

Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 9a, Dienstag, 27. Juni 2017
(Prioritätswarteschlangen, Binäre Heaps)

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

Blick über die Vorlesung heute

■ Organisatorisches

- Diese Woche alles **online** Und eine Umfrage dazu
- Erfahrungen **ÜB8** Suchbäume

■ Inhalt

- Prioritätswürgeschlangen Motivation + Definition
- Implementierung Prinzip + Code + Laufzeit
- **ÜB9, Aufgabe 1: Effiziente Berechnung der k größten Städte in unserer cities.txt mit einer PW**

Sie müssen die PW nicht selber implementieren, sondern können die von Python/Java/C++ verwenden → Folien 13-15

Blick über die Vorlesung heute

- Diese Woche ist **alles** online (auch die Vorlesungen)
 - Grund: Terminkonflikte diese Woche (siehe Post im Forum)
 - Es gibt aber aktuelle Videoaufzeichnungen
 - Den Beginn haben wir im Voraus neu aufgenommen
 - Der Rest ist von den Aufzeichnungen vom **SS 2015**
 - Uns interessieren Ihre Erfahrungen damit, insbesondere:
 - Gibt es für Sie einen Nachteil gegenüber der Live-Vorlesung?
 - Warum gab es in den letzten Vorlesungen so wenige Nachfragen? (trotz zahlreicher Anregungsversuche meinerseits)
- Siehe Frage dazu am Ende vom ÜB9

Erfahrungen mit dem ÜB8 1/2

■ Zusammenfassung / Auszüge

Stand 26. Juni 19:00

- Die freiere Aufgabenstellung (bei Aufgabe 1) hat vielen gefallen und Spaß gemacht
- Zahlreiche Fragen zum Umgang mit Sonderzeichen

Haben wir extra nicht entfernt, weil man die in **jedem** größeren Datensatz hat und also damit umgehen muss

Ausführliche Hilfestellung dazu auf dem Forum

- Einige haben trotz Warnung die cities.txt mit hochgeladen!
- Gar nicht so einfach, keinen Roman zu schreiben

■ Datenstruktur Aufgabe 1, zwei gute Optionen

- **Option 1:** balancierter Suchbaum (wer hätte das gedacht)

Pro Anfrage braucht man einen lookup (Zeit $O(\log n)$) und eine Folge von next Operationen (Zeit je $O(1)$)

Der Aufbau des Baumes geht in Zeit $O(n \cdot \log n)$ und wenn man ausnutzt, dass Eingabe schon sortiert, sogar in Zeit $O(n)$

- **Option 2:** sortiert abspeichern + binäre Suche / Anfrage
in $O(n \cdot \log n)$

Man kann die Eingabe auch erstmal ganz einlesen und dann einmal sortieren (oder ausnutzen, dass bereits sortiert)

Pro Anfrage reichen dann zwei binäre Suchen → Zeit $O(\log n)$

Option 2 sogar besser, weil effizienter beim Einlesen und einfacher (man braucht keine komplexe Datenstruktur)

■ Definition

- Eine **Prioritätswarteschlange (PW)** verwaltet eine Menge von Key-Value Paaren bzw. Elementen und es gibt wieder eine Ordnung \leq auf den Keys
- Es werden folgende Operationen unterstützt:
 - insert(item)**: füge das gegebene Element ein
 - getMin()**: gebe das Element mit dem kleinsten Key zurück
 - deleteMin()**: entferne das Element mit dem kleinsten Key
 - changeKey(item)**: ändere Key des gegebenen Elementes
 - remove(item)**: entferne das gegebene Element

■ 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, wobei n = Anzahl Schlüssel

- Der **BinarySearchTree** kann alles was eine **PriorityQueue** kann und **mehr** (nämlich **lookup** von beliebigen **Elem.**)

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 Folien 8 – 11

■ Mehrere Elemente mit dem gleichen Key

- Das ist für viele PW-Anwendungen nötig und darf man deswegen nicht einfach ausschließen
- Man muss dann nur klären, welches Element `getMin` und `deleteMin` auswählen, wenn es mehrere kleinste Keys gibt
- Das übliche Vorgehen ist so

`getMin` : gibt irgendein Element mit kleinstem Key zurück

`deleteMin` : löscht eben dieses Element

Bei unserer Implementierung gleich wird das quasi "von selber" der Fall sein

- Argument der Operationen **changeKey** und **remove**
 - Eine **PW** erlaubt **keinen** Zugriff auf ein beliebiges Element
 - Deshalb geben **insert** und **getMin** eine Referenz auf das entsprechende Element zurück
 - Mit so einer Referenz kann man dann später über **changeKey** bzw. **remove** den Schlüssel ändern bzw. das Element entfernen

■ Anwendungsbeispiel 1

- Man kann mit einer PW einfach **sortieren**, und zwar so:

Alle Elemente einfügen: `insert(x1)`, `insert(x2)`, ..., `insert(xn)`

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 $O(n \cdot \log n)$ Zeit

Also asymptotisch optimal für vergleichsbasiertes Sortieren

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

und noch nicht
sortiert

■ Anwendungsbeispiel 2

- Mischen von k sortierten Listen ... **englisch: k-way merge**
- Dazu k "Zeiger", auf jede Liste einen ... in jeder Iteration das kleinste von den betreffenden Elementen berechnen
- Das geht mit einer PW in Zeit $O(\log k)$ pro Iteration, also insgesamt Zeit $O(n \cdot \log k)$... n = Gesamtzahl Elemente

*initial wäre
 $\Theta(k)$ Zeit*

$L_1 : \overset{\cdot}{5}, \overset{\cdot}{7}, \overset{\cdot}{15}, \overset{\cdot}{23}^x$
 $L_2 : \overset{\cdot}{8}, \overset{\cdot}{9}, \overset{\cdot}{10}^x$
 $L_3 : \overset{\cdot}{11}, \overset{\cdot}{17}, \overset{\cdot}{19}, \overset{\cdot}{25}^x$

$R : 5, 7, 8, 9, 10, 11, 15, \dots$

RESULT

■ Anwendungsbeispiel 3

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

Das machen wir nächste Woche !

■ Anwendungsbeispiel 4

- Man kann mit einer PW einfach und effizient die k größten Elemente von einer Menge berechnen

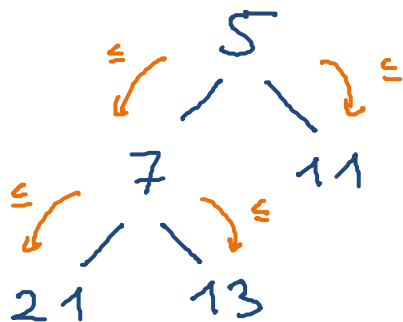
Das ist Aufgabe 1 vom Ü9

Allgemeiner geht das auch dann noch effizient, wenn sich die Menge laufend ändert, und zwar in Zeit $O(\log k)$ pro Element, das dazukommt bzw. weggenommen wird

■ Grundidee

- Wir speichern die Elemente in einem **binären Heap**
- Das ist ein **vollständiger binärer Baum**, für dessen Schlüssel die **Heap-Eigenschaft (HE)** gilt
- **HE** = Key jedes Knotens \leq die Keys von beiden Kindern

Das ist eine **schwächere** Eigenschaft als beim binären Suchbaum, insbesondere sind die Blätter **nicht** sortiert



Jede male hier und im Folgenden nur die Keys sein, nicht die Values (die stehen einfach immer nur mit dabei)

PW – Implementierung 2/15

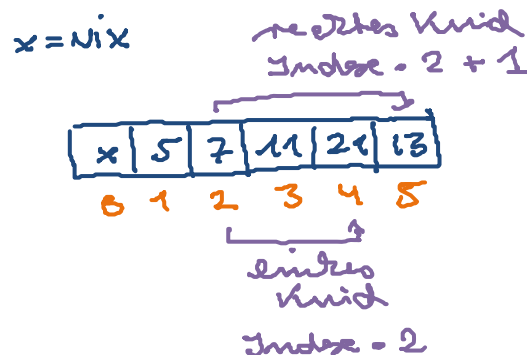
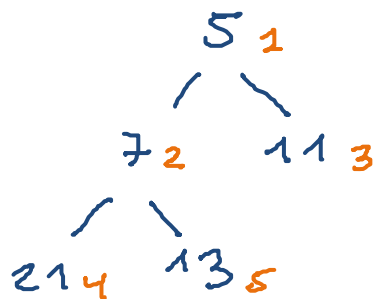
*dann ist die Formel einfacher
um zu Kindern / Eltern zu kommen*

■ Wie speichert man einen binären Heap

- Anders als beim **BinarySearchTree** geht das **ohne Zeiger**
- Wir nummerieren die Knoten von oben nach unten und von links nach rechts durch, beginnend mit **1**
- Wir können die Elemente dann in einem Feld speichern, und leicht zu Kinder- bzw. Elternknoten springen:

Die Kinder von Knoten i sind Knoten $2i$ und $2i + 1$

Der Elternknoten von einem Knoten i ist Knoten $\lfloor i/2 \rfloor$

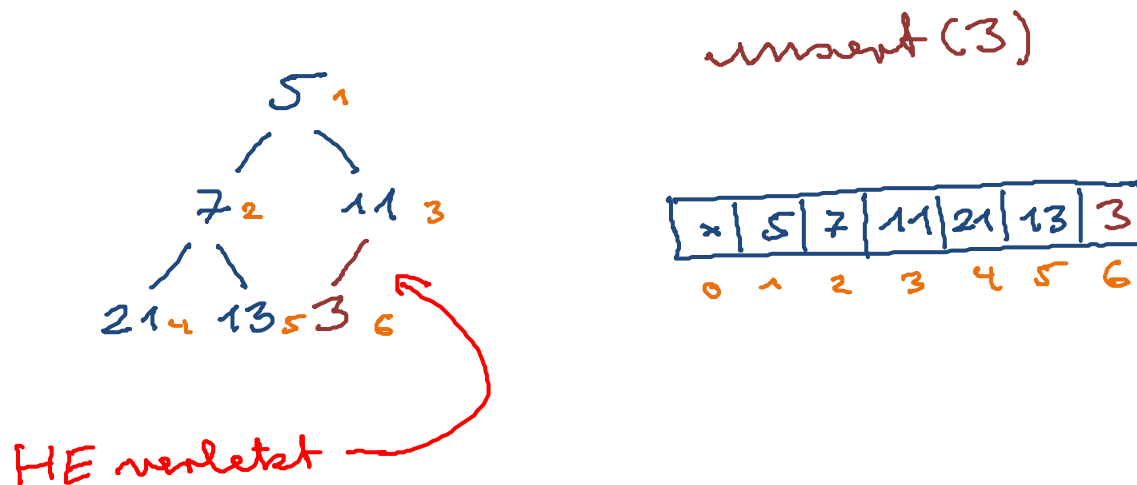


■ Die Operation **insert**

- Erstmal hinzufügen am Ende des Feldes
- Danach kann die Heapeigenschaft verletzt sein

Aber nur genau an dieser (letzten) Position

Wiederherstellung der HE siehe Folien 20 und 21



- Die Operation `getMin`

- Einfach das oberste Element zurückgeben

Im Feld ist das einfach das Element an Position 1

Falls Heap leer, einfach null zurückgeben

■ Die Operation `deleteMin`

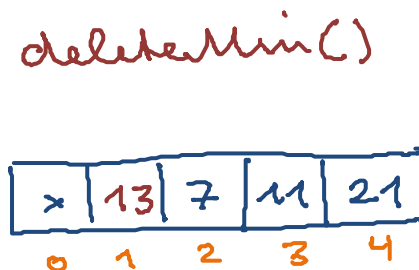
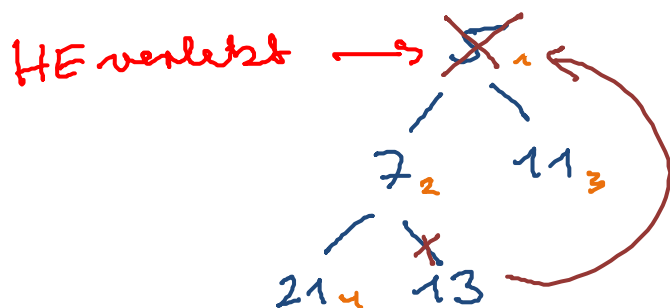
- Einfach das Element von der letzten Position an die erste Stelle setzen (falls Heap nicht leer)

Element an der ersten Stelle wird dabei überschrieben

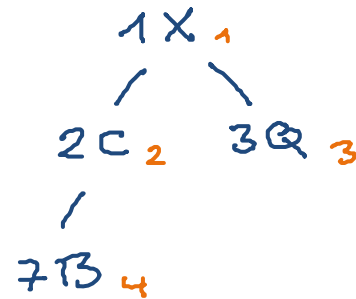
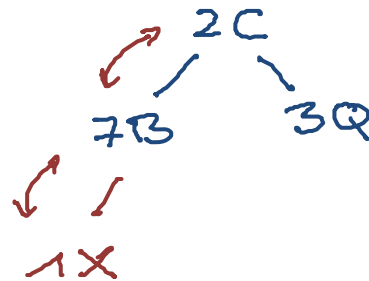
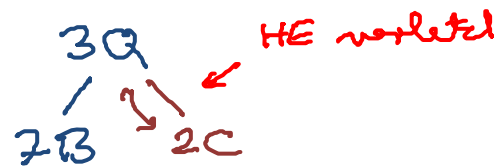
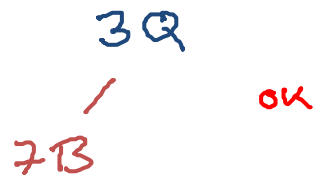
- Danach kann die Heapeigenschaft (HE) verletzt sein

Aber wieder nur genau an dieser (ersten) Position

Wiederherstellung der HE siehe Folie 21



insert(3Q)
 insert(7B)
 insert(2C)
 insert(1X)



x	1X	2C	3Q	7B
0	1	2	3	4

■ Die Operation `changeKey`

- Das Element wird als Argument übergeben ... wir können also einfach seinen Schlüssel ändern

- Danach kann die Heapeigenschaft (HE) verletzt sein

Aber wieder nur genau an dieser Position

Wiederherstellung der HE siehe Folien 20 und 21

Element muss dazu seine Position im Feld wissen

Siehe dazu Folie 22

■ Die Operation **remove**

- Das Element wird als Argument übergeben
- Element von der letzten Position an diese Stelle setzen
- Danach kann die Heapeigenschaft (HE) verletzt sein

Aber wieder nur genau an dieser Position

Element muss dazu wieder seine Position im Feld wissen

Siehe Folien 20 und 21

ähnlich wie deleteMin

■ Reparieren der Heapeigenschaft

- Nach `insert`, `deleteMin`, `changeKey`, `remove` kann die Heapeigenschaft (HE) verletzt sein

Aber nur an genau einer (bekannten) Position i

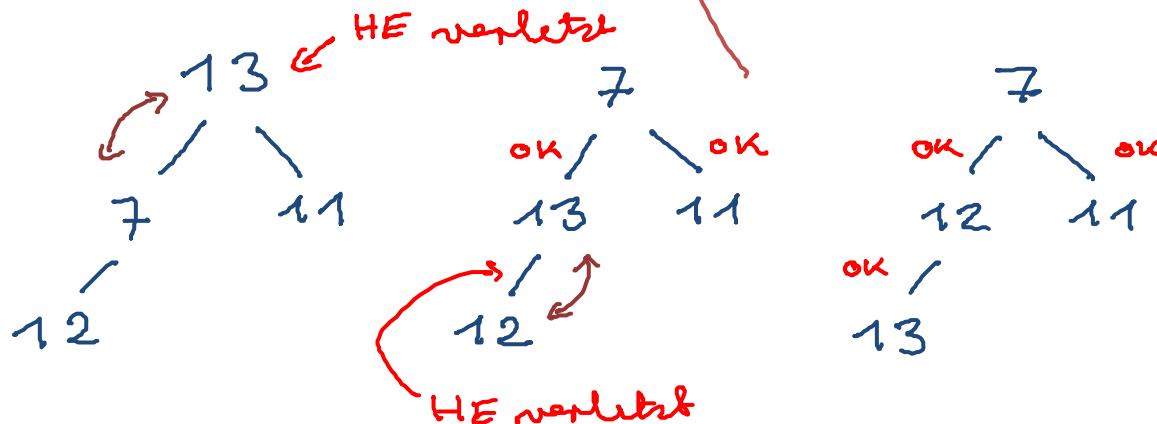
- Die HE kann auf eine von zwei Arten verletzt sein
 - Schlüssel an Position i ist nicht \leq der seiner Kinder
 - Schlüssel an Position i ist nicht \geq der vom Elternkn.
- Entsprechend brauchen wir zwei Reparaturmethoden `repairHeapDownwards` und `repairHeapUpwards`

■ Methode `repairHeapDownwards`

- Knoten `x` mit dem Kind `y` tauschen, das den kleineren Key von den beiden Kindern hat

Zum Elternknoten hin stimmt dann mit Sicherheit alles

- Jetzt ist bei diesem Kind eventuell die `HE` verletzt, indem sein Key größer ist als der von einem der beiden Kinder
- In dem Fall einfach da dasselbe nochmal, usw.



*Säßen wir mit der
11 getauscht wäre
es nicht OK*

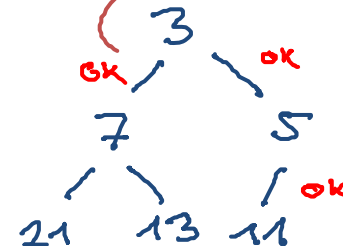
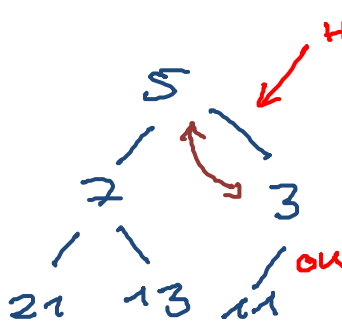
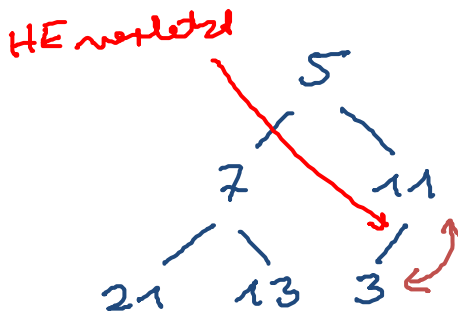
■ Methode `repairHeapUpwards`

- Knoten `x` mit dem Elternknoten `y` tauschen

Zum Kindknoten hin stimmt dann mit Sicherheit alles

- Jetzt ist bei dem Elternknoten eventuell die `HE` verletzt
- In dem Fall einfach da dasselbe nochmal, usw.

3 zu 7 ok, weil
etwas kleineres
nach oben getauscht
wurde



■ Index eines Elementes in der PW

5C@3 ↔ 1X@7
1X@3 5C@7

- **Achtung:** für `changeKey` und `remove` muss ein Element wissen, wo es im Heap steht
- Lösung: jedes Element hat eine zusätzliche Variable `heapIndex`, die angibt, wo es im internen Feld steht
- Wann immer das Element im Feld verschoben wird, darauf achten, `heapIndex` entsprechend anzupassen

Element wird nur innerhalb von `repairHeapDownwards` und `repairHeapUpwards` verschoben

■ Laufzeit

- Die Laufzeit von `repairHeapDownwards` bzw. `Upwards` ist $O(d)$, wobei d die Tiefe des (vollst. binären) Baumes ist
- Aus Vorlesung 8a wissen wir: $d = O(\log n)$
- Die Operationen `insert`, `deleteMin`, `changeKey` und `remove` laufen also in $O(\log n)$ Zeit

Nach jeder davon muss die HE wiederhergestellt werden

Es geht auch noch schneller, siehe Vorlesung 9b morgen

- Die Operation `getMin` läuft sogar in $O(1)$ Zeit

Das Minimum ist einfach das oberste Element im Heap

- Benutzung in **Java** `import java.util.PriorityQueue;`
 - Element-Typ unterscheidet nicht zwischen Key und Value
`PriorityQueue<T> pq;`
 - Defaultmäßig wird die Ordnung \leq auf `T` genommen
Eigene Ordnung über einen Comparator, wie bei sort
 - Operationen: `insert = add`, `getMin = peek`, `deleteMin = poll`
 - Die Operation `changeKey` gibt es nicht, dafür `remove`
Mit `remove` und `insert` kann man `changeKey` leicht simulieren

- Benutzung in C++
 - `#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`

Eine Implementierung ohne diese beiden ist effizienter, weil man den `heapIndex` pro Element nicht braucht

In Vorlesung 10a sehen wir, wie man ohne `changeKey` zurechtkommt, wenn man es eigentlich doch braucht

■ Benutzung in **Python** `from queue import PriorityQueue`

- Elemente sind beliebige Tupel

```
pq = PriorityQueue()  
pq.put((priority, value))
```

- Es wird die Ordnung auf den Tupeln genommen
- Operationen: `insert = put`, `deleteMin = get`
- Statt `getMin` kann man `pq.queue[0]` aufrufen
`queue` ist das interne Feld, aber Beginn bei 0
- Die Operationen `changeKey` und `remove` gibt es nicht
Workaround siehe Bemerkung auf der Folie vorher

■ Prioritätswarteschlangen

- In Mehlhorn/Sanders:

 - 6 Priority Queues [einfache und fortgeschrittenere Varianten]

- In Wikipedia

 - <http://de.wikipedia.org/wiki/Vorrangwarteschlange>

 - http://en.wikipedia.org/wiki/Priority_queue