

Algorithmen und Datenstrukturen (für ET/IT)

Sommersemester 2016

Dr. Tobias Lasser

Computer Aided Medical Procedures
Technische Universität München



Programm heute

7 Fortgeschrittene Datenstrukturen

Graphen

Bäume

Heaps

Priority Queues

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: $key(v.vater) \geq key(v)$

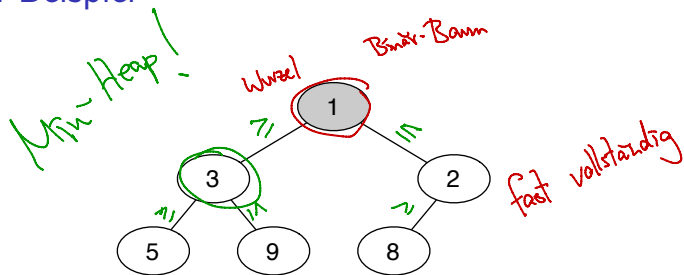
Entsprechend der Heap-Eigenschaft heißt G **Min-Heap** bzw. **Max-Heap**.

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

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

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 $\Theta(\log n)$
n Knoten *fast vollst. Binärbaum* 
- typische Operation: extrahiere kleinsten (bzw. größten) Key (d.h. Wurzel), kurz: **extractMin** (bzw. **extractMax**)
 - anschließendes Problem: Heap-Eigenschaft wiederherstellen, als Operation: **minHeapify** (bzw. **maxHeapify**)

Heap: extractMin

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

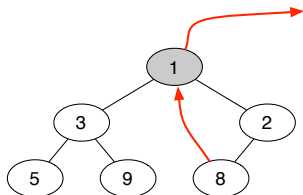
- Operation **extractMin**:
 - entferne Wurzel w aus Heap G und liefere $key(w)$ zurück
 - tausche letzten Knoten von G an Stelle von Wurzel
 - stelle Heap-Eigenschaft wieder her mit **minHeapify**

Output: minimaler Key in G

extractMin(G):

- $O(1)$ $min = key(w)$;
- $O(1)$ tausche Inhalt von w mit letztem Knoten in G ;
- ? **minHeapify**(G, w);
- $O(1)$ **return** min ;

$\Theta(\log n)$



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

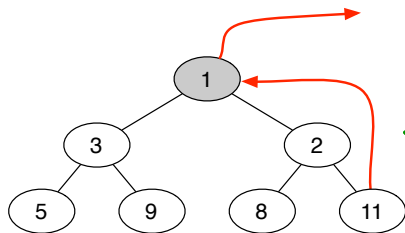
Input: Knoten v

minHeapify(G, v):

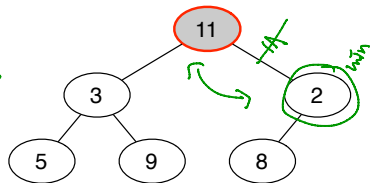
$O(1)$ if (v ist Blatt) **return**;
 $O(1)$ $knoten =$ Minimum (bzgl. key) von $v.links$ und $v.rechts$;
 $O(1)$ if ($key(knoten) < key(v)$) {
 $O(1)$ tausche Inhalt von $knoten$ und v ;
 minHeapify($G, knoten$);
}

- Komplexität: $\Theta(\log n)$

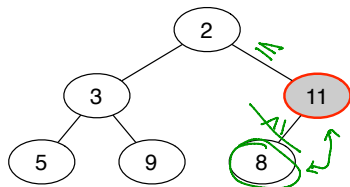
Beispiel extractMin / minHeapify $\rightarrow \Theta(\log n)$



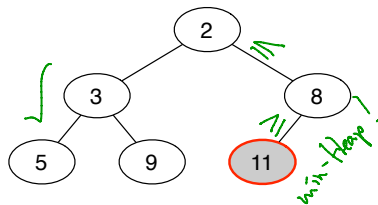
extractMin



minHeapify



minHeapify



minHeapify

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

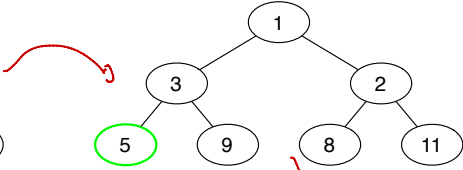
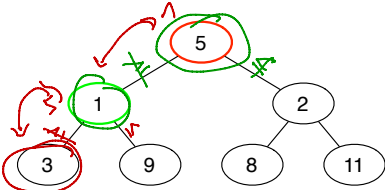
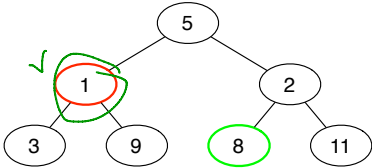
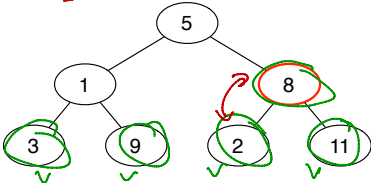
buildMinHeap(V):

$O(n)$ $G =$ erzeuge beliebigen fast vollständigen Binärbaum aus V ;
 n mal **for each** Knoten v in G von unten nach oben {
 $O(\log n)$ **minHeapify**(G, v);
}

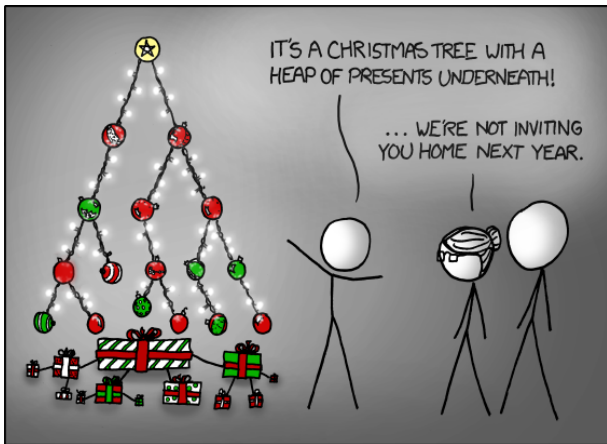
- Komplexität: $O(n)$ (nicht nur $O(n \log n)$!)

Beispiel buildMinHeap

Zusammengeknüpfelter
fast vollst. Binärbaum



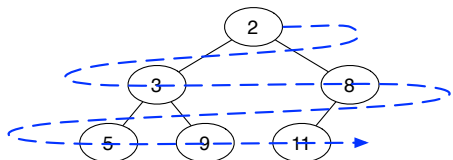
min-Heap!



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

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:



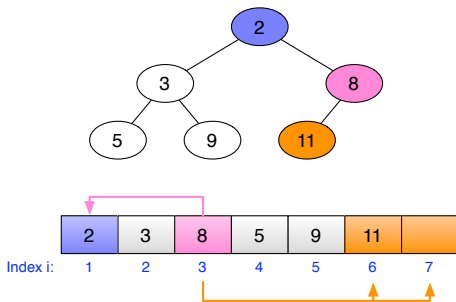
2	3	8	5	9	11	
---	---	---	---	---	----	--

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 $2i + 1$;



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):

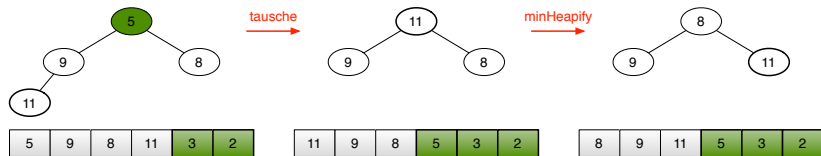
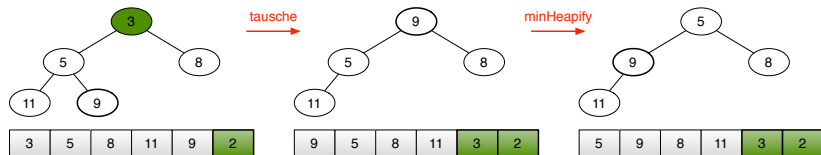
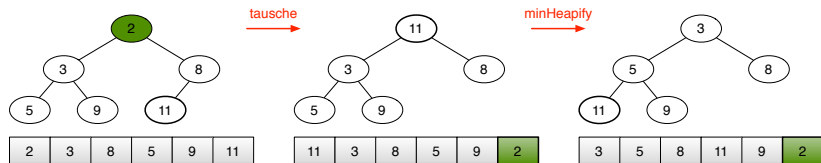
```
 $O(n)$  buildMinHeap(A);  
 $O(n)$  mal for  $i=n$  downto 2 {  
   $O(1)$  tausche  $A[1]$  mit  $A[i]$ ;  
   $O(1)$   $A.length = A.length - 1$ ;  
   $O(\log n)$  minHeapify(A, 1);  
}
```

Wurzel == Minimum!

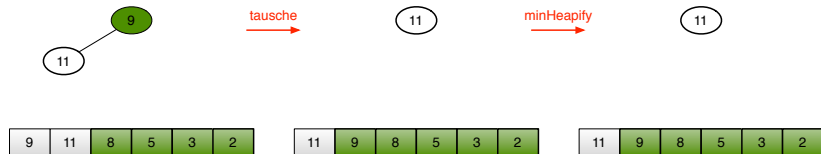
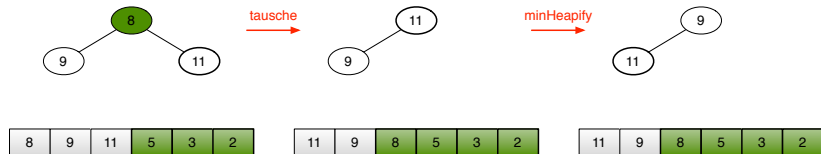
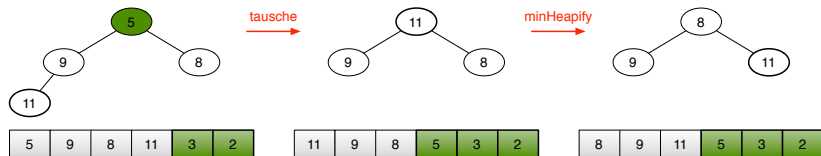
$O(n \log n)$

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

HeapSort: Beispiel 1



HeapSort: Beispiel II



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

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)

Sortier-Algorithmen illustriert

Animationen der Sortier-Algorithmen:



<http://www.sorting-algorithms.com>

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 $O(n \log n)$

Programm heute

7 Fortgeschrittene Datenstrukturen

Graphen

Bäume

Heaps

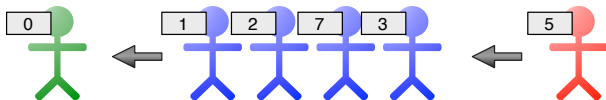
Priority Queues

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**

Definition Priority Queue (abstrakter)

Priority Queue P ist ein abstrakter Datentyp mit Operationen

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

und mit Bedingungen

- `isEmpty(initialize()) == true`
- `isEmpty(insert(P, x)) == false`
- `minimum(initialize())` ist nicht erlaubt (Fehler)
- `extractMin(initialize())` ist nicht erlaubt (Fehler)

(Fortsetzung nächste Folie)

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

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 $O(n)$

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)

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:

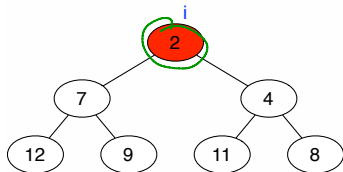
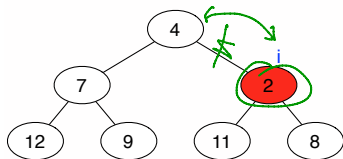
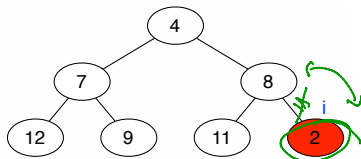
Input: Priority Queue P (als min-Heap mit seq. Liste A),
Element x

insert(A, x):

$O(n)$ füge Element x an Ende von Heap A ein;
 $O(1)$ $i =$ Index von letztem Element;
 $O(\log n)$ \rightarrow **while** (($i \neq 1$) && ($A[\mathbf{vater}(i)] > A[i]$)) {
 $O(1)$ tausche $A[i]$ mit $A[\mathbf{vater}(i)]$;
 $O(1)$ $i = \mathbf{vater}(i)$;
}

- Komplexität $O(\log n)$

Priority Queue: insert Beispiel



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):

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

O(n) A[i] = wert;

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

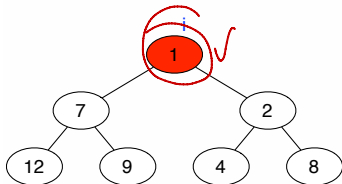
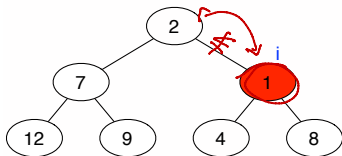
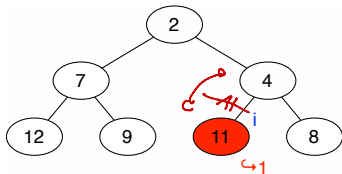
O(n) tausche A[i] mit A[vater(i)];

O(n) i = vater(i);

}

- Komplexität $O(\log n)$

Priority Queue: decreaseKey Beispiel



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] = ∞ ;

decreaseKey(A, A.length, x);

immer noch $O(\log n)$ Komplexität

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

extract Min



$O(\log n)$

Priority Queues und Sortieren

- 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**

Zusammenfassung

7 Fortgeschrittene Datenstrukturen

- Graphen

- Bäume

- Heaps

- Priority Queues