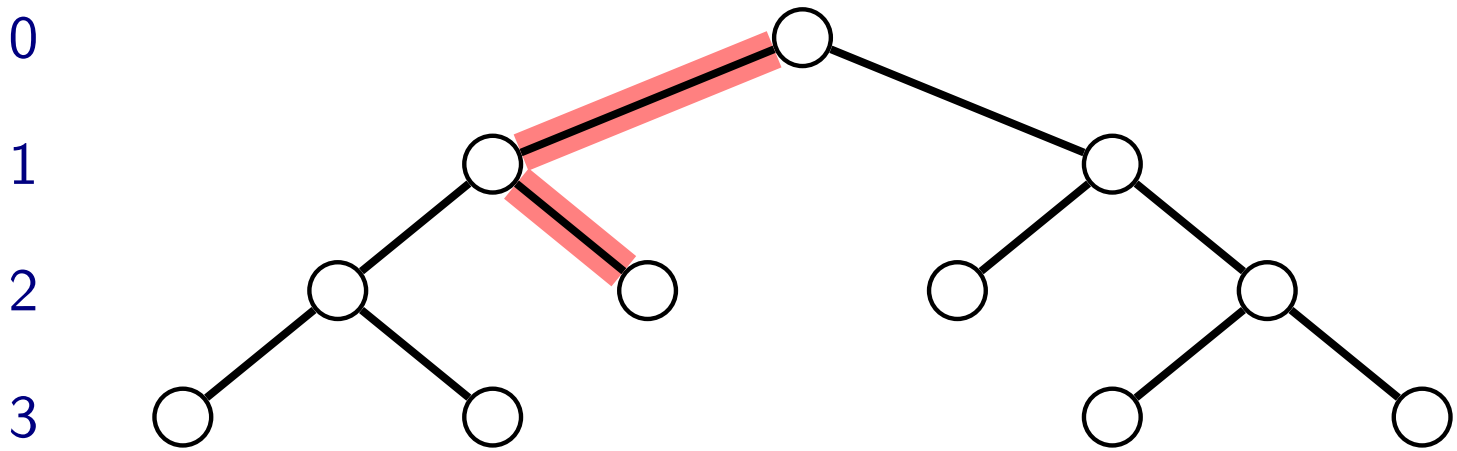
- Bäume sind *ungerichtete*, *azyklische*, *zusammenhängende* Graphen.
- Genau ein Knoten wird als *Wurzel* bezeichnet.
- Alle anderen Knoten haben genau eine Kante zu ihrem *Vaterknoten* v. Man nennt sie *Kind* von v.
- Die Anzahl der Kindknoten pro Vaterknoten ist nicht notwendigerweise eingeschränkt.
- Zwei Knoten sind durch genau einen Pfad verbunden.

Bäume



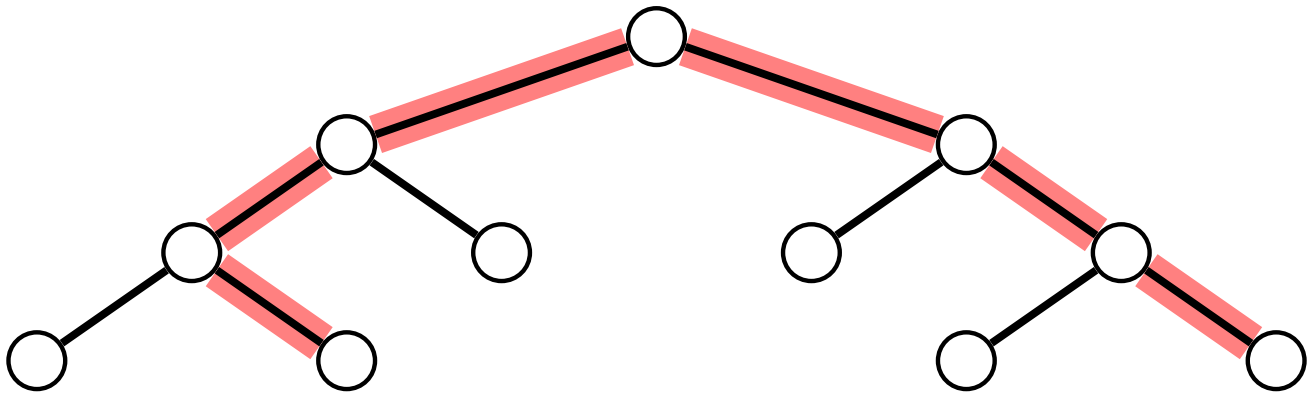
- Sei V die Menge der Knoten des Baumes und E die Menge der Kanten
- Falls $|V| = n$, dann $|E| = n - 1$, bzw. $|E| = |V| - 1$
  - Alle Knoten bis auf die Wurzel haben je eine Kante zum Vaterknoten!

Bäume



- Die *Tiefe* eines Knotens v ist die Länge des (eindeutigen) Pfads von der Wurzel zu v.
- Alle Knoten der gleichen Tiefe bilden eine *Ebene*.
- Die *Höhe* eines Baums ist die maximale Tiefe. Hier 3.

Bäume



- Der längste Pfad verbindet zwei *Blätter* der Ebene mit höchstem Index über die Wurzel.
- Die Länge entspricht also der doppelten Baumhöhe.

Binärbäume

- Ein Knoten hat 0, 1 oder 2 Kinder.
- Falls der Binärbaum vollständig ist und n Blätter hat, dann...
  - ...hat die Ebene darüber $\frac{n}{2}$ Knoten.
  - ...hat jede weitere Ebene jeweils halb so viele Knoten wie die nächst-tiefere.
  - ...errechnet sich die Höhe x als Lösung von $2^x = n$.

  Insgesamt ergibt sich also eine Höhe $x = \log_2 n$.

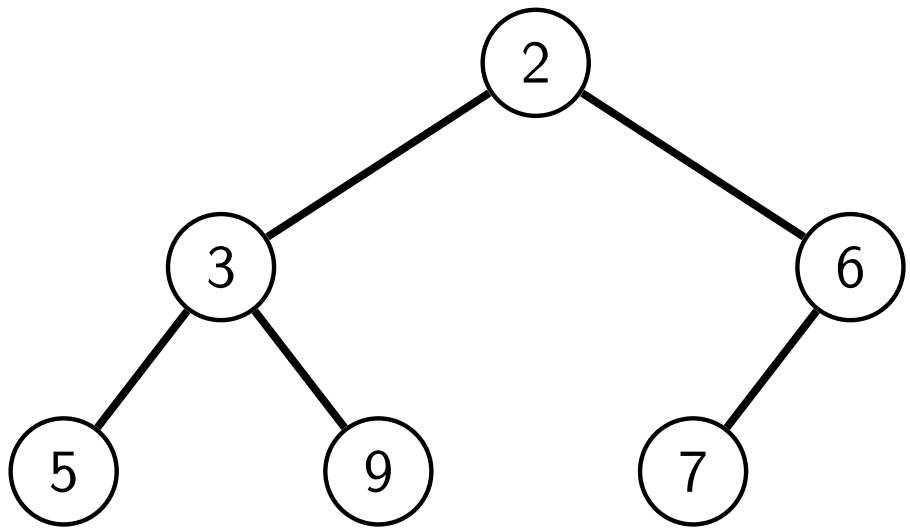
Komplexität von Merge und Quick Sort

- Beide teilen das Problem jeweils hälftig auf. (Zumindest besteht beim Quick Sort die Hoffnung!)
- Beide haben im Mittel die Komplexität $O(n \log n)$.
- Der Faktor $\log n$ ergibt sich aus der binär(-baum-artig)-en Aufteilung.
- Der Faktor n ergibt sich...
  - ...beim Merge Sort durch die Zusammensortierung der einzeln sortieren Teile.
  - ...beim Quick Sort durch die Partitionierung gemäß Pivotwert.
- Diese Betrachtung ist natürlich informell!

Heap

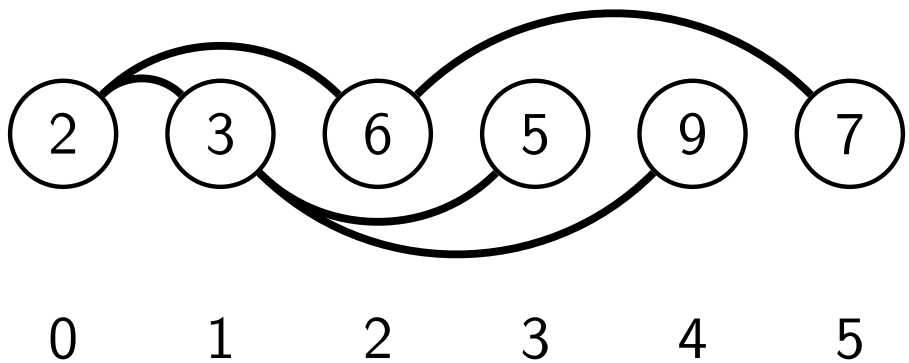
Der *Min-Heap* ist...

- ...ein Binärbaum, ...
- ...fast vollständig, und...
- ...der jedem Vaterknoten zugeordnete Wert $key(v)$ ist immer kleiner oder gleich den Werten der Kindknoten $key(c_l)$ und $key(c_r)$.



Anordnung im Speicher

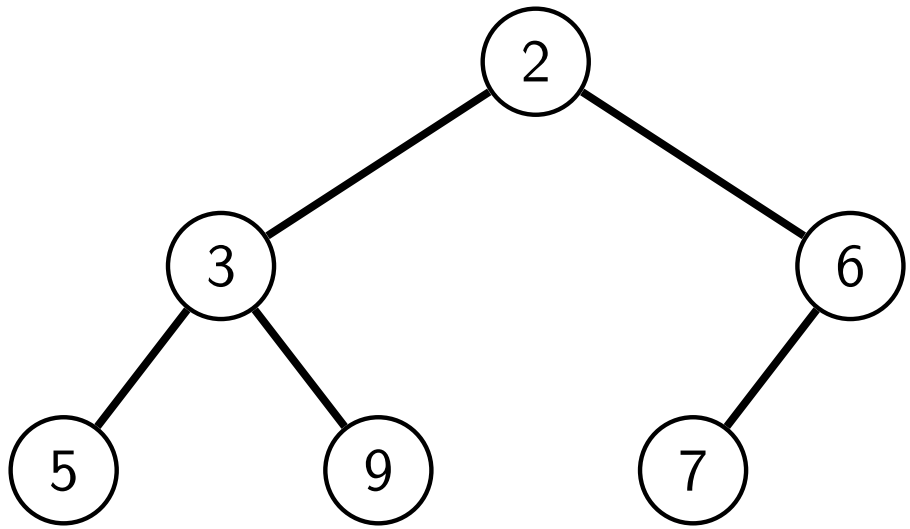
- Wie verkettete Liste mit Zeigern?
- Linearisiert dank (annähernder) Vollständigkeit!
  - Wurzel bei Index 0
  - Vaterknoten zu gegebenem Index i bei $\lfloor (i-1)/2 \rfloor$
  - Kindknoten bei $2 \cdot i + 1$ und $2 \cdot i + 2$




Code

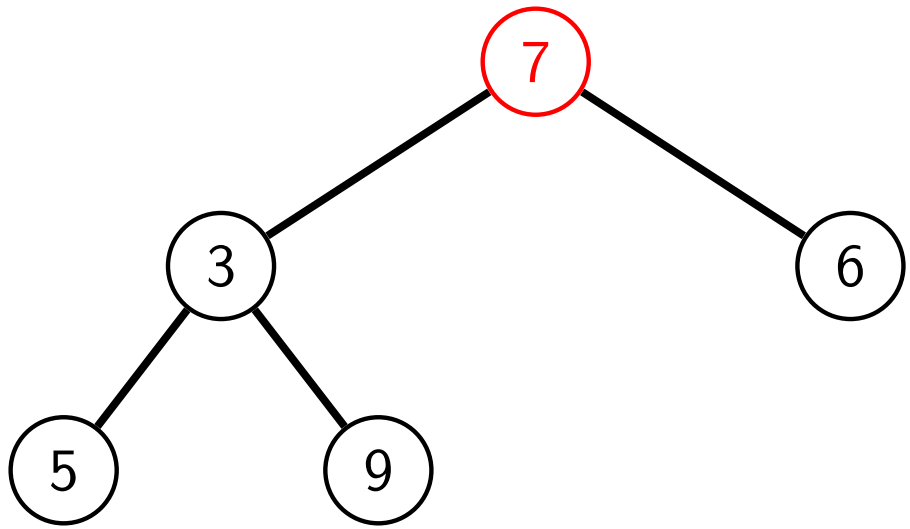
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



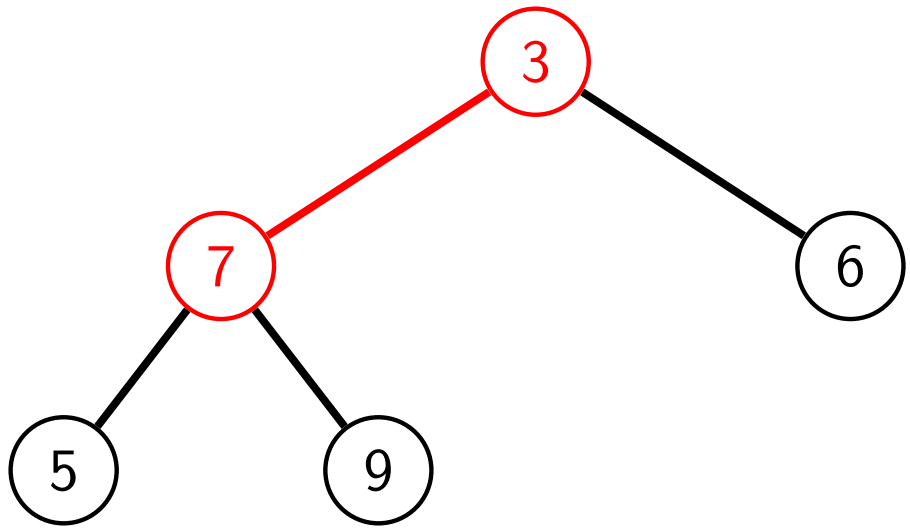
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



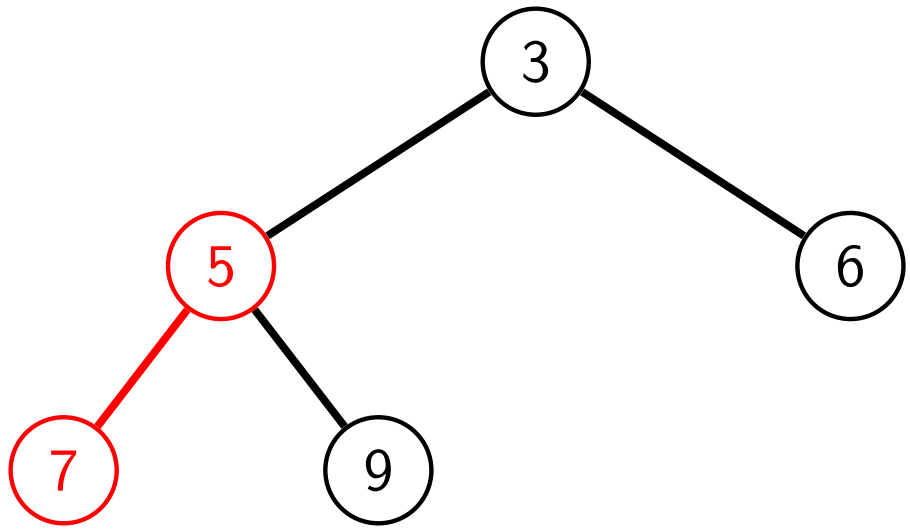
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



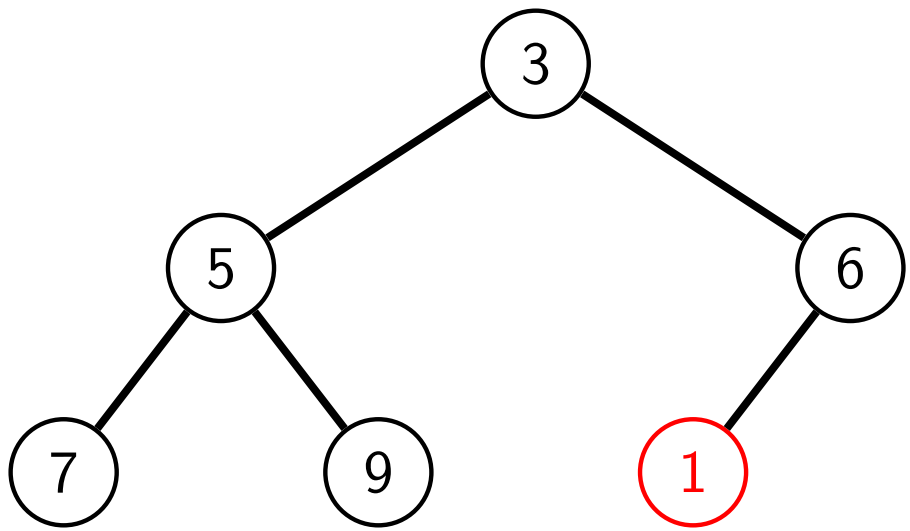
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



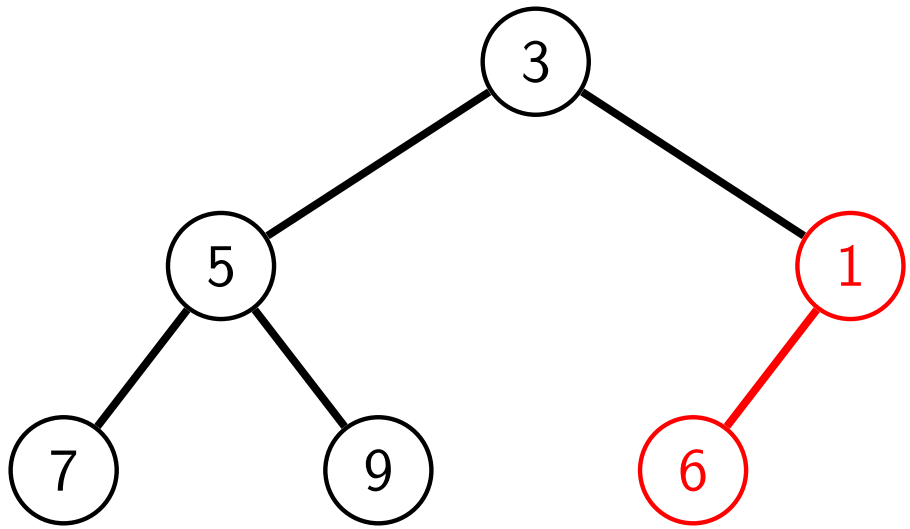
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



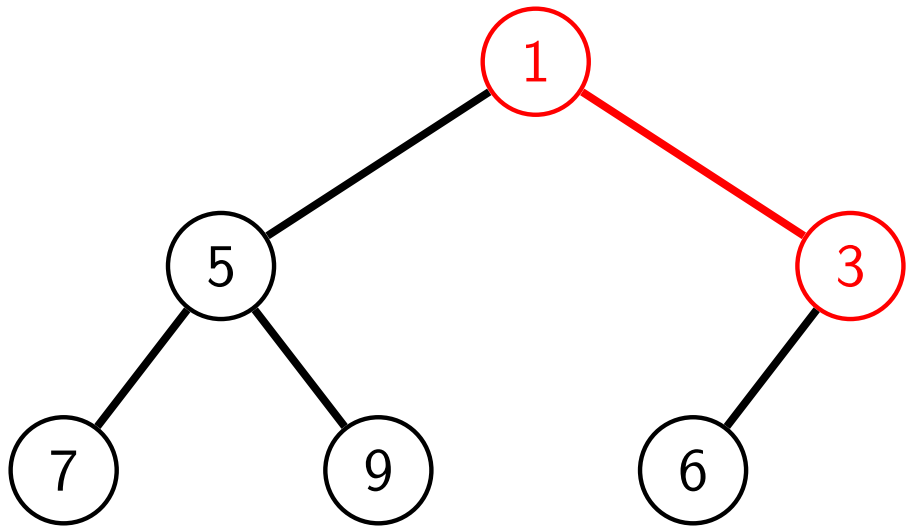
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Werte „absinken“ (**minHeapify**)
- **insert** fügt den neuen Wert als letztes Blatt ein und läßt kleine Wert „aufsteigen“



Heap Sort

- Idee: Behandle Array als Binärbaum und stelle Heap-Eigenschaft her (**buildMinHeap**)
- Möglichkeit 1: Lese wiederholt das Minimum aus (**extractMin**)
  - Aufsteigende Ordnung bei Min-Heap
  - Zusätzlicher Speicherplatz erforderlich
- Möglichkeit 2: Tausche Wurzel ans Ende, ignoriere diesen Wert in Zukunft, und lasse dann die neue große Wurzel absinken (**minHeapify**)
  - Absteigende Ordnung bei Min-Heap
  - In-place

Heap erzeugen: buildMinHeap

- erzeuge fast vollständigen Binärbaum aus V
- wende **minHeapify** auf alle Knoten $v \in V$ an, von unten nach oben (nicht nötig für unterste Ebene des Baumes!)

Input: Knoten-Liste V
Output: Min-Heap G
buildMinHeap(V):
    G = erzeuge beliebigen fast vollständigen Binärbaum aus V;
    for each Knoten v in G von unten nach oben {
        minHeapify(G, v);
    }

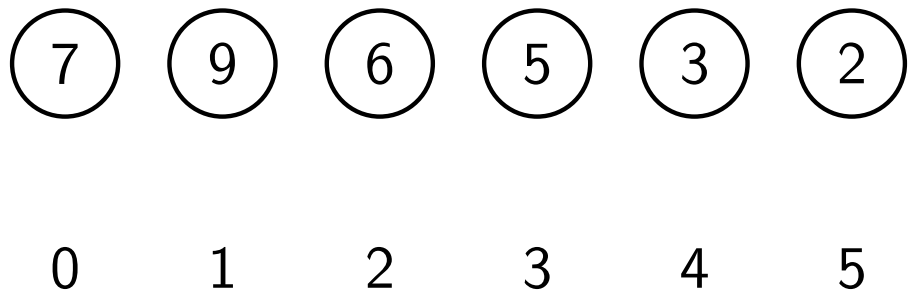
Heap: minHeapify

- Operation **minHeapify** auf Knoten $v \in V$ zur Wiederherstellung der Min-Heap-Eigenschaft
- lasse v durch Heap absinken, bis Min-Heap-Eigenschaft wiederhergestellt

Input: Knoten v
minHeapify(G, v):
    if (v ist Blatt) **return**;
    $knoten$ = Minimum (bzgl. key) von v.links und v.rechts;
    if ($key(knoten) < key(v)$) {
        tausche Inhalt von $knoten$ und v;
        minHeapify$(G, knoten)$;
    }

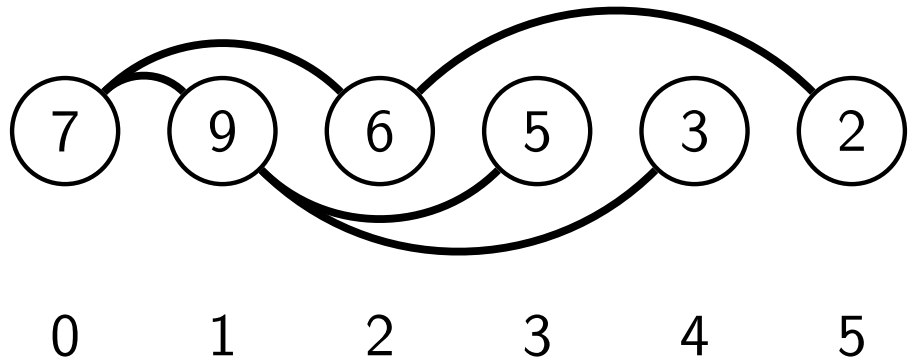
Heap Sort

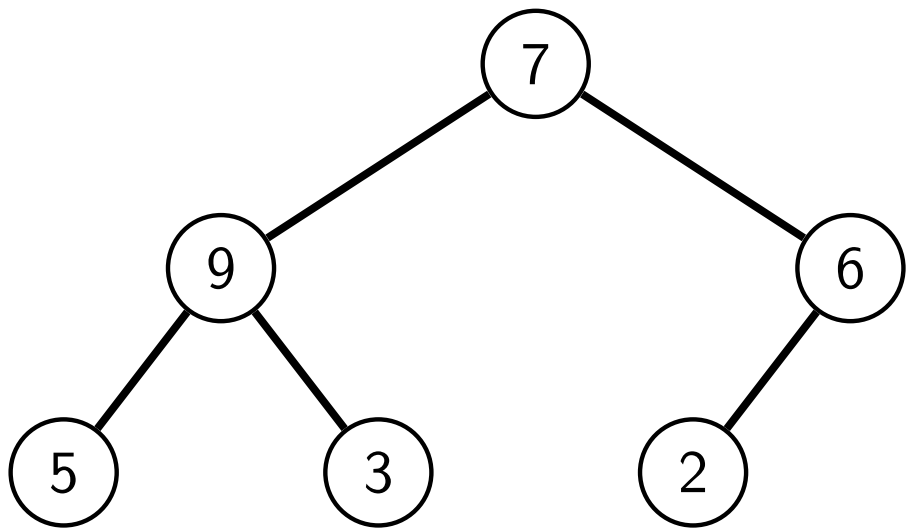
- Eingabedaten



Heap Sort

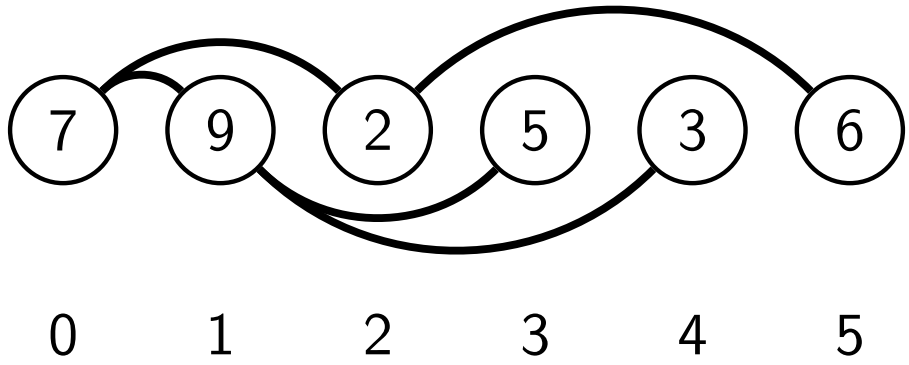
- Interpretation als (fast) vollständiger Binärbaum

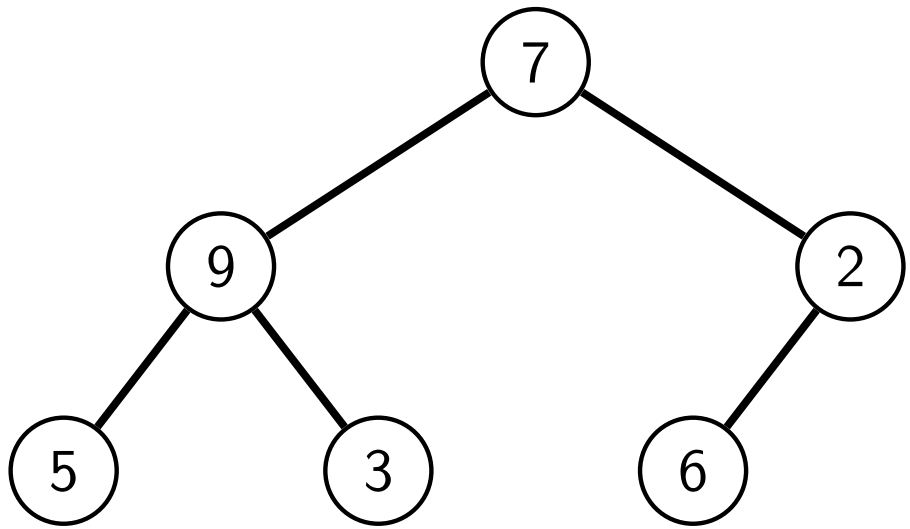




Heap Sort

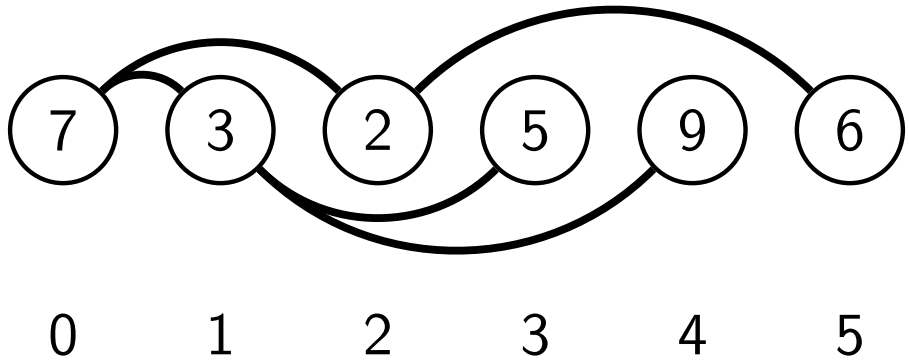
- Heap-Eigenschaft herstellen via **buildMinHeap**

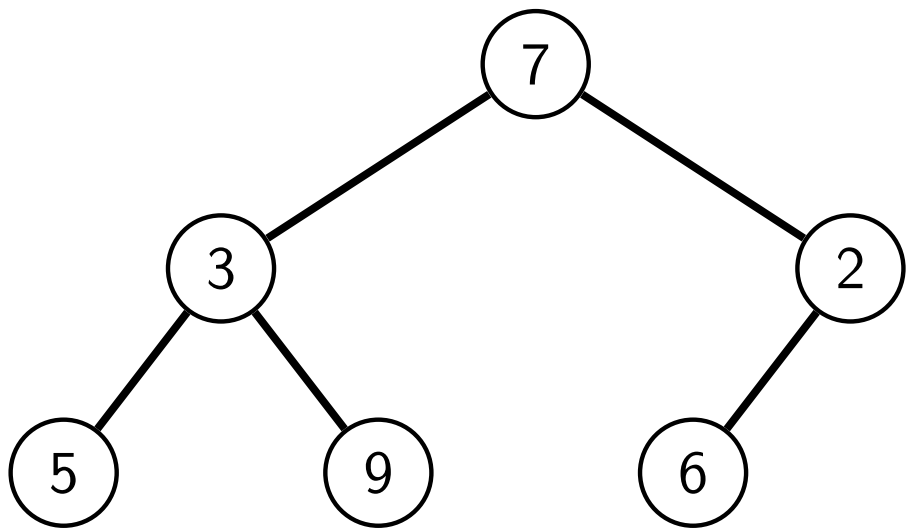




Heap Sort

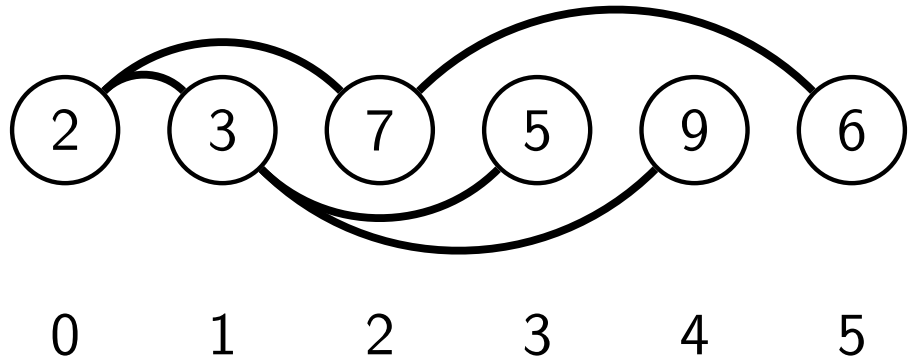
- Heap-Eigenschaft herstellen via **buildMinHeap**

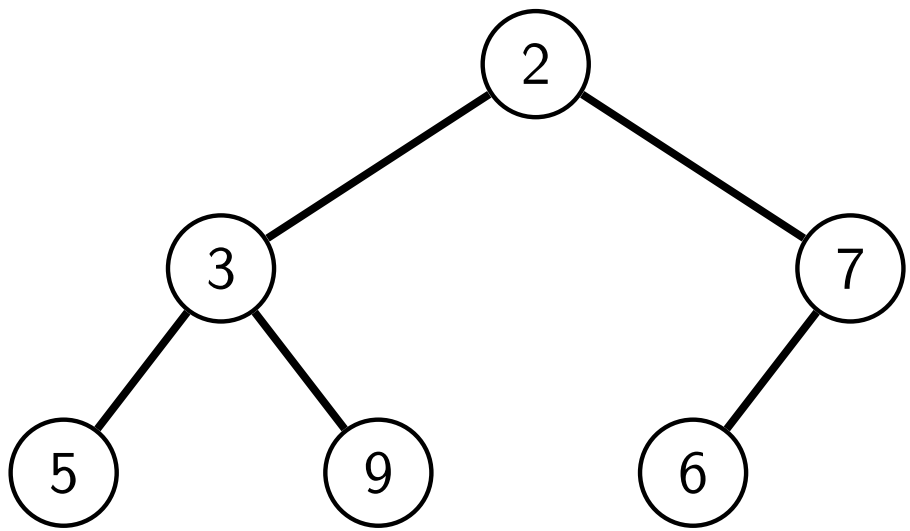




Heap Sort

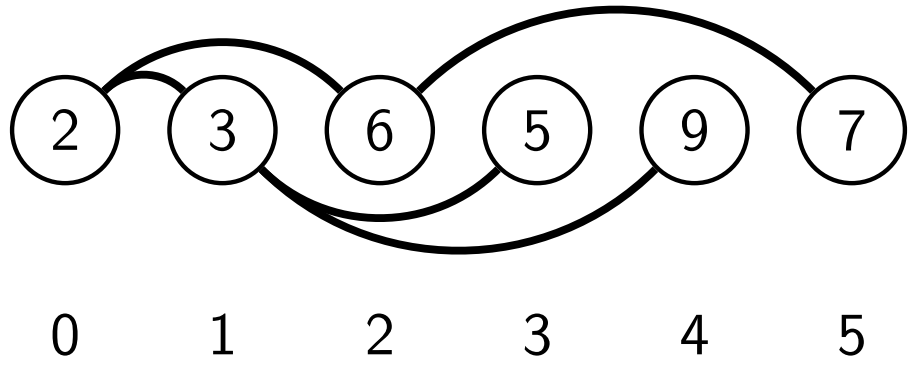
- Heap-Eigenschaft herstellen via **buildMinHeap**

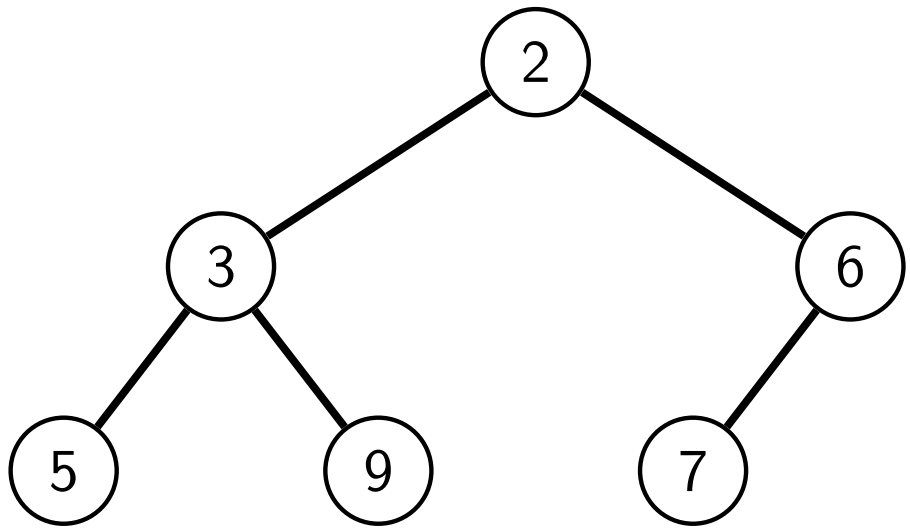




Heap Sort

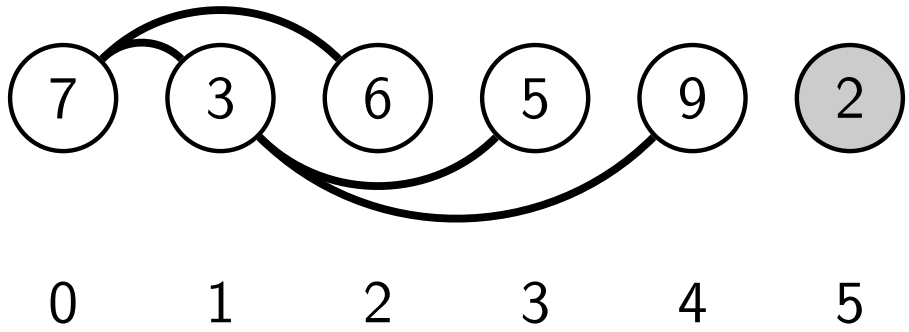
- Heap-Eigenschaft herstellen via **buildMinHeap**

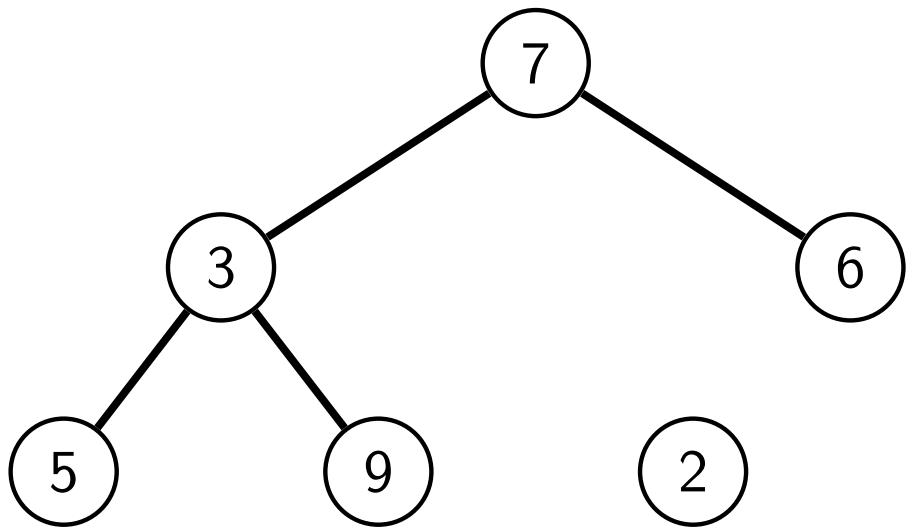




Heap Sort

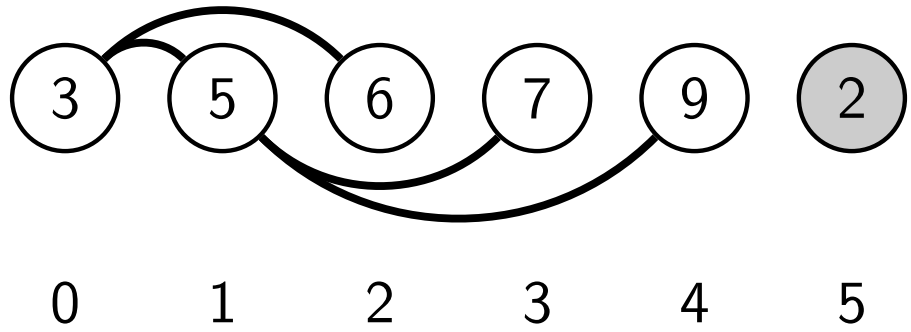
- Wurzel mit Ende vertauscht, „Kürzung“

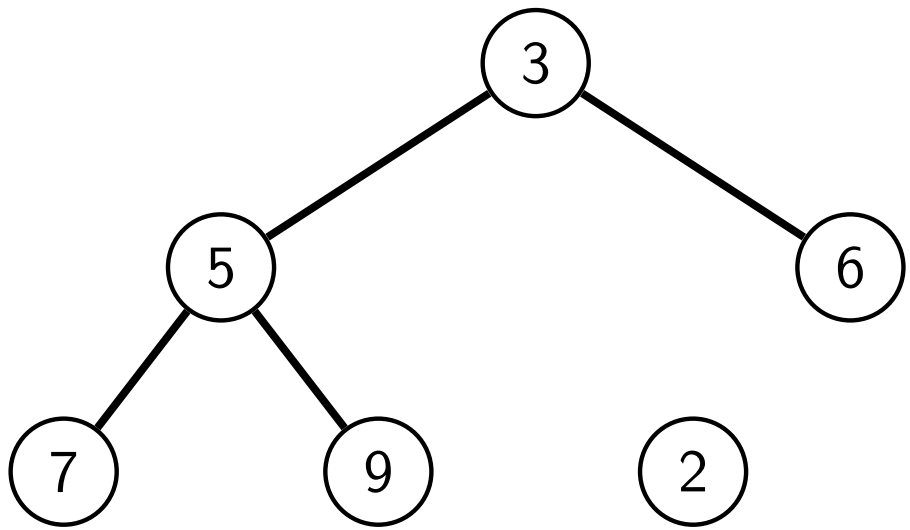




Heap Sort

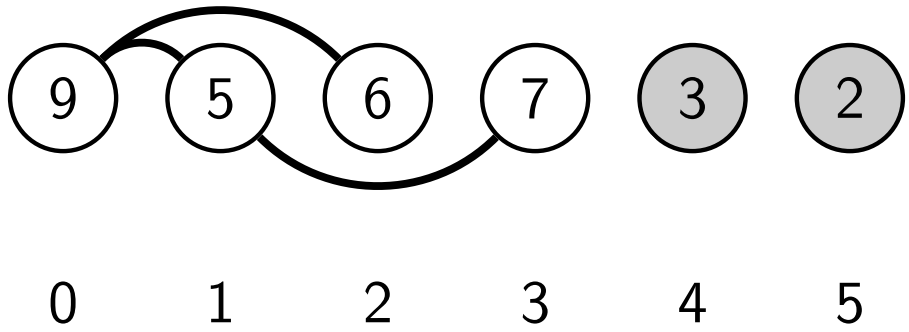
- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel

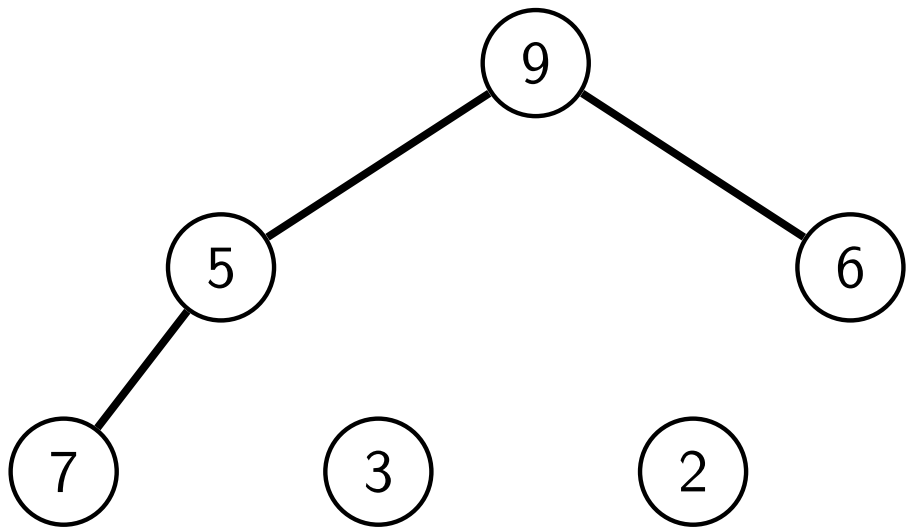




Heap Sort

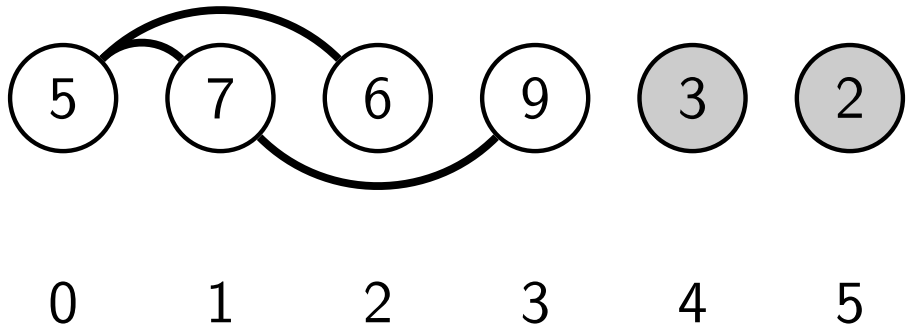
- Wurzel mit Ende vertauscht, „Kürzung“

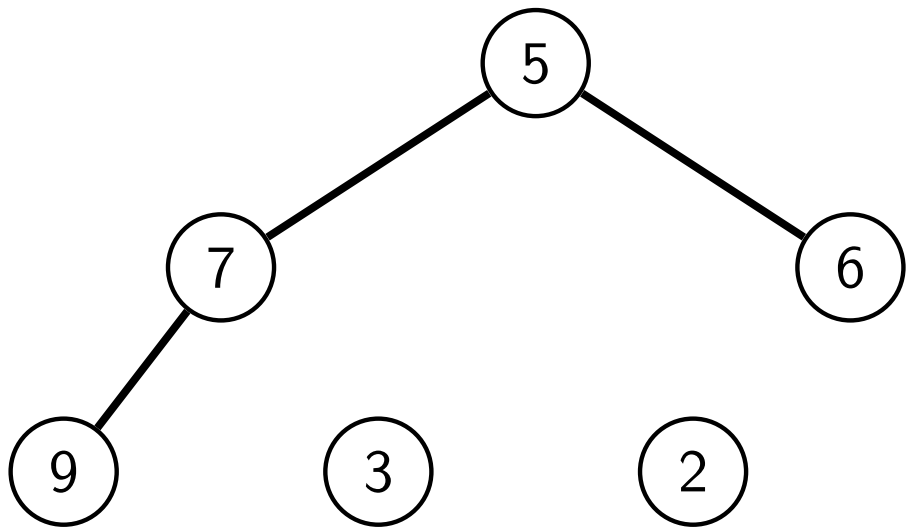




Heap Sort

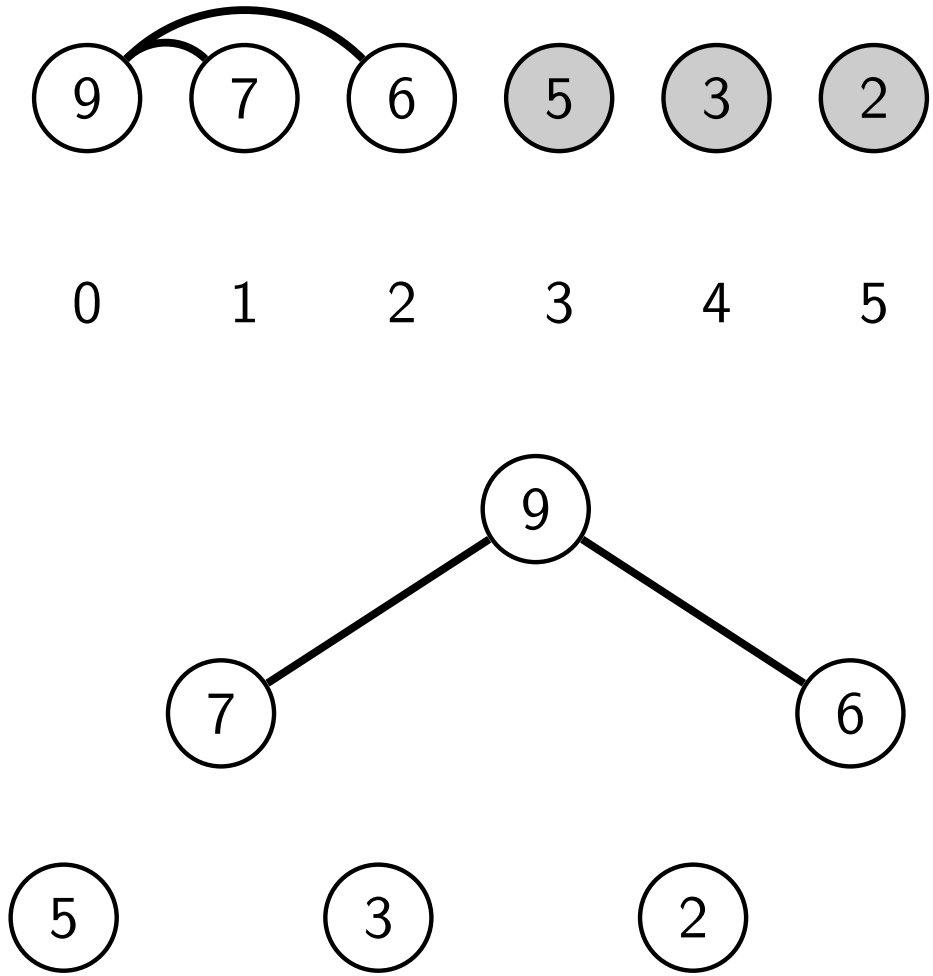
- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel





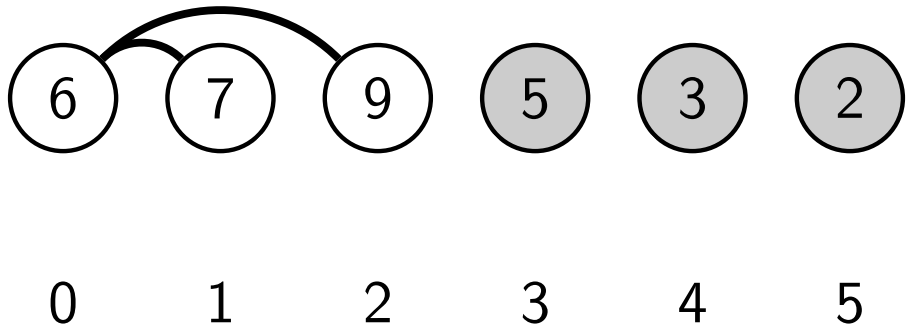
Heap Sort

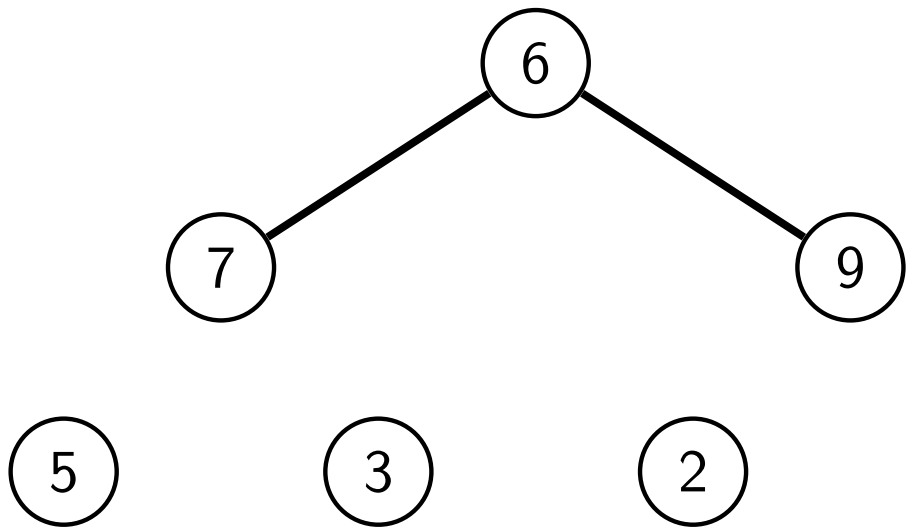
- Wurzel mit Ende vertauscht, „Kürzung“



Heap Sort

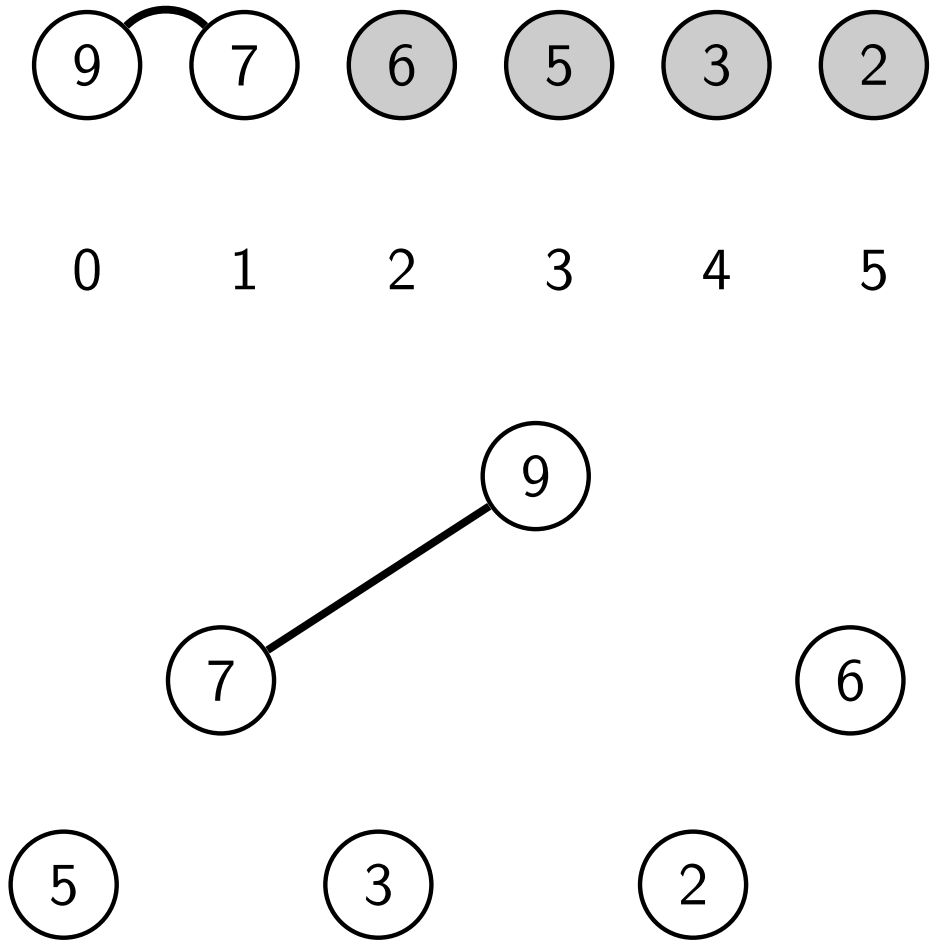
- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel





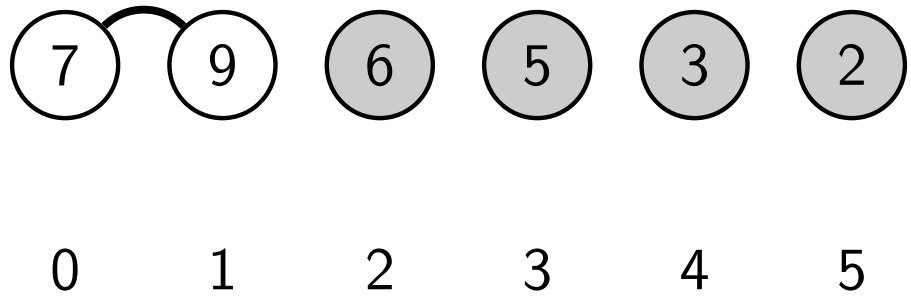
Heap Sort

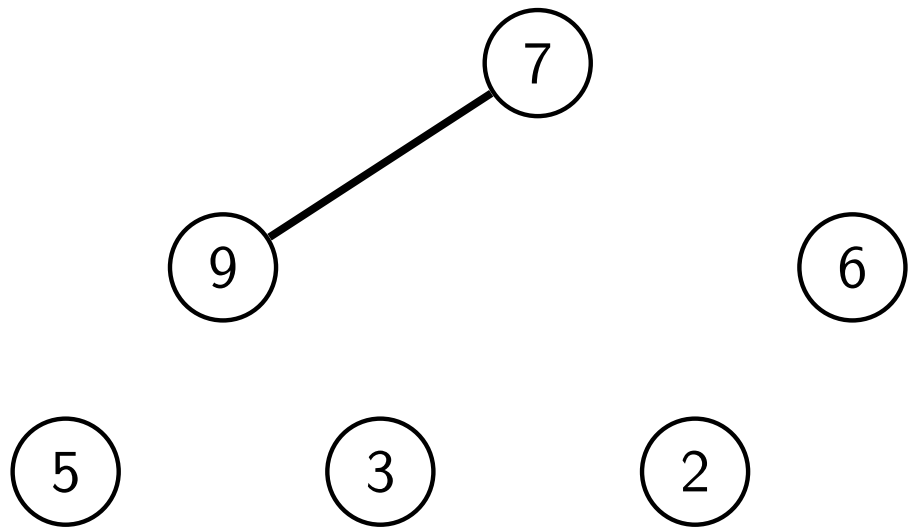
- Wurzel mit Ende vertauscht, „Kürzung“



Heap Sort

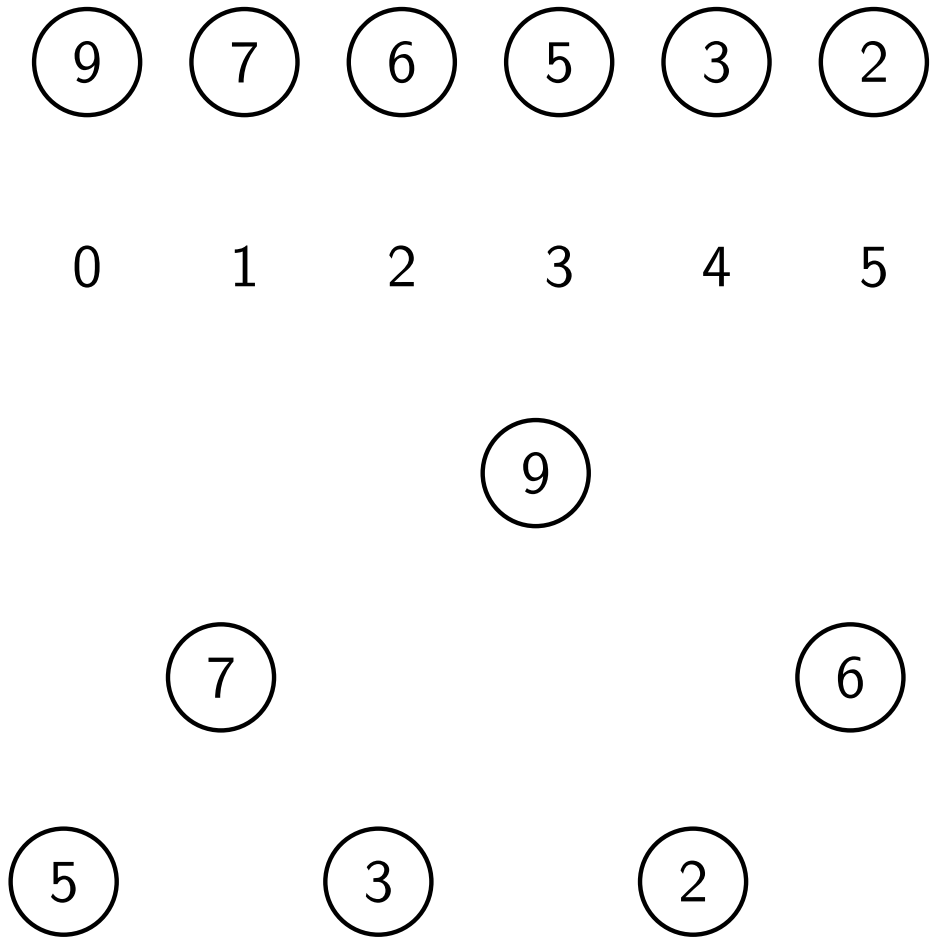
- Heap-Eigenschaft hergestellt via **minHeapify** ab Wurzel





Heap Sort

- Wurzel mit Ende vertauscht, **absteigende Ordnung!**




Code

Pfadsuche

- Beispiel für einen Graphenalgorithmus: Suche kürzeste bzw. günstigste Pfade in einem gewichteten Graphen!
- *Single Pair Shortest Path (SPSP)*: Finde günstigsten Pfad zwischen zwei Knoten!
- *Single Source Shortest Path (SSSP)*: Finde günstigste Pfade zu allen Knoten von einem einzigen Startknoten aus!

- Dijkstra-Algorithmus!

Dijkstra-Algorithmus

$pred = \{\emptyset, ..., \emptyset\}$; $d = \{\infty, ..., \infty\}$; $d[v_{\text{start}}] = 0$;
MinPriorityQueue Q mit Elementen v, Schlüsseln d;

while ($\neg$isEmpty(Q)) {
    v = extractMin(Q);

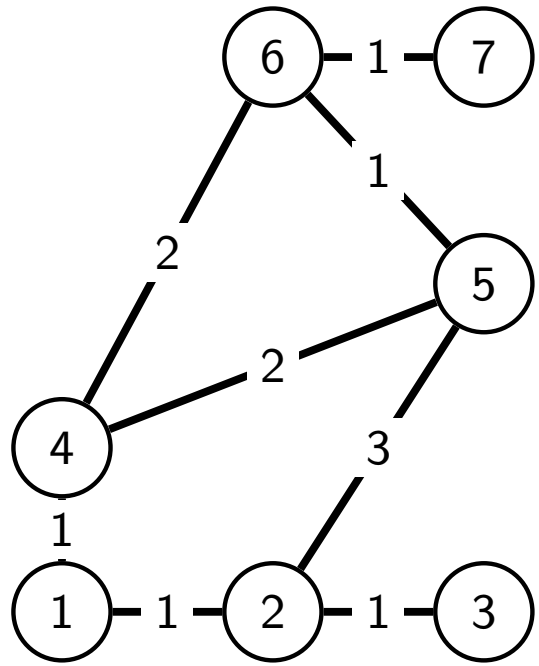
    // Ankunft am Zielknoten bei SPSP
    if ($v = v_{\text{stop}}$) {
        return;
    }

    // Für jeden unbesuchten Nachbarn n
    foreach (Knoten $n : \{v, n\} \in E \wedge n \in Q$) {
        // Günstigeren Pfad nach n über v gefunden
        if ($d[v] + w[\{v, n\}] < d[n]$) {
            $pred[n] = v$; $d[n] = d[v] + w[\{v, n\}]$;
            decreaseKey($Q, n, d[n]$);
        }
    }
}

Beispiel: Kürzester Pfad von 1 nach 7

- Initialisiere Startknoten 1.

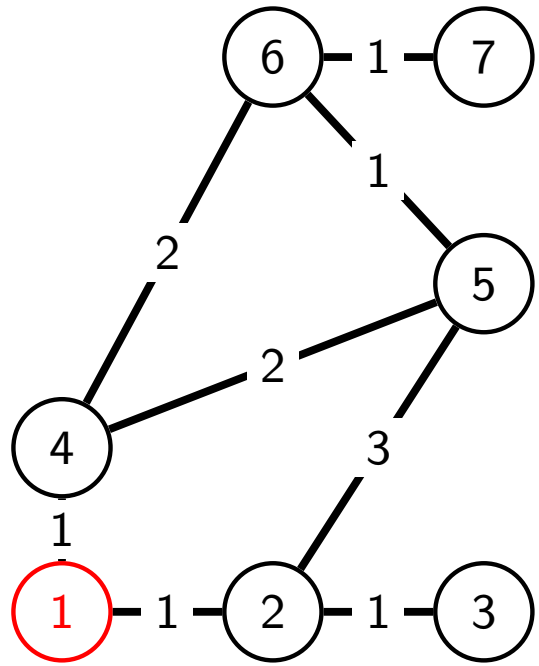
	1	2	3	4	5	6	7
pred							
dist	0	∞	∞	∞	∞	∞	∞
Q	1	2	3	4	5	6	7



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 1, da minimale Distanz.

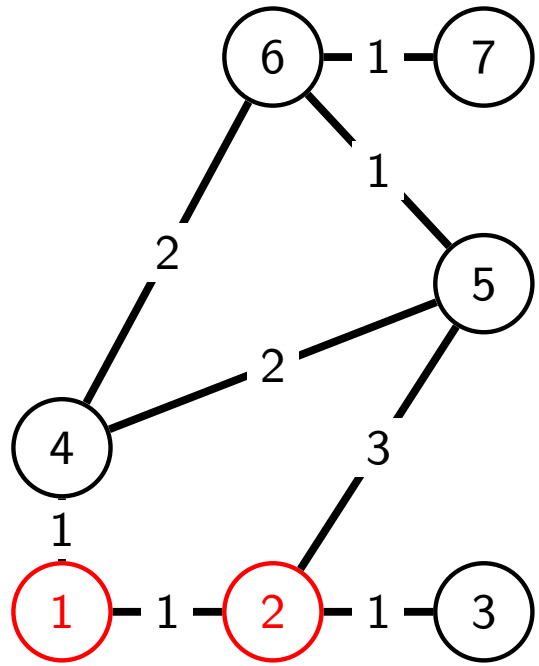
	1	2	3	4	5	6	7
pred		1		1			
dist	0	1	∞	1	∞	∞	∞
Q	2	4	3	5	6	7	



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 2.

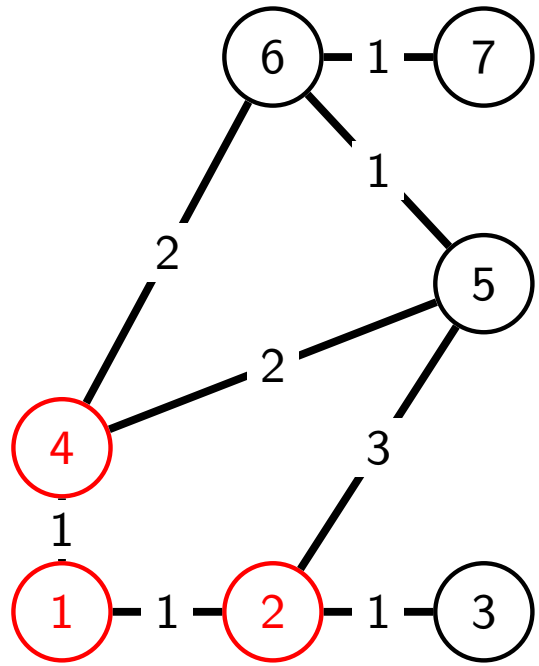
	1	2	3	4	5	6	7
pred		1	2	1	2		
dist	0	1	2	1	4	∞	∞
Q	4	3	5	6	7		



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 4, Update für Knoten 5.

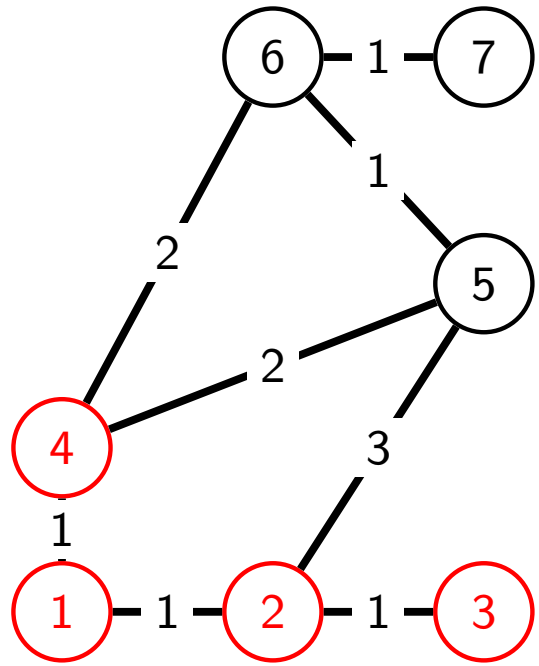
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	3	5	6	7			



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 3, keine unbesuchten Nachbarn.

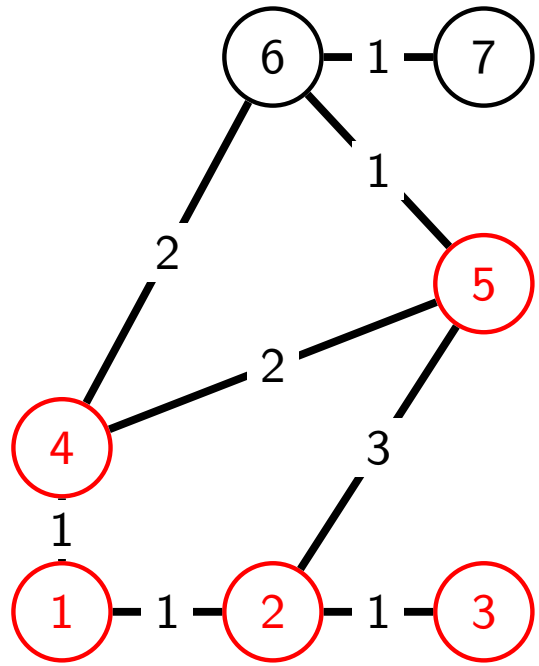
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	5	6	7				



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 5, Pfad zu Knoten 6 über 5 schlechter.

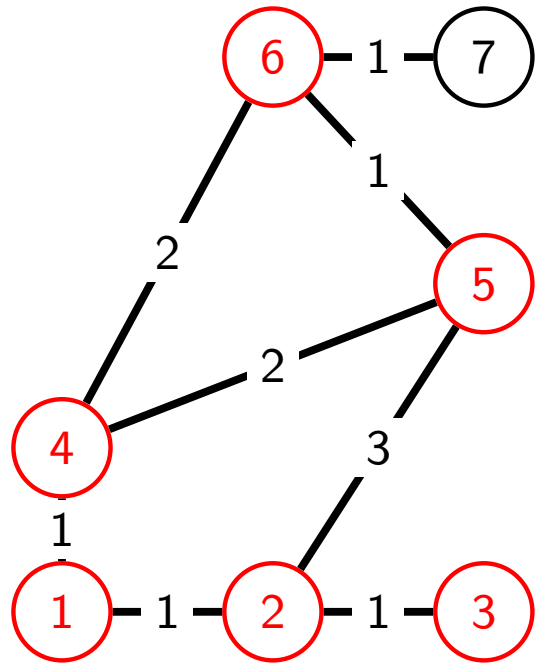
	1	2	3	4	5	6	7
pred		1	2	1	4	4	
dist	0	1	2	1	3	3	∞
Q	6	7					



Beispiel: Kürzester Pfad von 1 nach 7

- Besuche Knoten 6.

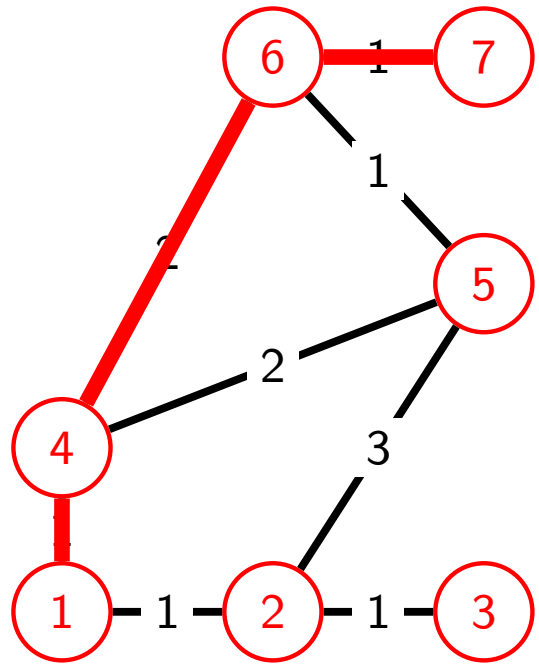
	1	2	3	4	5	6	7
pred		1	2	1	4	4	6
dist	0	1	2	1	3	3	4
Q	7						



Beispiel: Kürzester Pfad von 1 nach 7

- Erreiche Ziel 7, Ergebnis $1 \to 4 \to 6 \to 7$ mit Kosten 4.

	1	2	3	4	5	6	7
pred		1	2	1	4	4	6
dist	0	1	2	1	3	3	4
Q							




Code