Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 9a, Dienstag, 18. Juni 2013

(Prioritätswarteschlangen, Binärer Heap)

Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü8 (Suchbäume)
- Prioritätswarteschlangen / Priority Queues
  - Noch eine grundlegende Datenstruktur
  - Anwendungsbeispiele
  - Realisierung über einen **binären Heap** / **binary heap**
  - Beginn einer Implementierung (insert + repairHeapUpwards)
  - Ü9, Aufgabe 1: Implementierung der übrigen Methoden (getMin, repairHeapDownwards, deleteMin, changeKey, size)

Erfahrungen mit dem Ü8 (Suchbäume)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 18. Juni 16:00
  - Auch dieses Blatt wieder gut machbar
  - Eine Aufgabe Programm, eine Aufgabe Mathe ist gut
  - Was gelernt ohne zu Verzweifeln
  - Blätter jetzt wieder einfacher, dafür langweiliger
  - Rückgabewert von lookup im Code-Vorschlag war irritierend

    Für lookup braucht man nur den Key, aber für die Benutzung innerhalb von insert braucht man das Blatt, bis zu dem gesucht wurde
  - Potenzialfunktion hier eingängiger als bei dynamischen Feldern
  - Ü nicht lösbar: Wetter zu gut + wohne direkt am See (x 10)

Prioritätswarteschlangen 1/5

UNI FREIBURG

- Definition
  - Eine **Prioritätswarteschlange** (PW) speichert wieder eine Menge von Key-Value Paaren / Elementen

    wie auch schon bei HashMap und BinarySearchTree
  - Es gibt wieder eine totale Ordnung $\leq$ auf den Keys
  - Die PW unterstützt auf dieser Menge folgende Operationen
    - insert(key, value): füge das gegebene Element ein
    - getMin(): liefert das Element mit dem kleinsten Key
    - deleteMin(): entferne das Element mit dem kleinsten Key
  - Und manchmal auch noch
    - changeKey(item): ändere Key des gegebenen Elementes
    - remove(item): entferne das gegebene Element

Prioritätswarteschlangen 2/5

UNI FREIBURG

- Vergleich mit HashMap und BinarySearchTree
  - Bei der HashMap sind die Keys in keiner besonderen Ordnung abgespeichert

    Von daher würden uns getMin und deleteMin dort $\Theta(n)$ Zeit kosten, n = Anzahl Schlüsssel
  - Der BinarySearchTree kann alles was eine PriorityQueue kann und **mehr** (nämlich beliebiges lookup)

    Wir werden sehen, dass dafür die PriorityQueue, für das was sie kann und macht, effizienter ist

    Und tatsächlich gibt es viele Anwendungen, wo eine PriorityQueue ausreicht ... siehe spätere Folien

Prioritätswarteschlangen 3/5

UNI FREIBURG

- Mehrere Elemente mit dem gleichen Key
  - Kein Problem, und für viele Anwendungen nötig
  - Falls es mehrere Elemente mit dem kleinsten Key gibt:
    - gibt getMin irgend eines davon zurück
    - und deleteMin löscht eben dieses
- Argument der Operationen changeKey und remove
  - Eine PW erlaubt **keinen** Zugriff auf ein beliebiges Element
  - Deshalb geben (bei unserer Implementierung) insert und getMin eine Referenz auf das entsprechende Element zurück
  - Mit dieser Referenz kann man dann später über changeKey bzw. remove den Schlüssel ändern / das Element entfernen

Prioritätswarteschlangen 4/5

UNI FREIBURG

- Benutzung in **Java**
  - Im Vorspann: import java.util.PriorityQueue;
  - Element-Typ unterscheidet nicht zwischen Key und Value

    PriorityQueue<T> pq;
  - Defaultmäßig wird die Ordnung ≤ auf T genommen
    - eigene Ordnung über einen Comparator, wie bei sort
    - siehe unseren Code zum Sortieren mit einer PW
  - Operationen: insert = add, getMin = peek, deleteMin = poll
  - Die Operation changeKey gibt es nicht
  - Dafür gibt es remove = entferne ein gegebenes Element
  - Mit remove und insert kann man ein changeKey simulieren !

Prioritätswarteschlangen 5/5

UNI FREIBURG

- Benutzung in **C++**
  - Im Vorspann: `#include <queue>;`
  - Element-Typ unterscheidet nicht zwischen Key und Value

    `std::priority_queue<T> pq;`
  - Es wird die Ordnung $\geq$ auf `T` genommen, und nicht $\leq$
  - Beliebige Vergleichsfunktion wie bei `std::sort`
  - Operationen: `insert` = `push`, `getMin` = `top`, `deleteMin` = `pop`
  - Es gibt kein `changeKey` und auch kein beliebiges `remove`

    (aus Effizienzgründen: es macht die Implementierung komplexer, aber viele Anwendungen brauchen es nicht)

PWs — Anwendungen 1/3

UNI FREIBURG

- Anwendungsbeispiel 1
  - Man kann mit einer PW einfach **sortieren**:

    Seien die Elemente x_1, x_2, …, x_n

    Erst alle einfügen: insert(x_1), insert(x_2), …, insert(x_n)

    Dann wieder rausholen, immer das kleinste was noch da ist: getMin(), deleteMin(), getMin(), deleteMin(), …

    Der entsprechende Algorithmus heißt **HeapSort**
  - Wir sehen später: alle Operationen gehen in O(log n) Zeit

    Damit läuft HeapSort in O(n · log n) Zeit

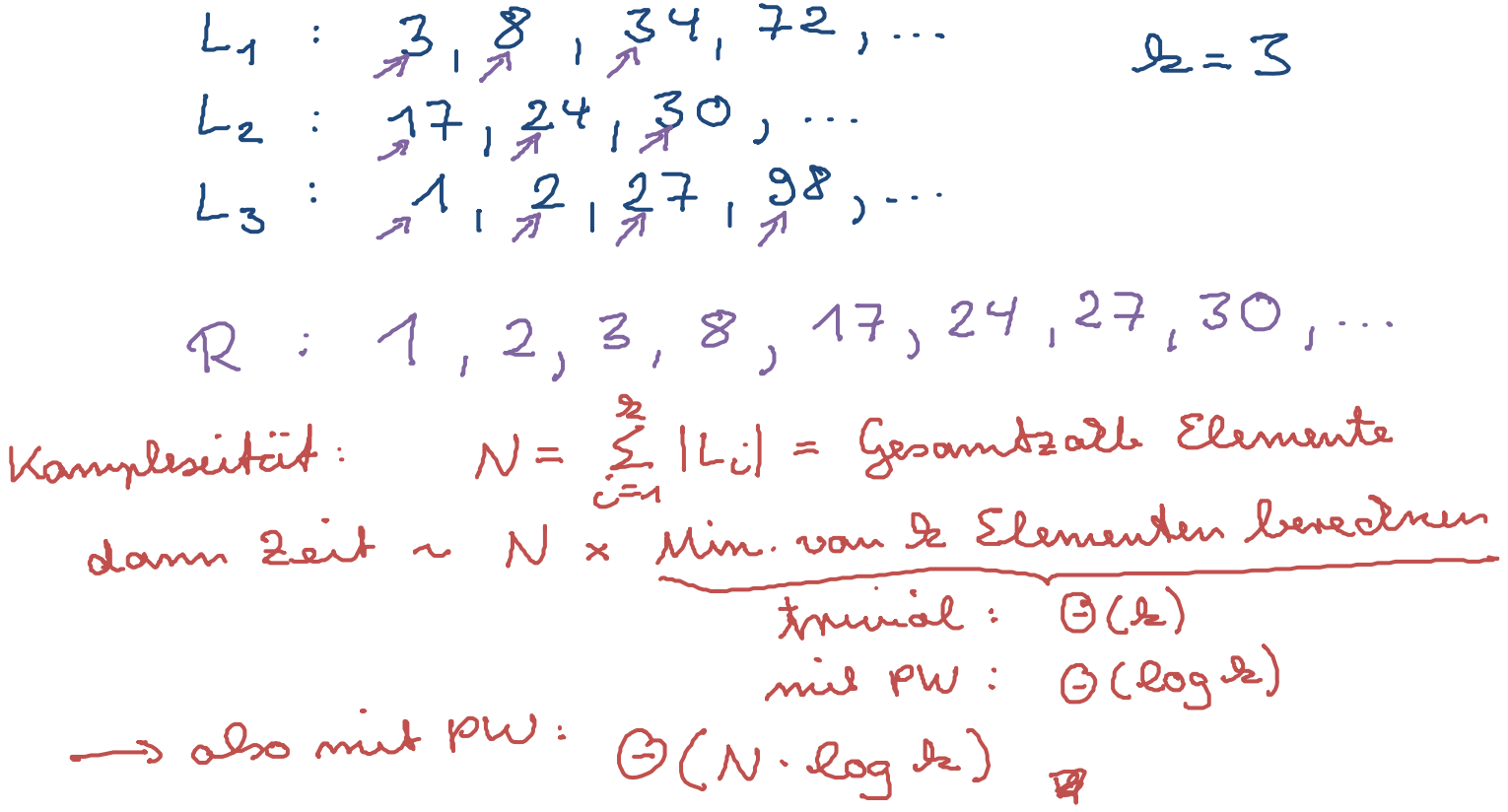
    Also asymptotisch **optimal** für vergleichsbasiertes Sortieren

    Insbesondere genauso gut wie MergeSort (im allgemeinen Fall) und QuickSort (im besten Fall)

PWs — Anwendungen 2/3

UNI FREIBURG

- Anwendungsbeispiel 2
  - Berechnung der Vereinigungsmenge von k sortierten Listen (sogenannter multi-way merge oder k-way merge)



PWs — Anwendungen 3/3

UNI FREIBURG

- Anwendungsbeispiel 3
  - Die PW ist die grundlegende Datenstruktur bei **Dijkstra's Algorithmus** zur Berechnung kürzester Wege

    Das machen wir nächste Woche !

Implementierung 1/7

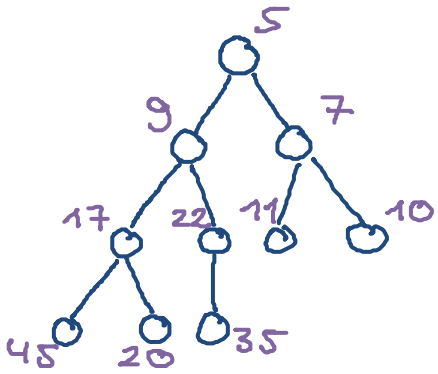
UNI FREIBURG

- Grundidee
  - Wir speichern die Elemente in einem **binären Heap**

    Das ist ein vollständiger binärer Baum

    Es gilt die Heap-Eigenschaft = der Key jedes Knotens ist ≤ die Keys von den beiden Kindern

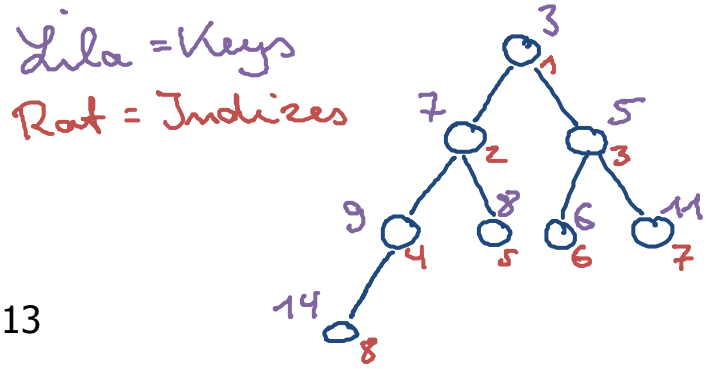
    Wichtig: das ist eine **schwächere** Eigenschaft als beim BinarySearchTree, insbesondere Blätter nicht sortiert

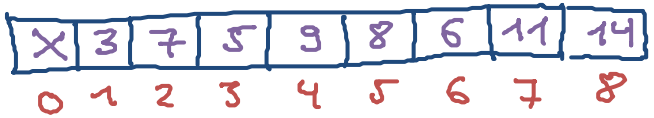


Implementierung 2/7

- Wie speichert man einen binären Heap
  - Anders als beim BinarySearchTree geht das **ohne Zeiger**
  - Wir numerieren die Knoten von oben nach unten und links nach rechts durch, beginnend mit **1**
  - Dann sind die Kinder von Knoten i die Knoten 2i und 2i + 1
  - Und der Elternknoten von einem Knoten i ist floor(i/2)
  - Die Elemente stehen dann einfach in einem Array:

```
ArrayList<PriorityQueueItem> heap;      // Java.
std::vector<PriorityQueueItem> heap;    // C++.
```





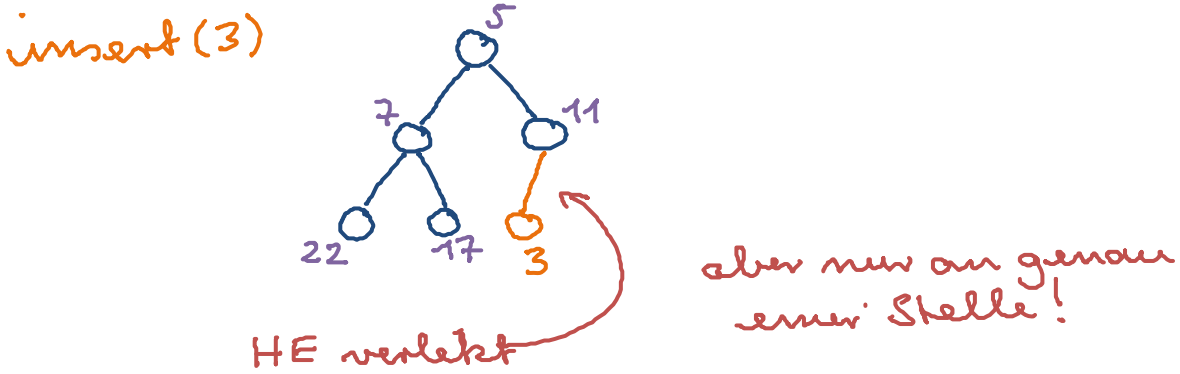
Implementierung 3/7

- Einfügen eines Elementes (insert)
  - Erstmal hinzufügen am Ende des Arrays

    heap.add(keyValuePair); // Java.

    heap.push_back(keyValuePair); // C++.
  - Danach kann die Heapeigenschaft verletzt sein

    ... aber nur genau an dieser (letzten) Position !
  - Wiederherstellung der Heapeigenschaft → spätere Folie



Implementierung 4/7

UNI FREIBURG

- Rückgabe des Elem. mit kleinstem Key (getMin)
  - Einfach das oberste Element zurückgeben

    ```
    return heap.get(1);   // Java.
    return heap[1];   // C++.
    ```

  - Falls Heap leer, einfach null zurückgeben

Implementierung 5/7

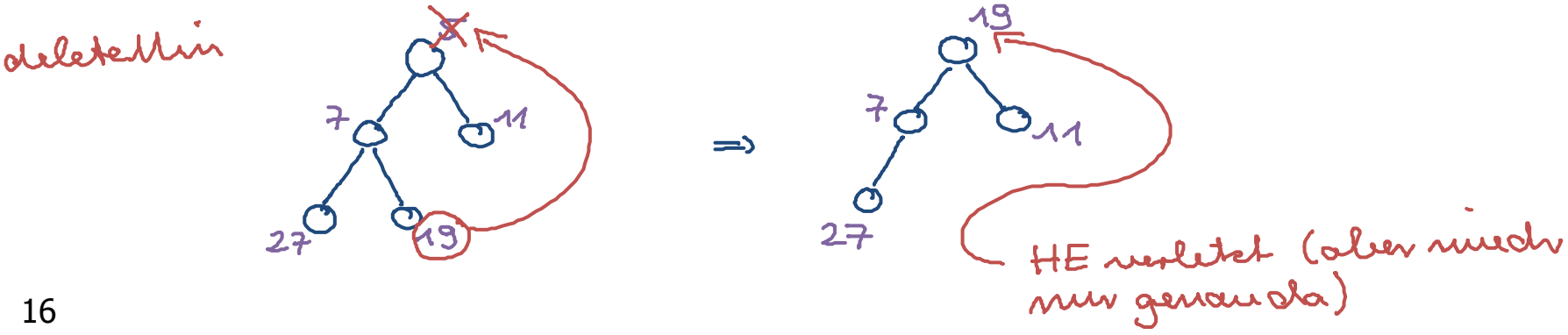
UNI FREIBURG

- Löschen des Elem. mit kleinstem Key (deleteMin)
  - Einfach das Element von der letzten Position an die erste Stelle setzen (falls heap nicht leer)

    heap.get(1) = heap.remove(heap.size() - 1); // Java.

    heap[1] = heap.back(); heap.pop_back(); // C++.
  - Danach kann die Heapeigenschaft verletzt sein

    ... aber wieder nur genau an dieser (ersten) Position !
  - Wiederherstellung der Heapeigenschaft → spätere Folie

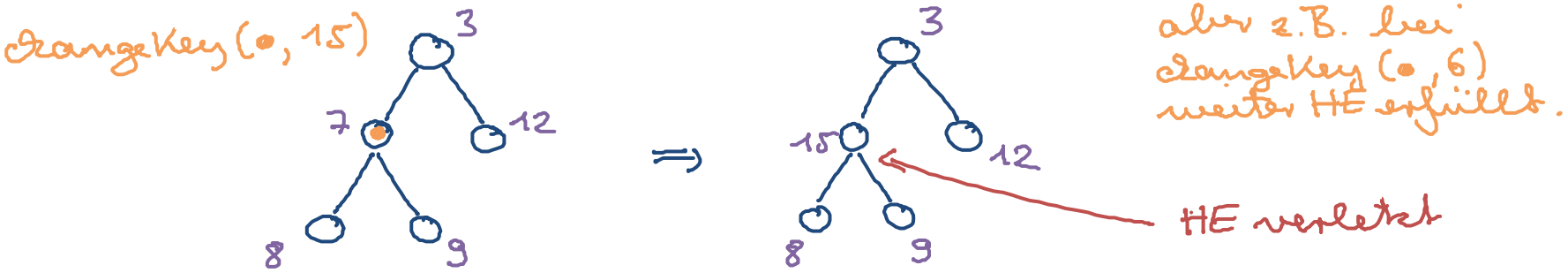


Implementierung 6/7

- Ändern eines Schlüssels (changeKey)
  - Element (pqItem) wurde als Argument übergeben !
  - Dann einfach den Schlüssel ändern

    pqItem.key = newKey;
  - Danach kann die Heapeigenschaft verletzt sein

    ... aber wieder nur genau an dieser Position !
  - Wiederherstellung der Heapeigenschaft → spätere Folie
  - Jedes pqItem muss also seine Position kennen → spätere Folie



Implementierung 7/7

UNI FREIBURG

- Entfernen eines Elementes (remove)
  - Element (pqItem) wurde als Argument übergeben !
  - Dann einfach das Element von der letzten Position an diese Stelle setzen
  - Danach kann die Heapeigenschaft verletzt sein

    ... aber wieder nur genau an dieser Position !
  - Wiederherstellung der Heapeigenschaft → spätere Folie
  - Jedes pqItem muss also seine Position kennen → spätere Folie

Reparieren der Heapeigenschaft 1/4

UNI FREIBURG

- Nach insert, deleteMin, changeKey, remove
  - ... kann die Heapeigenschaft (HE) verletzt sein
  - Aber nur an genau einer (bekannten) Position i
  - Die HE kann auf zwei Arten verletzt sein:
    - Schlüssel an Position i ist nicht ≤ der seiner Kinder
    - Schlüssel an Position i ist nicht ≥ der vom Elternkn.
  - Entsprechend brauchen wir zwei Reperaturmethoden

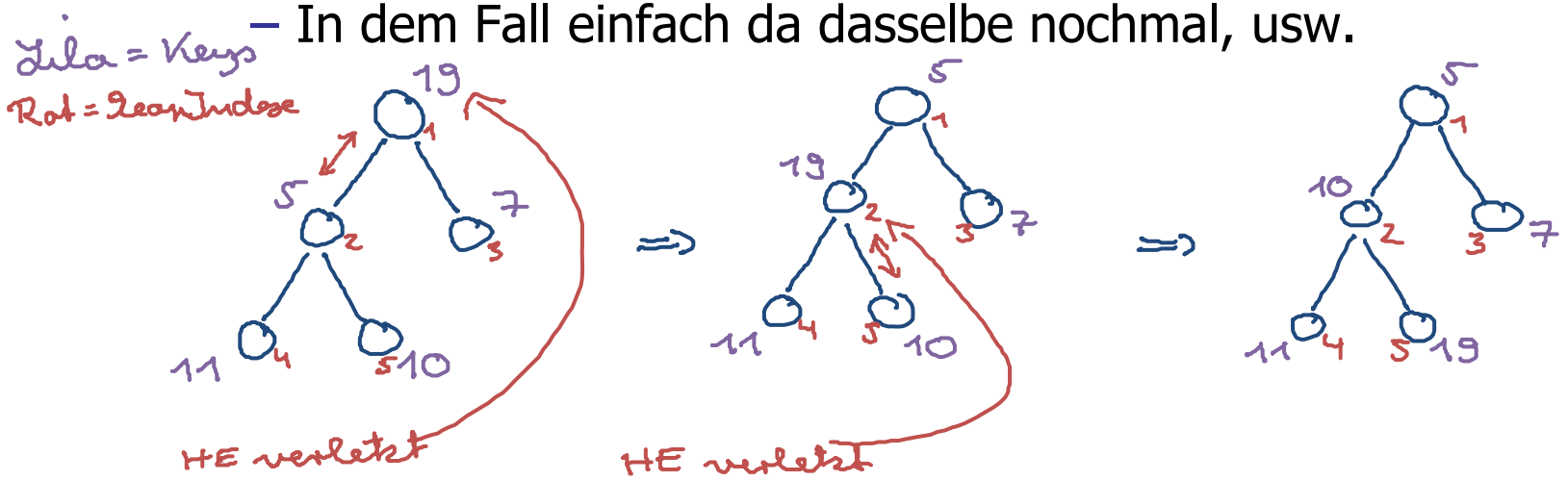
    repairHeapDownwards

    repairHeapUpwards
  - Siehe die nächsten drei Folien ...

Reparieren der Heapeigenschaft 2/4

- Methode repairHeapDownwards
  - Knoten x mit dem Kind y tauschen, das den kleineren Key von den beiden Kindern hat
  - Jetzt ist bei diesem Kind evtl. die Heapeigenschaft verletzt
  - Wenn, dann Key > der von den Kindern

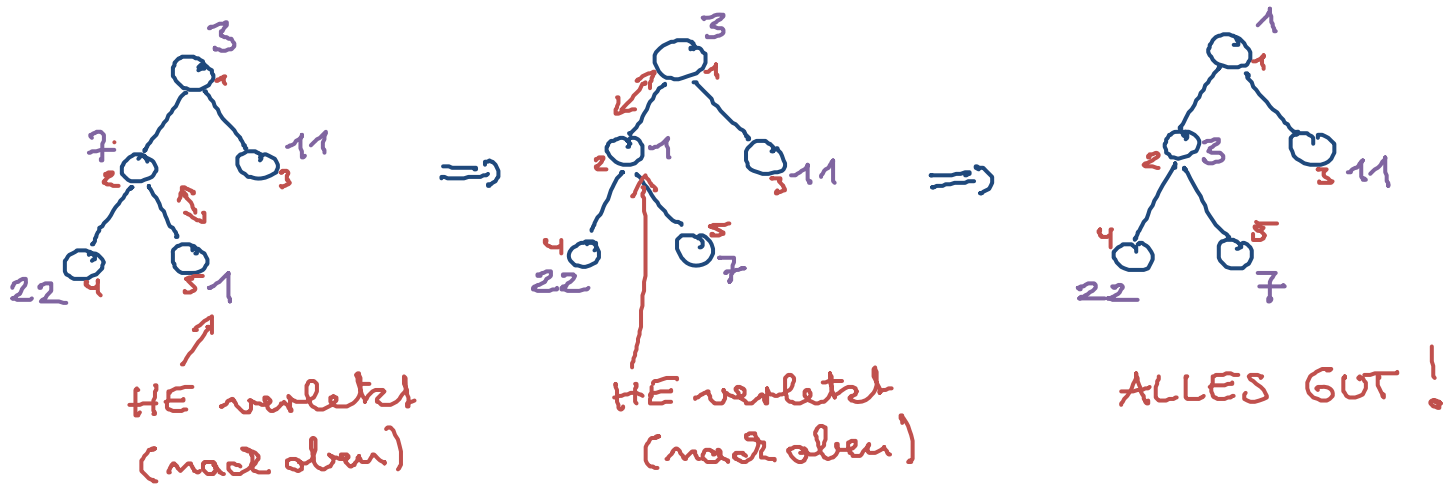
    Key von x > Key von y (deshalb haben wir x und y getauscht)
  - In dem Fall einfach da dasselbe nochmal, usw.



Reparieren der Heapeigenschaft 3/4

- Methode repairHeapUpwards
  - Knoten x mit dem Elternknoten y tauschen
  - Jetzt ist bei dem Elternknoten evtl. die Heapeig. verletzt
  - Wenn, dann Key < der von dessen Elternknoten

    Key von x < Key von y (deshalb haben wir x und y getauscht)
  - In dem Fall einfach da dasselbe nochmal, usw.



Reparieren der Heapeigenschaft 4/4

UNI FREIBURG

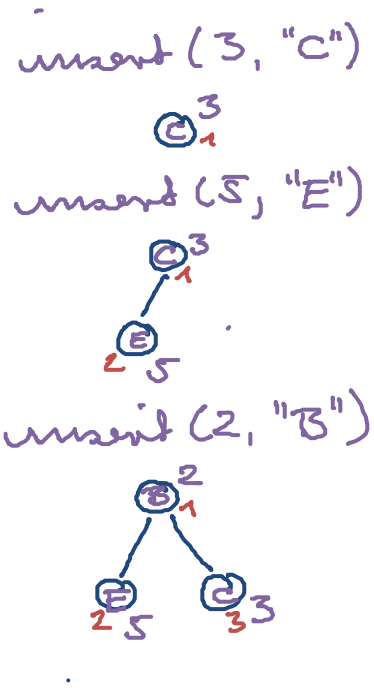
- Index eines PriorityQueueItems
  - Achtung: für changeKey und remove muss ein PriorityQueueItem wissen, wo es im Heap steht

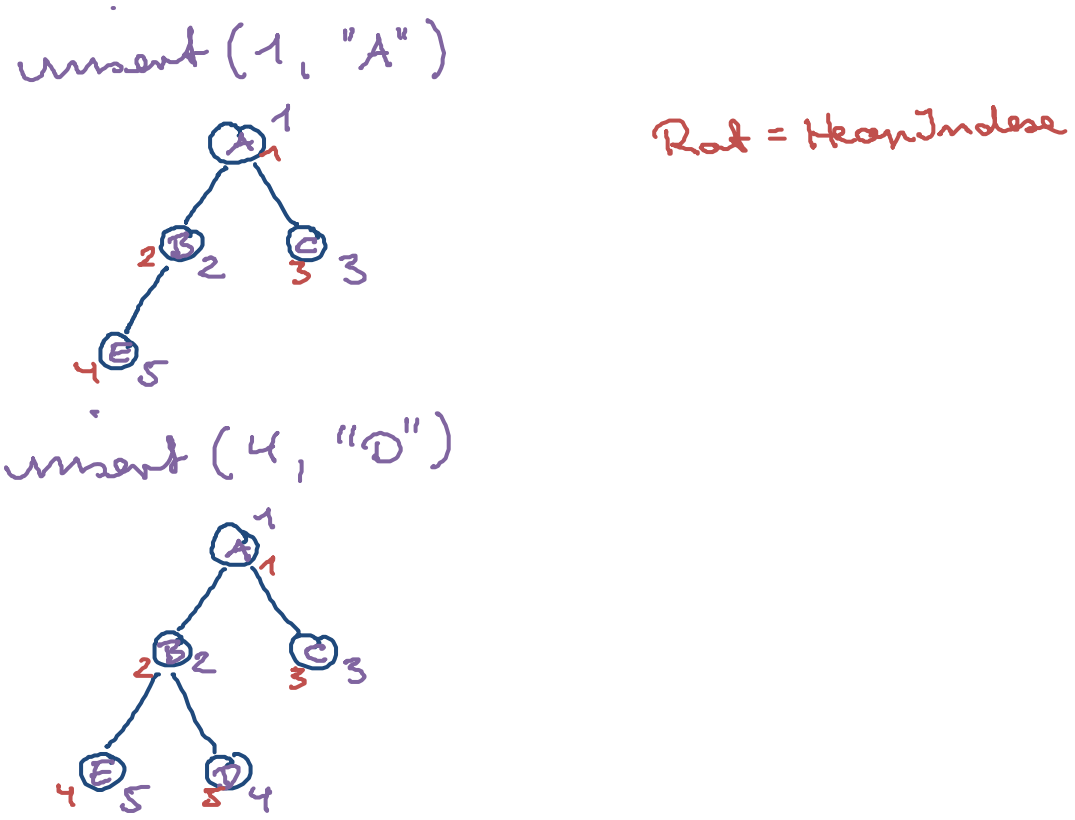
```
class PriorityQueueItem {
  int key;
  Object value;   // In C++, use a template T.
  int heapIndex;
}
```

  - Bei repairHeapDownwards und repairHeapUpwards beachten:

    Wann immer wir ein Element im Heap verschieben, muss der heapIndex des Elementes geupdated werden !

Beispiel (aus unserem Unit Test)





Root = HeapIndex

Komplexität 1/3

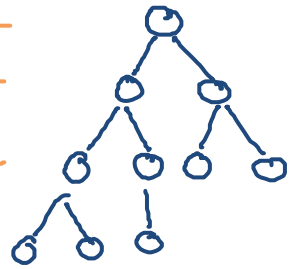
bis auf „unten rechts"

UNI FREIBURG

- Tiefe eines vollständigen binären Baumes
  - Ein vollständiger binärer Baum mit n Elementen hat Tiefe d = O(log n)

    Das heißt, die Anzahl der Elemente auf einem Pfad von einer beliebigen Position im Heap nach oben zur Wurzel oder nach unten zu einem Blatt ist O(log n)

Tiefe 0 – $= 2^0$
Tiefe 1 – $= 2^1$
Tiefe 2 – $= 2^{d-1}$
Tiefe d $\leq 2^d$



$$\Rightarrow n \geq \sum_{i=0}^{d-1} 2^i + 1$$

$$= \underbrace{1 + 2 + 4 + 8 + \cdots + 2^{d-1}}_{2^d - 1} + 1$$

$$= 2^d$$

$$\Rightarrow d \leq \log_2 n = O(\log n)$$ ∎

Komplexität 2/3

UNI FREIBURG

- Damit gilt für die Laufzeit
  - Für repairHeapDownwards : $O(d) = O(\log n)$
  - Für repairHeapUpwards : $O(d) = O(\log n)$
  - Für insert : $O(d) = O(\log n)$
  - Für deleteMin : $O(d) = O(\log n)$
  - Für changeKey : $O(d) = O(\log n)$
  - Für remove : $O(d) = O(\log n)$
  - Für getMin : $O(1)$
  - Es geht noch besser, siehe Vorlesung morgen ...

Komplexität 3/3

$n \geq 1 + b + b^2 + \cdots + b^{d-1}$

$= \frac{b^d - 1}{b-1} \Rightarrow b^d \leq b \cdot n + 1$

$d = O(\log_b n)$

$2/\ln 2 = 2.885\ldots$

$3/\ln 2 = 2.730\ldots$

FAZIT: wir bleiben bei $b = 2$

- Erweiterung auf b-ary heap
  - Also einen vollständigen b-ären Baum = jeder Knoten hat genau b Kinder, außer evtl. "unten rechts"
  - Die Elemente könnten dann weiterhin in einem Feld gespeichert werden, und Kind / Eltern (wenn auch nicht mehr ganz so einfach) über die Indizes berechnet werden
  - Laufzeit von repairHeapUpwards dann $\Theta(\log_b n)$
  - Laufzeit von repairHeapDownwards dann $\Theta(b \cdot \log_b n)$
  - Für welches b (ganzzahlige) is $b \cdot \log_b n$ optimal ?

$\left[\frac{f}{g}\right]' = \frac{f'g - fg'}{g^2}$

$f(x) = x \cdot \log_x n = x \cdot \frac{\ln n}{\ln x} = \frac{x}{\ln x} \cdot \ln n$ (hängt nicht von x ab)

$f'(x) = \frac{1 \cdot \ln x - x \cdot \frac{1}{x}}{\ln^2 x} \cdot \ln n$ wo ist das minimal

$= \frac{\ln n}{\ln^2 x}[\ln x - 1] \stackrel{!}{=} 0 \Rightarrow x = e$, und da ist ein MIN (weil $f''(e) > 0$)

Literatur / Links

UNI FREIBURG

- Prioritätswarteschlangen
  - In Mehlhorn/Sanders:

    6 Priority Queues [einfache und fortgeschrittenere Varianten]
  - In Cormen/Leiserson/Rivest

    20 Binomial Heaps [gleich die fortgeschrittenere Variante]
  - In Wikipedia

    http://de.wikipedia.org/wiki/Vorrangwarteschlange

    http://en.wikipedia.org/wiki/Priority_queue
  - In C++ und in Java

    http://www.sgi.com/tech/stl/priority_queue.html

    http://download.oracle.com/javase/1.5.0/docs/api/java/util/PriorityQueue.html