Übung zu
Algorithmen und Datenstrukturen (für ET/IT)

Wintersemester 2012/13

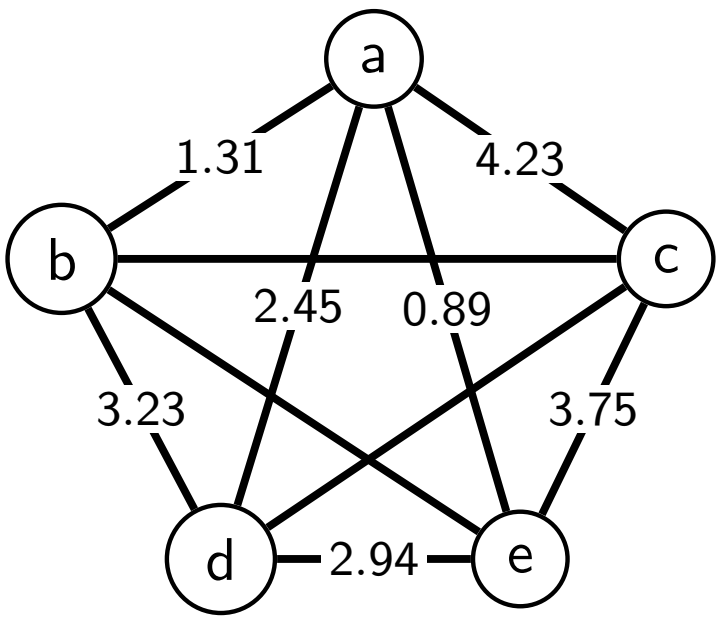
Jakob Vogel

Computer-Aided Medical Procedures
Technische Universität München

Adjazenz-Matrix

Wir betrachten Graphen mit folgenden Eigenschaften:

- ungerichtet (Adjazenz-Matrix ist *symmetrisch*!)
- gewichtet (Matrix enthält Gewichte!)
- vollständig (Matrix enthält viele unterschiedliche Werte!)



	a	b	c	d	e
a	∞	1.31	4.23	2.45	0.89
b	1.31	∞	0.27	3.23	1.98
c	4.23	0.27	∞	1.77	3.75
d	2.45	3.23	1.77	∞	2.94
e	0.89	1.98	3.75	2.94	∞

Speicherung von Matrizen

- Betrachte Matrix (oder Bitmap-Bild)

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

- *Row-Major*-Speicherung in linearem Speicher

1	2	3	4	5	6

  Normalerweise benutzt für Bitmap-Bilder, manchmal für Matrizen.

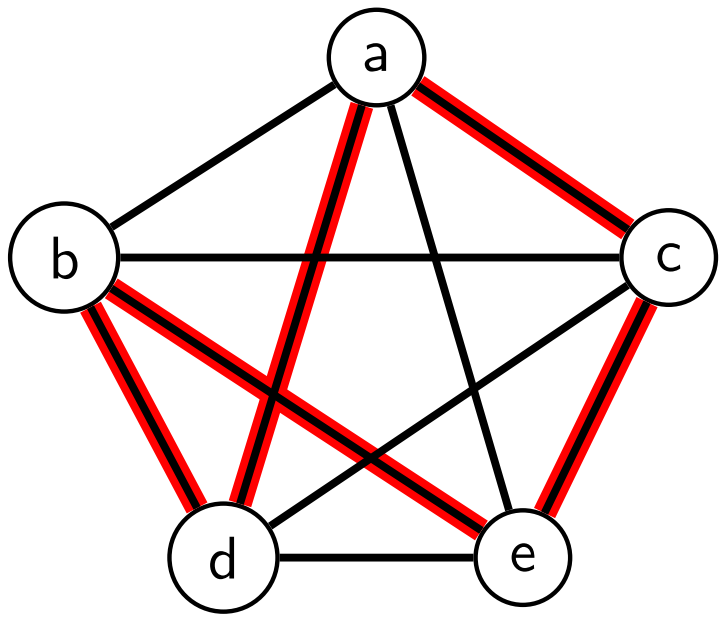
- *Column-Major*-Speicherung in linearem Speicher

1	4	2	5	3	6

  Im HPC-Bereich oft benutzt für Matrizen.

Traveling Salesman Problem (TSP)

- Gegeben ist ein gewichteter Graph, vielleicht auch gerichtet oder nicht vollständig
- Beispiel: Straßenkarte mit Kilometerangaben
- **Problem:** Finde Rundreise, so dass jeder Knoten (jede Stadt) genau einmal besucht wird, und die Gesamtstrecke möglichst kurz ist!



Traveling Salesman Problem (TSP)

- Die Beschreibung des Problems ist trivial!
- Die exakte Lösung aber extrem komplex!
  - Verwende viel Zeit zur expliziten Untersuchung aller Möglichkeiten!
  - Nutze Heuristiken, riskiere aber, bessere Lösungen zu übersehen!

TSP-Lösung via Back-Tracking

```
TSP(trip)
    if (trip besucht jede Stadt) {
        erweitere trip um Reise zum Startort;
        speichere trip als Rundreise;
    } else {
        foreach (unbesuchte Stadt s) {
            trip' = trip erweitert um s;
            TSP(trip');
        }
    }
```

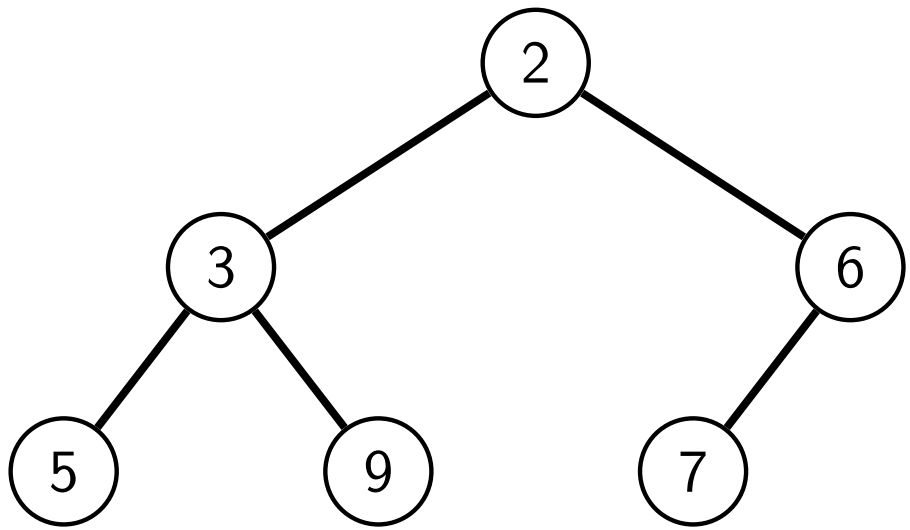
- Berechne alle Erweiterungen einer gegebenen Teil-Lösung
- Wiederhole das rekursiv, bis alle Lösungen bekannt sind


Code

Heap

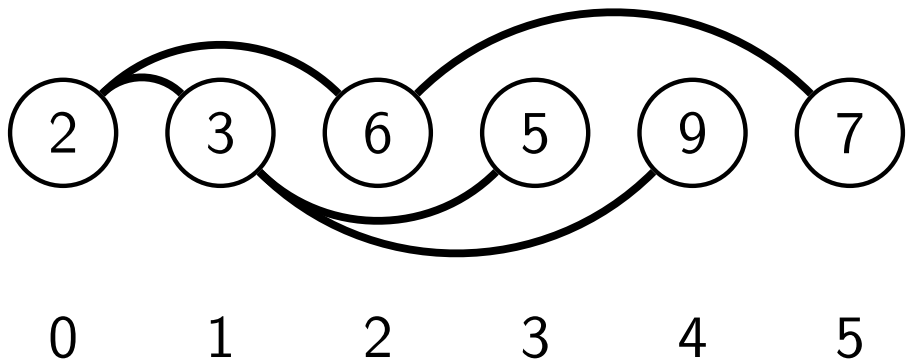
Der *Min-Heap* ist...

- ...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 $key(c_r)$.



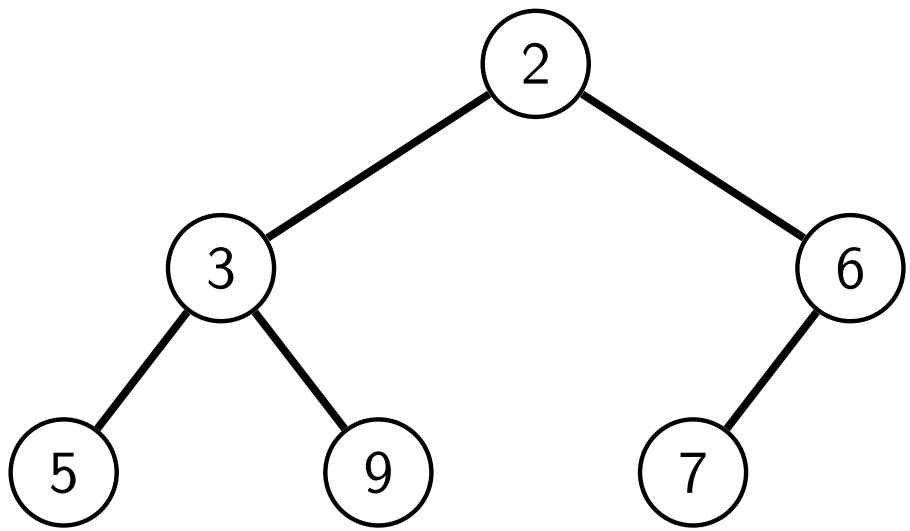
Anordnung im Speicher

- 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 $2 \cdot i + 2$



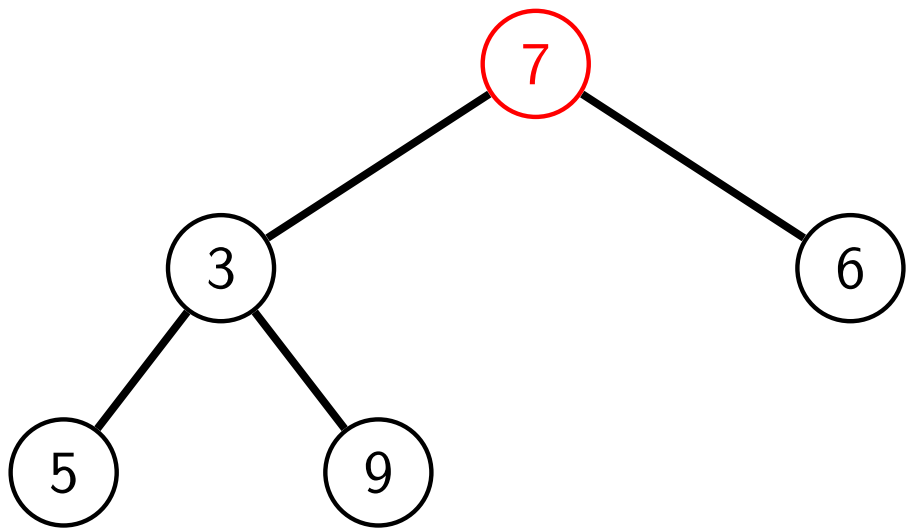
Operationen

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



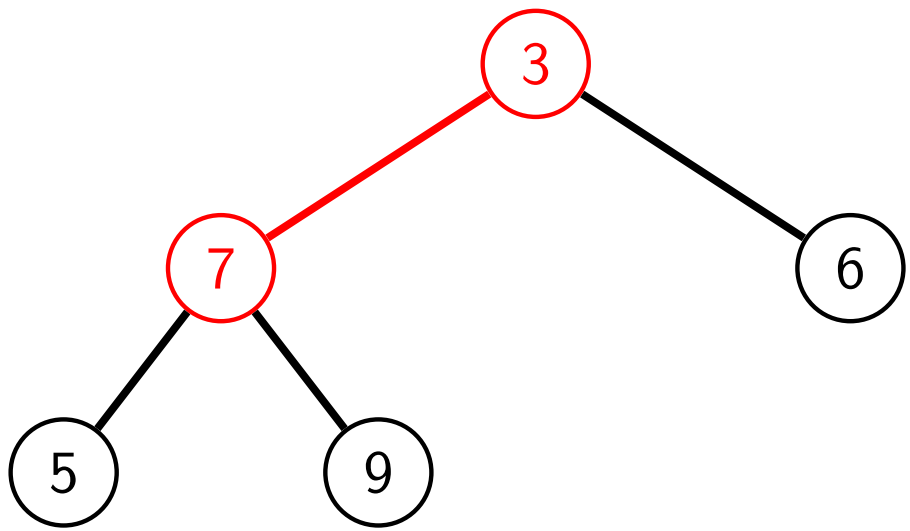
Operationen

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



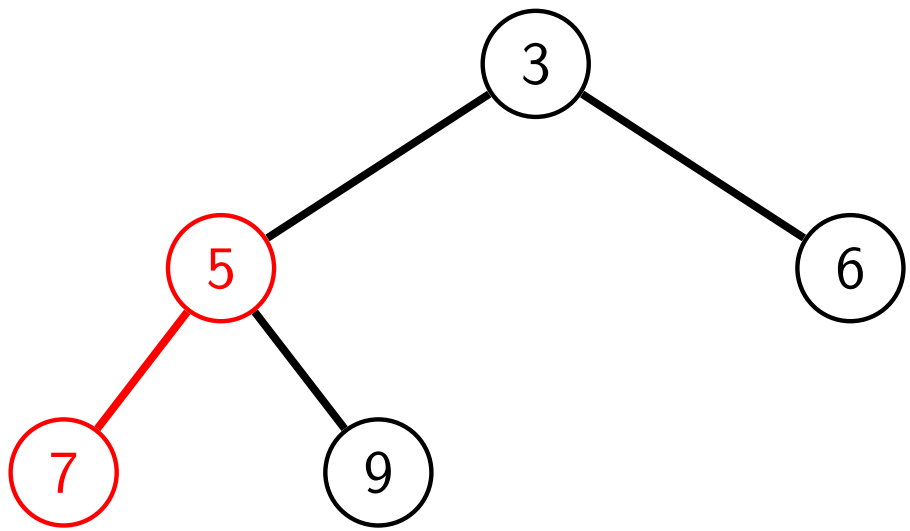
Operationen

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



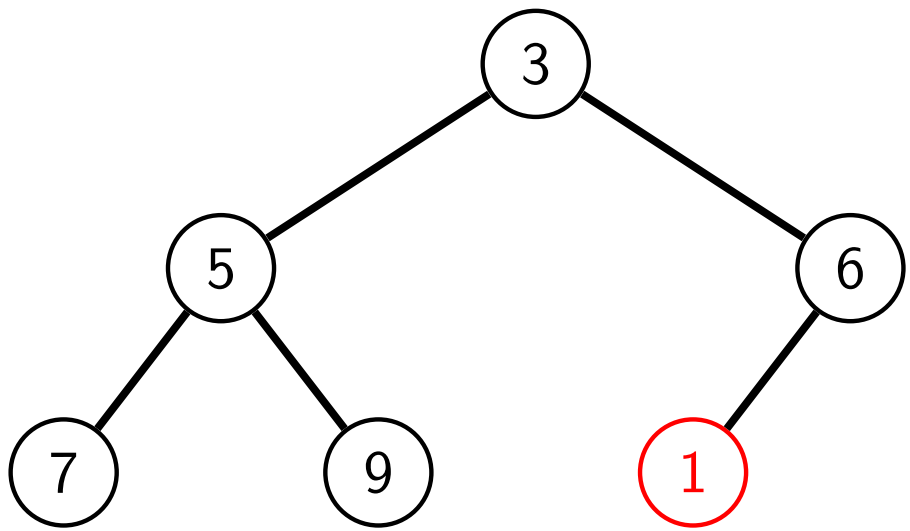
Operationen

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



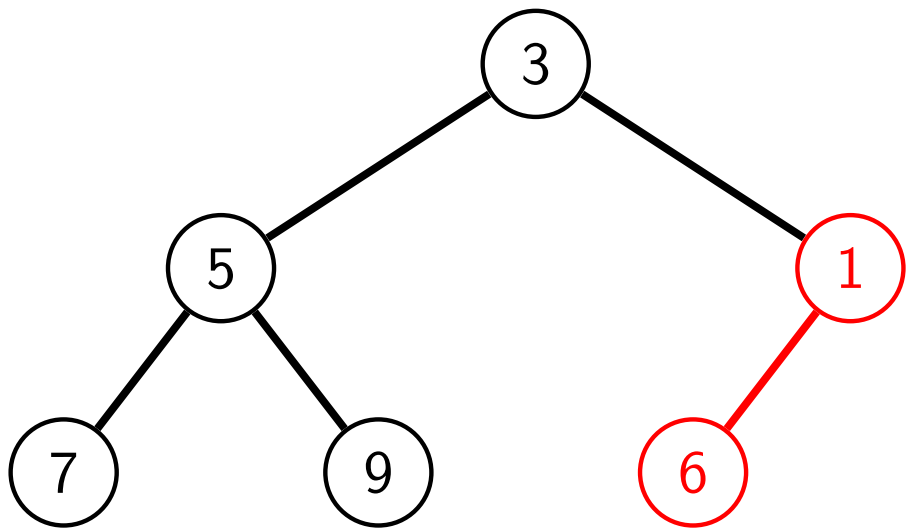
Operationen

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



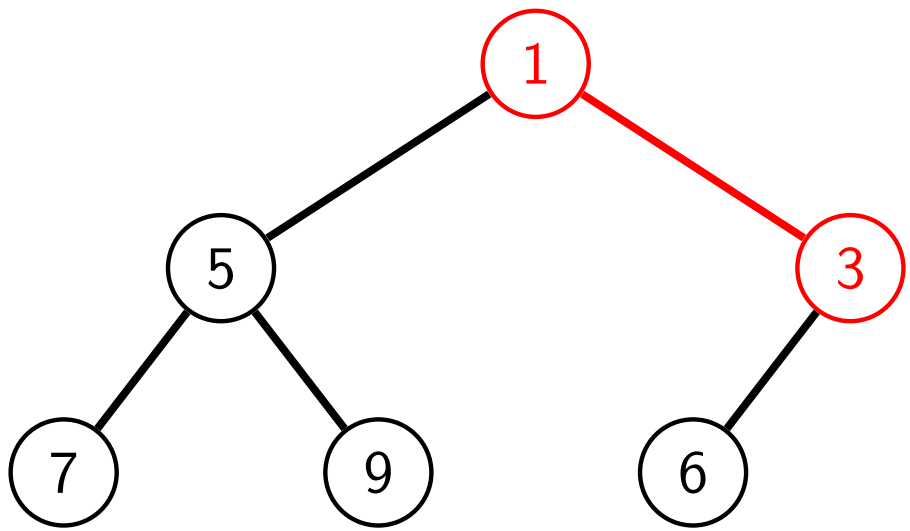
Operationen

- **minimum** ließt den minimalen Wert von der Wurzel
- **extractMin** entfernt die Wurzel (2), setzt das letzte Blatt an die Wurzel (7), und läßt große Wert „absinken“
- **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 Wert „absinken“
- **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
  - Siehe Vorlesungsfolien!


Code

Min-Priority-Queue

Warteschlange, aber...

- ...die Auslese-Reihenfolge wird nicht definiert durch die Schreibe-Reihenfolge (FI-FO), sondern...
- ...durch eine den Elementen zugeordnete *Priorität*.
- Besonders gut implementierbar durch Min-Heap!
- Sinnvoll etwa als Datenstruktur für Greedy-Ansätze...


Code

Binäre Suche

- Nutze Sortierung/Ordnung des Datensatzes zur schnelleren Suche
- Beispiel: Suche „München“ in (gedrucktem) Lexikon
  - Erstes Aufschlagen ergibt Artikel zur Stadt „Metz“, also weiter hinten
  - Wir finden „Nanga Parbat“, also wieder etwas weiter nach vorne
  - Es ergibt sich erst „Müller“, dann „München“
- Vergleiche Intervallschachtelung in der Mathematik...