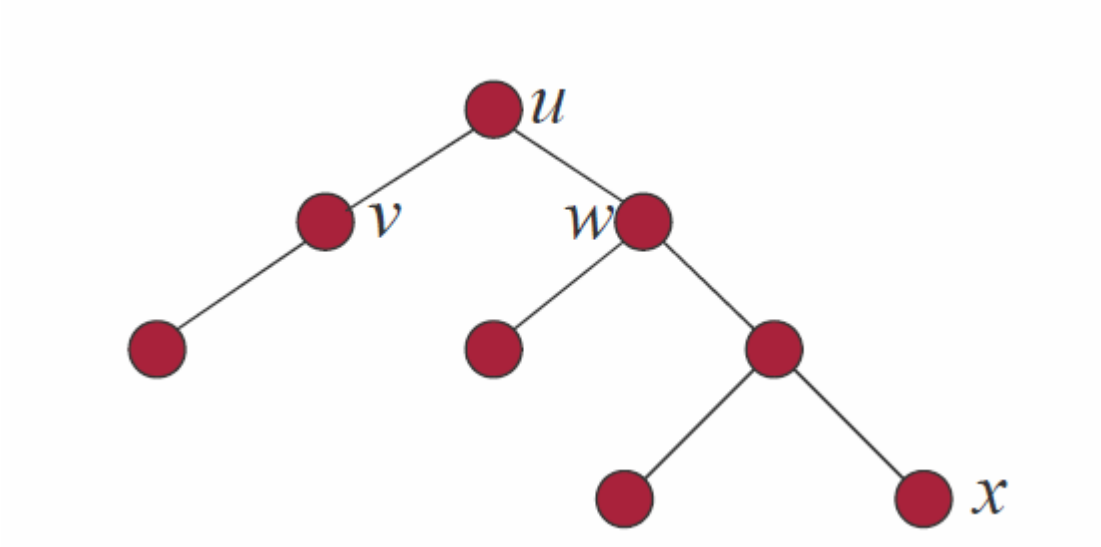


Sortieren mit Halde

Fachdidaktik Informatik
Referent: Hien Nguyen

Gliederung

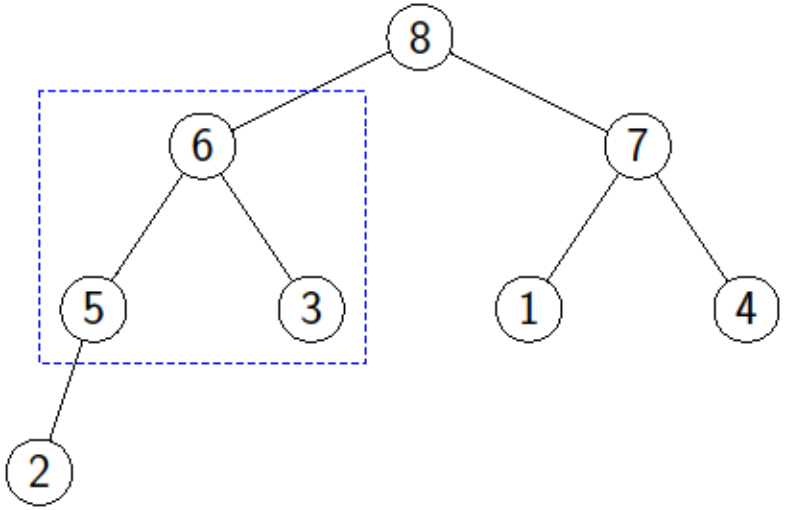
- Prinzip der Sortierung mit *Heapsort*
- Struktur eines Heaps und Eigenschaften
- Entwicklung eines Algorithmus
- Analyse der Laufzeit
- Beispiele

Prinzip der Sortierung mit *Heapsort*

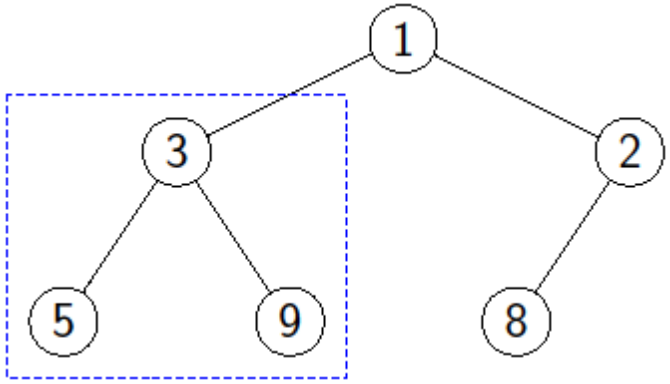
- Heapsort ist ein vergleichsbasiertes Sortierverfahren.
- Datenstruktur: Heap (Halde), die einen binären Baum mit bestimmten Eigenschaften bezeichnet:
  - Baum ist vollständig, d.h. jede Blattebene ist von links nach rechts gefüllt
  - Schlüssel eines Elternknoten ist kleiner oder gleich (größer oder gleich) dem Schlüssel seiner 2 Kinder
    - $\leq$: Min-Heap, d.h. $arr[parent(i)] \leq arr[i]$
    - $\geq$: Max-Heap, d.h. $arr[parent(i)] \geq arr[i]$
    - Wird als Heap-Eigenschaft bezeichnet

Beispiele

Max - Heap



Min - Heap



Anwendung und Eigenschaften

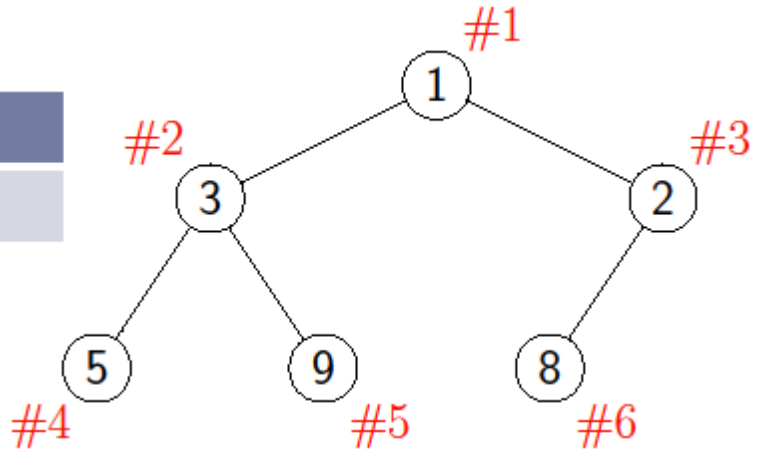
- Das kleinste bzw. größte Element befindet sich immer in der Wurzel
- Heap-Strukturen eignen sich, wenn jeweils nur das kleinste bzw. größte Element ausgewählt werden soll

- **Anwendung:**
  - Priority Queue (Warteschlange),
  - Sortierung Heapsort

Struktur und Eigenschaften

- Die Heap-Struktur kann lückenlos in der Implementierung leicht durch ein Array repräsentiert werden, weil ein binärer Heap ein vollständiger binärer Baum ist.
- Effiziente Tausch-Operationen für Elemente möglich

Index	1	2	3	4	5	6
Key	1	3	2	5	9	8



- Nummerierung der Knoten des Heaps in Level Order:
  - Kinder des k-ten Knotens auf Position
    - Links: $2k$
    - Rechts: $2k + 1$
  - Eltern-Knoten auf $\left\lfloor \frac{k}{2} \right\rfloor$

Eigenschaft auf Arrays

- Das Array $arr[1, \ldots, n]$ erfüllt die Heap-Eigenschaft, wenn:
  - $arr[k] \leq arr[2k]$
  - $arr[k] \leq arr[2k+1]$

Grundidee von Heapsort

- Beim Entfernen der Wurzel wird beim:
  - Max-Heap: größte Element
  - Min-Heap: kleinste Element

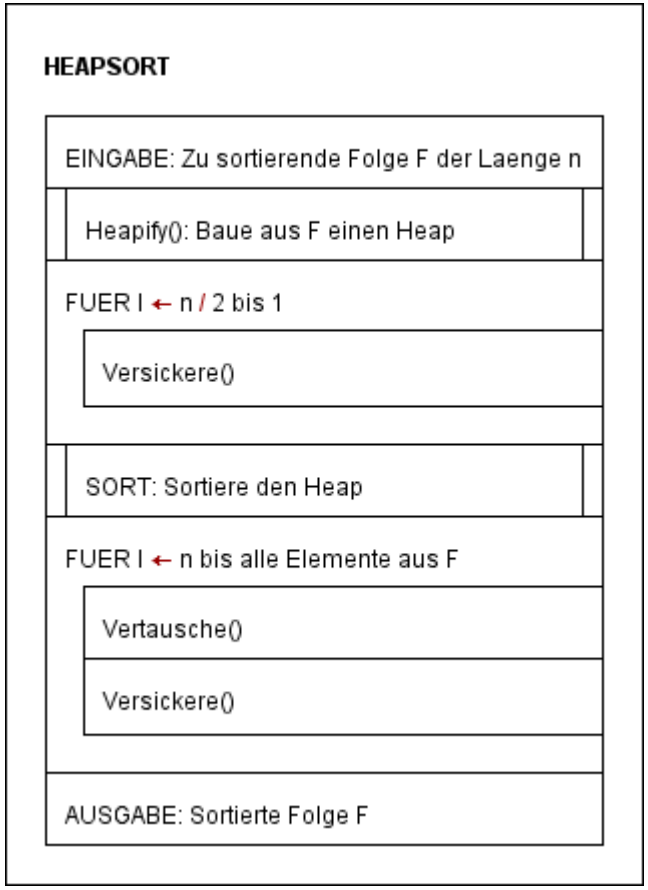
  aus dem Heap genommen und durch Versickern die Heap-Eigenschaft wiederhergestellt.

- Wiederholung auf Rest-Heap bis sortierte Reihenfolge vorliegt
- Ohne zusätzlichen Speicher

Entfernen der Wurzel

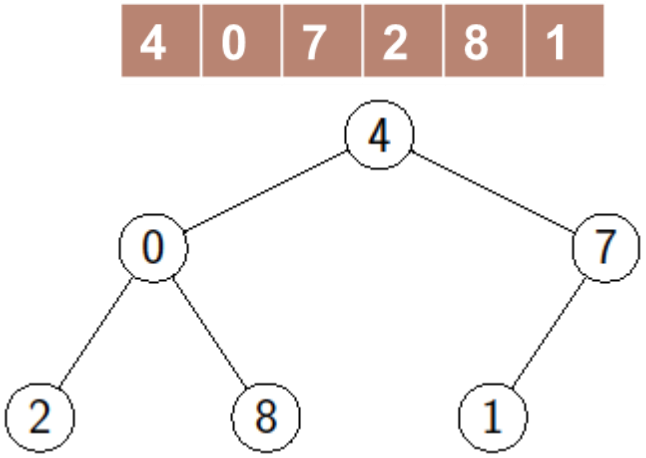
- **Problem:** Durch Entfernen der Wurzel entsteht eine „Lücke“ im Baum – die Heap-Eigenschaft wird verletzt
- **Lösung:**
  - **Bewege letzte Element des Heaps zur Wurzel**
  - **Vergleiche mit Nachfolgeelementen und „versickere (percolate)“ durch Tauschen der Nachfolger bis Heap-Eigenschaft wiederhergestellt ist**

Algorithmus und Implementierung

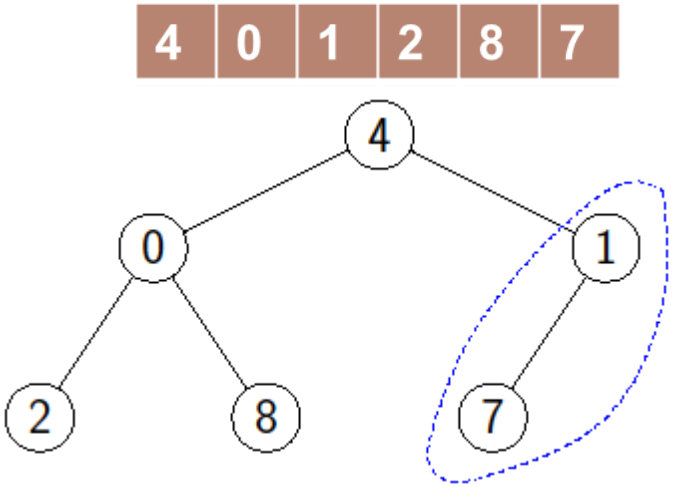


Phase 1: Aufbau des Heaps

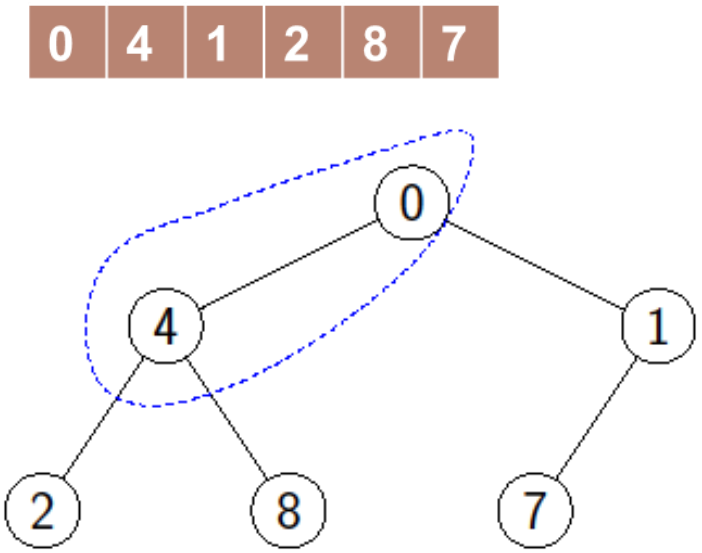
- Anwendung des Prinzips des Versickerns
- Starten beim letzten Element und versuchen dieses weiter im Baum nach unten zu bewegen
- Versickern immer in Teilbaum mit der kleineren Wurzel



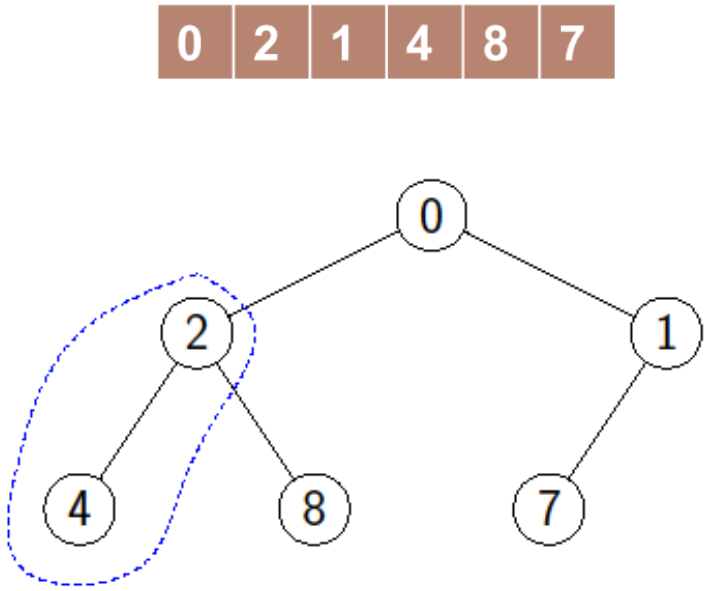
- Betrachtetes El: 7
- Heap-Eigenschaft verletzt



Heap nach Versickern von 7;
Naechstes betrachtetes El. 0



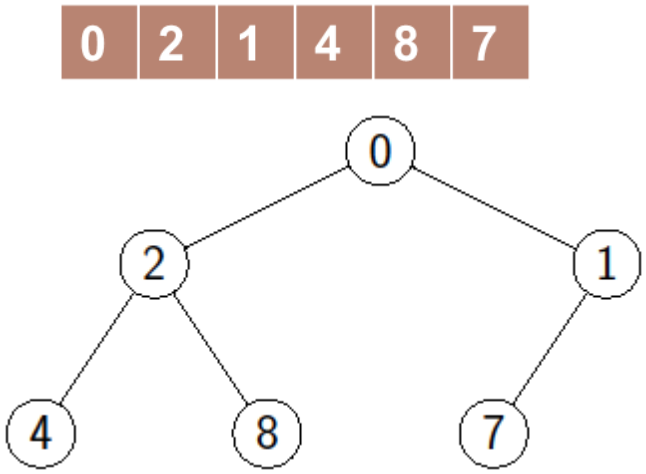
Heap nach versickern von El. 4, weil fuer El. 0 nichts zu tun war;
Heap-Eigenschaft immernoch verletzt

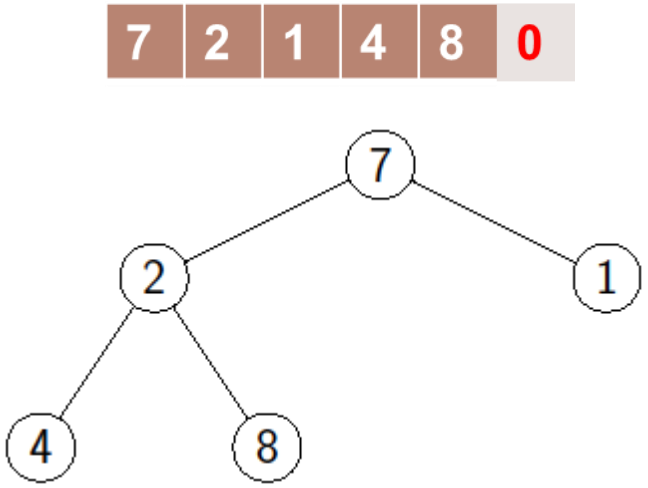


Heap nach weiter versickern von El. 4, weil Heap-Eigenschaft weiterhin verletzt war

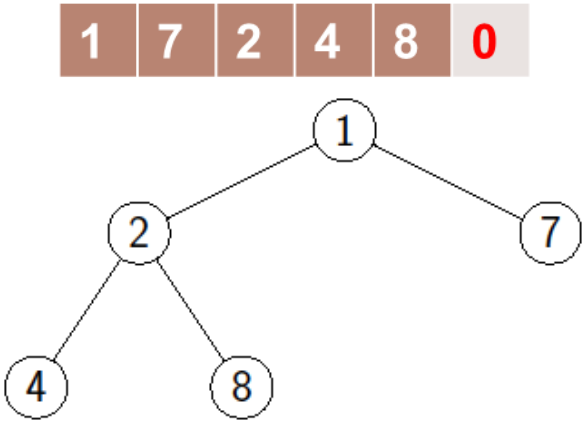
Phase 2: Sortierung des Heaps

- Anwendung der Grundidee:
  - Sortierung wird erreicht durch schrittweises Entfernen der Wurzel mit Wiederherstellung der Heap-Eigenschaft
  - Start vor Phase 2

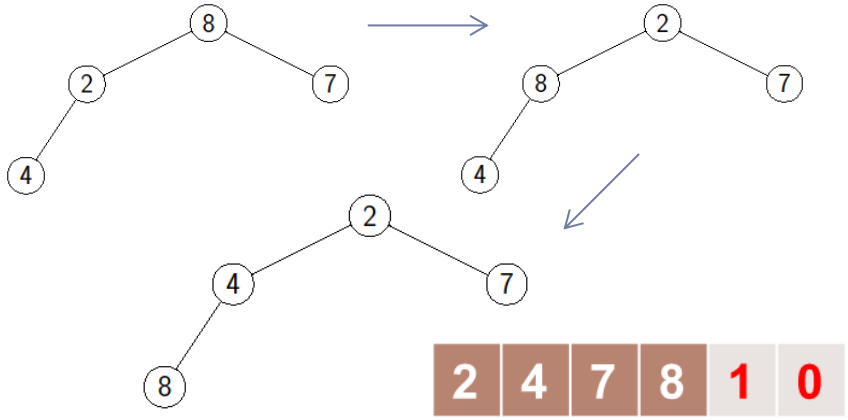




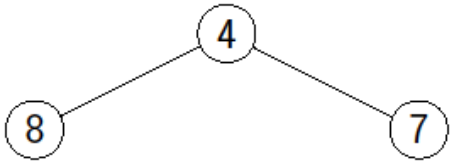
Löschen der Wurzel (0);
Ersetze durch letzte Element (7);
Füge (0) als letztes Element ein

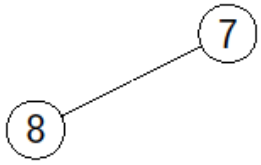


Heap-Eigenschaft wiederhergestellt



Entfernen von Wurzel (1);
Heap-Eigenschaft wieder herstellen





Fertig!

Implementierung

- Implementierung in Java
- Zwei Hilfsmethoden
  - Swap
    - Vertauscht zwei Elemente im Array
  - Percolate (Versickern)
    - Versickern eines Elements im Heap repräsentiert als Array

Analyse der Laufzeit

- 2 Bestandteile
  - generateHeap
    - Aufwand für das Durchsickern multipliziert mit Anzahl der Elemente n
    - Percolate: $\log n$
    - Gesamt: $\frac{n}{2} * \log n$
  - Sortierung
    - Aufwand fuer Entfernen
    - Aufwand fuer Heapify durch Durchsickern $\log n$
    - Gesamt: $n * \log n$
  - Gesamtlaufzeit: $O(n \log n)$