

## Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2017

Dr. Stefanie Demirci

Computer Aided Medical Procedures  
Technische Universität München



### 7 Fortgeschrittene Datenstrukturen

Graphen

Bäume

**Heaps**

Priority Queues

2

## Definition Heap

### Definition Heap

Sei  $G = (V, E)$  ein **Binärbaum** mit Wurzel  $w \in V$ . Jeder Knoten  $v \in V$  sei mit einem Wert  $key(v)$  verknüpft, die Werte seien durch  $\leq, \geq$  geordnet.

$G$  heißt **Heap**, falls er folgende zwei Eigenschaften erfüllt:

- $G$  ist **fast vollständig**, d.h. alle Ebenen sind vollständig gefüllt, ausser auf der untersten Ebene, die von links her nur bis zu einem bestimmten Punkt gefüllt sein muss.
- $G$  erfüllt die **Min-Heap-Eigenschaft** bzw. die **Max-Heap-Eigenschaft**, d.h. für alle Knoten  $v \in V, v \neq w$  gilt
  - Min-Heap:  $key(v.vater) \leq key(v)$
  - Max-Heap:  $key(v.vater) \geq key(v)$

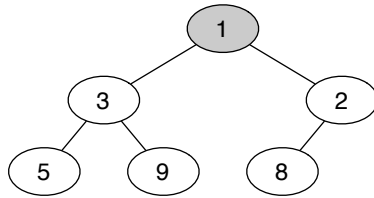
Entsprechend der Heap-Eigenschaft heißt  $G$  **Min-Heap** bzw. **Max-Heap**.

## Bemerkungen zur Heap Definition

Sei  $G = (V, E)$  Min-Heap.

- wir beschränken uns hier auf **Min-Heaps**, alle Aussagen gelten mit entsprechenden trivialen Änderungen auch für Max-Heaps
- typische **Keys** von Knoten sind Zahlen, z.B.  $key : V \rightarrow \mathbb{R}$ 
  - weiteres Beispiel: Strings als Keys, lexikographisch geordnet
- ein Heap ist **kein** abstrakter Datentyp!

## Min-Heap: Beispiel



- Keys sind hier natürliche Zahlen
- Baum ist fast vollständiger Binärbaum
- Baum erfüllt Min-Heap-Eigenschaft:
  - für alle Knoten  $v$  (ausser Wurzel) gilt

$$\text{key}(v.\text{vater}) \leq \text{key}(v)$$

5

## Heap Eigenschaften

Sei  $G = (V, E)$  Heap mit Wurzel  $w \in V$ .

- $G$  als Min-Heap bzw. Max-Heap hat immer Element mit kleinstem bzw. größtem Key in Wurzel  $w$
- $G$  ist aber nicht vollständig sortiert (d.h. Traversierung liefert nicht notwendigerweise vollständig sortierte Folge)
- Ist  $|V| = n$ , so hat  $G$  Höhe von  $\Theta(\log n)$
- typische Operation: extrahiere kleinsten (bzw. größten) Key (d.h. Wurzel), kurz: `extractMin` (bzw. `extractMax`)
  - anschließendes Problem: Heap-Eigenschaft wiederherstellen, als Operation: `minHeapify` (bzw. `maxHeapify`)

6

## Heap: extractMin

Sei  $G = (V, E)$  Min-Heap mit Wurzel  $w \in V$ .

- Operation `extractMin`:
  - entferne Wurzel  $w$  aus Heap  $G$  und liefere  $\text{key}(w)$  zurück
  - tausche letzten Knoten von  $G$  an Stelle von Wurzel
  - stelle Heap-Eigenschaft wieder her mit `minHeapify`

**Output:** minimaler Key in  $G$

`extractMin(G):`

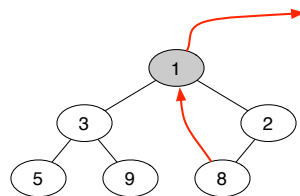
`min = key(w);`

`tausche Inhalt von  $w$  mit letztem`

`Knoten in  $G$ ;`

`minHeapify(G, w);`

`return min;`



7

## Heap: minHeapify

Sei  $G = (V, E)$  Min-Heap mit Wurzel  $w \in V$  und  $|V| = n$ .

- Operation `minHeapify` auf Knoten  $v \in V$  zur Wiederherstellung der Min-Heap-Eigenschaft
- **Voraussetzung:** nur Knoten  $v$  verletzt 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  $v$ ;`

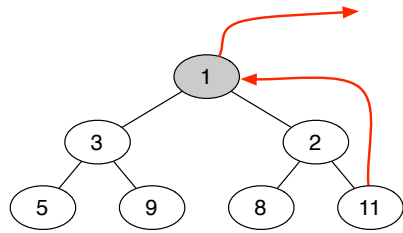
`minHeapify(G, knoten);`

}

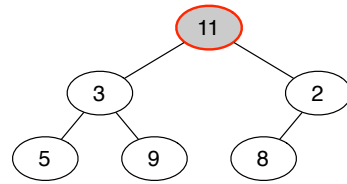
- Komplexität:  $O(\log n)$

8

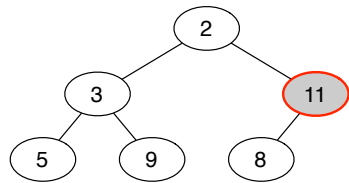
## Beispiel extractMin / minHeapify



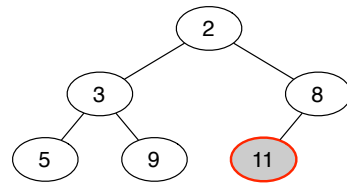
extractMin



minHeapify



minHeapify



minHeapify

9

## Heap erzeugen: buildMinHeap

Gegeben sei Knoten-Liste  $V$  mit  $|V| = n$  und Keys  $key(v)$  für  $v \in V$ .

- wie erzeugt man aus  $V$  einen Min-Heap  $G = (V, E)$ ?
  - erzeuge irgendwie 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( $V$ ):**

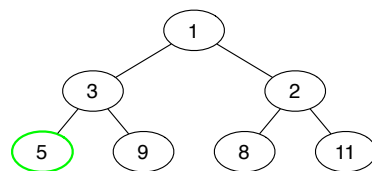
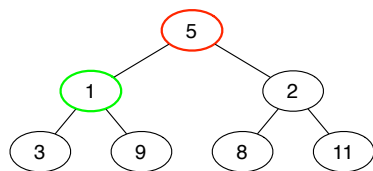
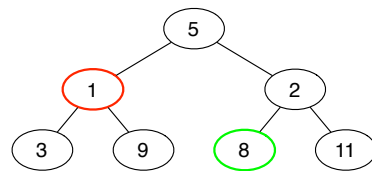
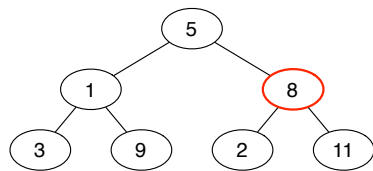
$G =$  erzeuge beliebigen fast vollständigen Binärbaum aus  $V$ ;

**for each** Knoten  $v$  in  $G$  von unten nach oben {  
     **minHeapify**( $G, v$ );  
}

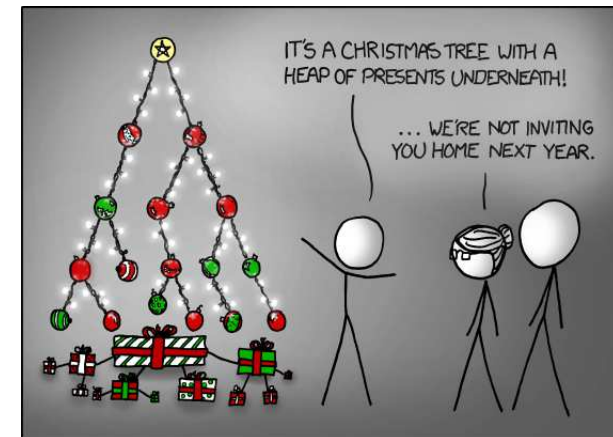
- Komplexität:  $O(n)$  (nicht nur  $O(n \log n)$ !)

10

## Beispiel buildMinHeap



11

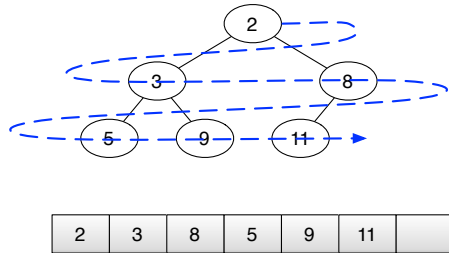


<http://xkcd.com/835/>

12

## Wiederholung: Binärbaum als sequentielle Liste I

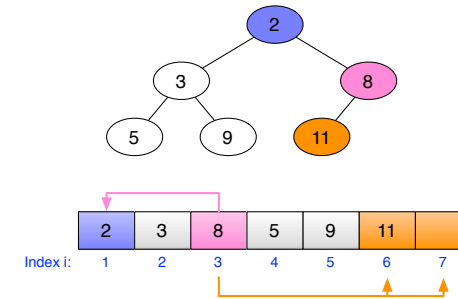
- vollständiger Binärbaum Höhe  $k$  hat  $2^{k+1} - 1$  Knoten
  - speichere Knoten von oben nach unten, von links nach rechts in sequentieller Liste (Array)
  - maximale Grösse von Array:  $2^{k+1} - 1$
- Beispiel fast vollständiger Binärbaum:



13

## Wiederholung: Binärbaum als sequentielle Liste II

- Wurzel: an Position 1
  - Knoten an Position  $i$ :
    - Vater-Knoten an Position  $\lfloor i/2 \rfloor$
    - linkes Kind an Position  $2i$
    - rechtes Kind an Position  $2i + 1$
- Pseudocode:
- **vater**( $i$ ): return  $\lfloor i/2 \rfloor$ ;
  - **links**( $i$ ): return  $2i$ ;
  - **rechts**( $i$ ): return  $2i + 1$ ;



14

## HeapSort

- Sortieren mit Heap
- Idee:
  - Heap erstellen mit **buildMinHeap**
  - wiederhole **extractMin** bis Heap leer
- mit Heap direkt im Eingabefeld:

**Input:** Feld  $A[1..n]$  der Länge  $n$

**HeapSort**( $A$ ):

**buildMinHeap**( $A$ );

**for**  $i=n$  **downto** 2 {

    tausche  $A[1]$  mit  $A[i]$ ;

$A.length = A.length - 1$ ;

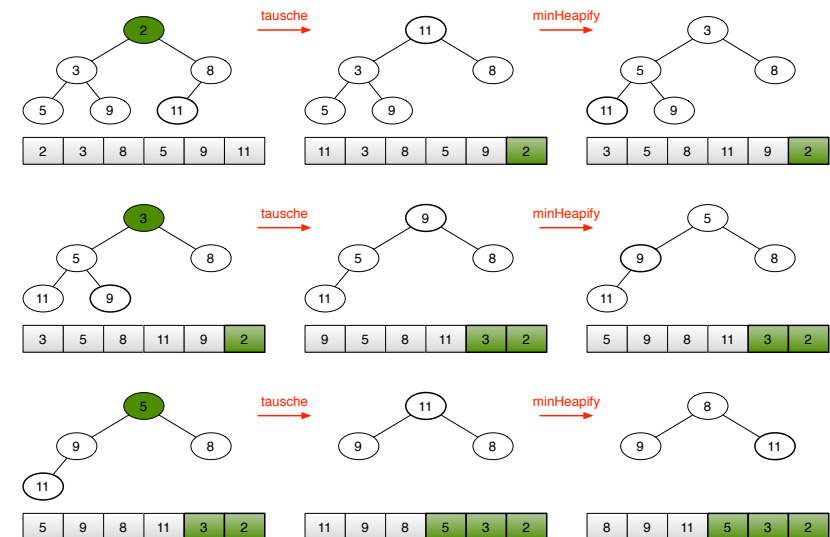
**minHeapify**( $A, 1$ );

  }

- **min-Heap** sortiert in **absteigender** Reihenfolge
- **max-Heap** sortiert in **aufsteigender** Reihenfolge

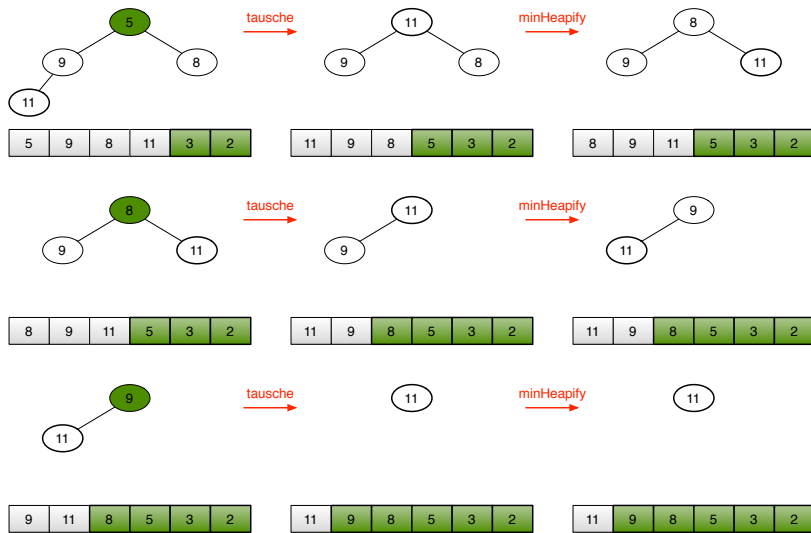
15

## HeapSort: Beispiel I



16

## HeapSort: Beispiel II



17

## HeapSort Eigenschaften

- sortiert in-place
- Komplexität  $O(n \log n)$ 
  - besser als QuickSort im worst case!
  - in Praxis aber erst bei grossem  $n$
- nicht stabil

18

## Stabilität von Sortierverfahren

### Stabilität

Ein Sortierverfahren heißt **stabil**, wenn es die Reihenfolge von gleichrangigen Elementen bewahrt.

Beispiel:

- unsortierte Liste:

5 2 3 4 3 (blau vor rot)

- sortierte Liste (stabil):


2 3 3 4 5 (blau vor rot)

- sortiere Liste (nicht stabil):

2 3 3 4 5 (rot vor blau)

19

## Sortier-Algorithmen illustriert

Animationen der Sortier-Algorithmen:  
 <http://www.sorting-algorithms.com>

20

## Sortier-Algorithmen Zusammenfassung

- Insertion Sort
  - in-place, stabil
  - Komplexität  $O(n^2)$ , best case:  $O(n)$
- Selection Sort (Übung)
  - in-place, nicht stabil
  - Komplexität  $O(n^2)$
- MergeSort
  - benötigt zusätzlichen Speicher, stabil
  - Komplexität  $O(n \log n)$
- QuickSort
  - in-place, nicht stabil
  - Komplexität im Mittel  $O(n \log n)$ , worst case:  $O(n^2)$
- HeapSort
  - in-place, nicht stabil
  - Komplexität  $O(n \log n)$

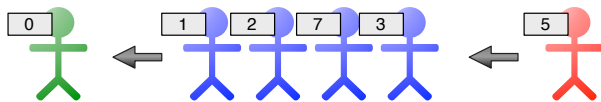
21

## Definition Priority Queue

### Definition Priority Queue

Eine Priority Queue ist ein **abstrakter Datentyp**. Sie beschreibt einen **Queue-artigen** Datentyp für eine Menge von Elementen mit zugeordnetem **Schlüssel** und unterstützt die Operationen

- Einfügen von Element mit Schlüssel in die Queue,
- Entfernen von Element mit **minimalem Schlüssel** aus der Queue,
- Ansehen des Elementes mit **minimalem Schlüssel** in der Queue.



- entsprechend gibt es auch eine Priority Queue mit Entfernen/Ansehen von Element mit **maximalem Schlüssel**

23

## Programm heute

### 7 Fortgeschrittene Datenstrukturen

Graphen  
Bäume  
Heaps  
Priority Queues

22

## Definition Priority Queue (abstrakter)

Priority Queue P ist ein abstrakter Datentyp mit Operationen

- $\text{insert}(P, x)$  wobei  $x$  ein Element
- $\text{extractMin}(P)$  liefert ein Element
- $\text{minimum}(P)$  liefert ein Element
- $\text{isEmpty}(P)$  liefert `true` or `false`
- $\text{initialize}$  liefert eine Priority Queue Instanz

und mit Bedingungen

- $\text{isEmpty}(\text{initialize}()) == \text{true}$
- $\text{isEmpty}(\text{insert}(P, x)) == \text{false}$
- $\text{minimum}(\text{initialize}())$  ist nicht erlaubt (Fehler)
- $\text{extractMin}(\text{initialize}())$  ist nicht erlaubt (Fehler)

(Fortsetzung nächste Folie)

24

## Definition Priority Queue (abstrakter)

Fortsetzung Bedingungen Priority Queue P:

- `minimum(insert(P, x))` liefert zurück
  - falls `P == initialize()`, dann `x`
  - sonst: `min(x, minimum(P))`
- `extractMin(insert(P, x))`
  - falls `x == minimum(insert(P, x))`, dann liefert es `x` zurück und hinterlässt `P` im Originalzustand
  - sonst liefert es `extractMin(P)` zurück und hinterlässt `P` im Zustand `insert(extractMin(P), x)`

(entsprechend für die Priority Queue mit maximalem Schlüssel)

## Priority Queue: Implementationen I

- mit sortierten Feldern (als sequentielle oder verkettete Liste)
  - `insert` legt Element an richtiger Stelle in sortierter Liste ab mit  $O(n)$  Komplexität
  - `minimum`, `extractMin` als  $O(1)$  Operation
- mit unsortierten Feldern (als sequentielle oder verkettete Liste)
  - `insert` hängt Element einfach an Ende an mit  $O(1)$
  - `minimum`, `extractMin` suchen nach Element mit kleinstem Schlüssel mit  $O(n)$

→ beides **nicht sonderlich effizient** (je nach Abfolge der Operationen aber ok)

25

26

## Priority Queue: Implementationen II

Priority Queue P als **min-Heap**  $G = (V, E)$  mit Wurzel  $w$ :

- `minimum` von P liefert **Wurzel  $w$**  zurück
  - Komplexität  $O(1)$
- `extractMin` von P entspricht `extractMin` von  $G$ 
  - Komplexität  $O(\log n)$
- `insert` von P erfordert ein klein wenig Extra-Aufwand:

**Input:** Priority Queue P (als min-Heap mit seq. Liste A), Element  $x$

**insert(A, x):**

füge Element  $x$  an Ende von Heap A ein;

$i$  = Index von letztem Element;

**while** ( ( $i \neq 1$ ) && ( $A[\text{vater}(i)] > A[i]$ ) ) {

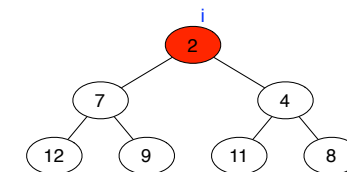
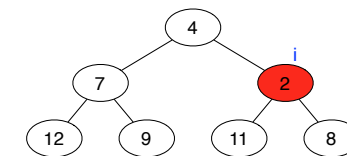
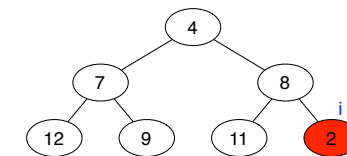
  tausche  $A[i]$  mit  $A[\text{vater}(i)]$ ;

$i = \text{vater}(i)$ ;

}

- Komplexität  $O(\log n)$

## Priority Queue: insert Beispiel



27

28

## Priority Queue: dynamisches Anpassen von Keys

- manchmal ändert sich der "Priorität" von Schlüsseln
  - Beispiel Algorithmen dafür in Kapitel 9!
- Operation `decreaseKey` verringert Schlüssel von bestimmten Element

**Input:** Priority Queue P (als min-Heap mit seq. Liste A),  
Element mit Index i, neuer Schlüsselwert wert

**decreaseKey**(A, i, wert):

**if** (wert > A[i]) **error** "neuer Schlüssel größer als alter!"

A[i] = wert;

**while** ( (i != 1) && (A[vater(i)] > A[i]) ) {

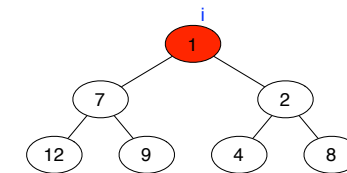
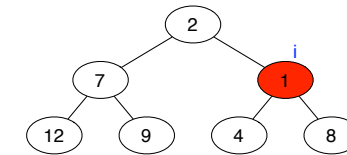
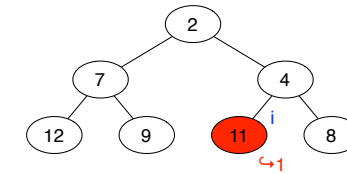
  tausche A[i] mit A[vater(i)];

  i = vater(i);

}

- Komplexität  $O(\log n)$

## Priority Queue: decreaseKey Beispiel



29

30

## Priority Queue: decreaseKey / insert

- mit Operation `decreaseKey` läßt sich `insert` anders formulieren:

**Input:** Priority Queue P (als min-Heap mit seq. Liste A),  
Element x

**insert**(A, x):

A.length = A.length + 1;

A[A.length] =  $\infty$ ;

**decreaseKey**(A, A.length, x);

## Priority Queue: Ausblick

- Priority Queue mit `Heap`: insert und decreaseKey sind  $O(\log n)$
- dies läßt sich mit `Fibonacci-Heap` bzw. `Radix-Heap` verbessern auf (amortisiert)  $O(1)$

→ Effiziente Algorithmen in Informatik

31

32



- mit **Priority Queues** lassen sich **Sortier-Algorithmen** implementieren
- **Schema:**
  - alle Elemente in Priority Queue **einfügen**
  - der Reihe nach alle Elemente mit **extractMin** / **extractMax** entfernen
- **Beispiele:**
  - Priority Queue mit Heap: **HeapSort**
  - Priority Queue mit sortierter sequentieller Liste: **Insertion Sort**
  - Priority Queue mit unsortierter sequentieller Liste: **Selection Sort**

- ⑦ **Fortgeschrittene Datenstrukturen**
  - Graphen
  - Bäume
  - Heaps
  - Priority Queues