Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 9a, Dienstag, 23. Juni 2015
(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)
- Inhalt: Prioritätswarteschlangen
  - Grundlagen … Definition + Anwendungen
  - Implementierung … Prinzip + Code + Laufzeit
  - Ü9, Aufgabe 1: Effiziente Berechnung der k größten Städte in cities.txt mit einer PW

    Sie können dazu die PW von Python / Java / C++ verwenden, siehe dazu Folien 13 – 15

Erfahrungen mit dem Ü8 (Suchbäume)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 23. Juni 12:00
  - Die cities.txt Aufgabe fanden die meisten interessant
  - Einige Probleme mit den Sonderzeichen bzw. Fragen dazu

    UTF-8 als ISO-8859-1 sieht merkwÃ¼rdig aus
  - Zeitmessung von einzelnen lookups teilweise schwierig

    C++: lookup zu schnell bzw. Zeit von clock() zu grob

    Java: die Java VM funkt immer wieder dazwischen
  - Sommersonnenwende: Rollläden unten lassen … sonst wird man extrem früh (gegen 12 Uhr) von der Sonne geweckt

PW – Grundlagen 1/8

UNI FREIBURG

- 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

PW – Grundlagen 2/8

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 Θ(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

PW – Grundlagen 3/8

UNI FREIBURG

- 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

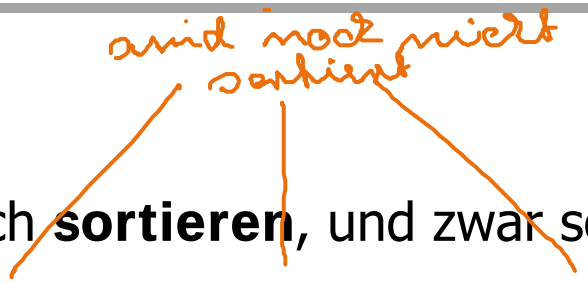
    Bei unserer Implementierung gleich wird das quasi "von selber" der Fall sein

PW – Grundlagen 4/8

UNI FREIBURG

- 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

PW – Grundlagen 5/8



- Anwendungsbeispiel 1
  - Man kann mit einer PW einfach **sortieren**, und zwar so:

    Alle Elemente 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

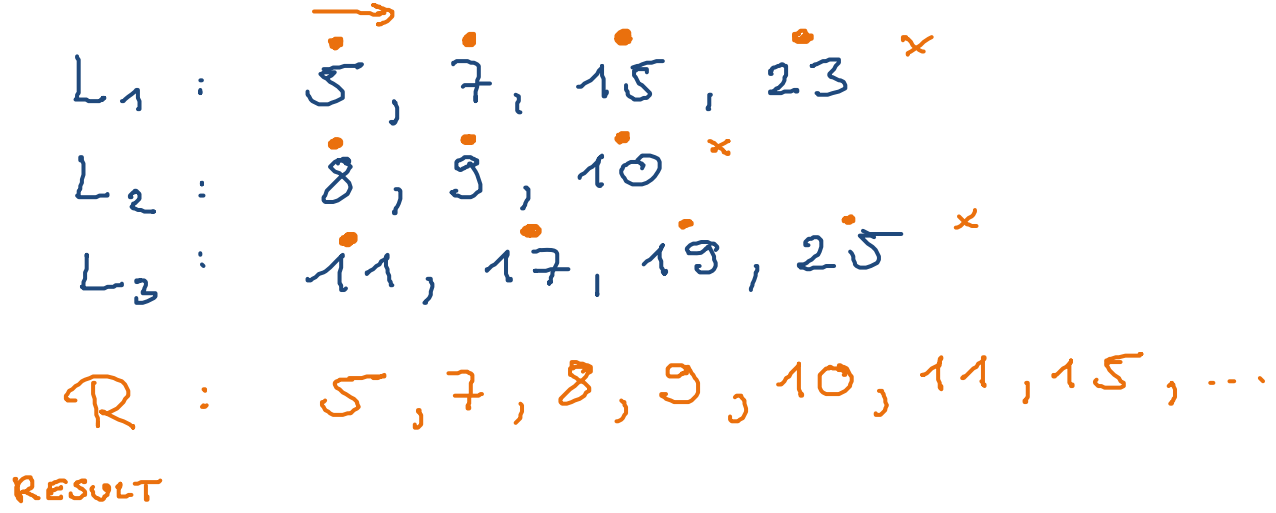
    Insbesondere genauso gut wie MergeSort (im allgemeinen Fall) und QuickSort (im besten Fall)

PW – Grundlagen 6/8

UNI FREIBURG

- 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 · log k) … n = Gesamtzahl Elemente

trivial wäre $\Theta(k)$ Zeit



PW – Grundlagen 7/8

UNI FREIBURG

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

    Das machen wir nächste Woche !

PW – Grundlagen 8/8

UNI FREIBURG

- Anwendungsbeispiel 4
  - Man kann mit einer PW einfach und effizient die k größten Elemente von einer Menge berechnen

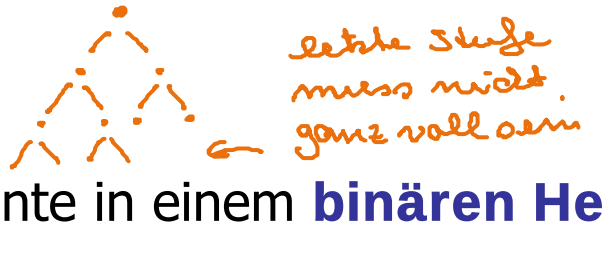
    Das ist Aufgabe 1 vom Ü9

    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

PW – Implementierung 1/15

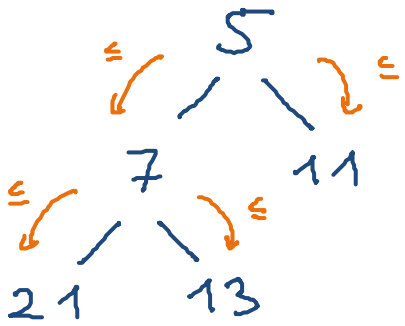
UNI FREIBURG

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



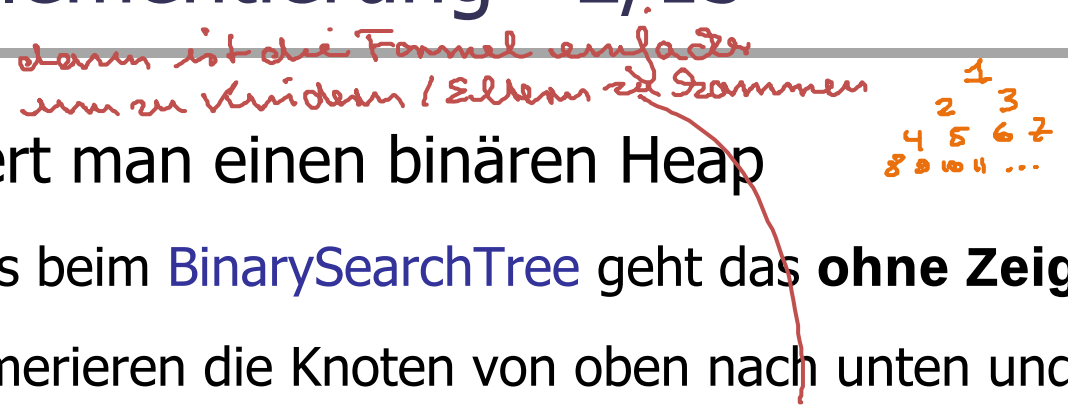
  - Das ist ein vollständiger binärer Baum, für dessen Schlüssel die Heap-Eigenschaft (HE) gilt
  - HE = Key jedes Knotens ≤ die Keys von beiden Kindern

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



Ich male drei und im Folgenden nur die Keys dran, nicht die Values (die stehen einfach immer mit dabei)

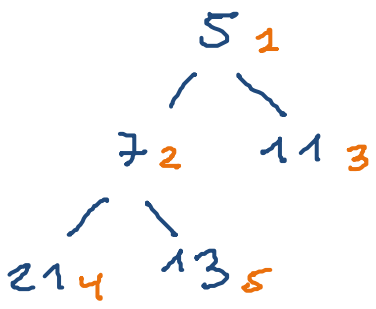
PW – Implementierung 2/15

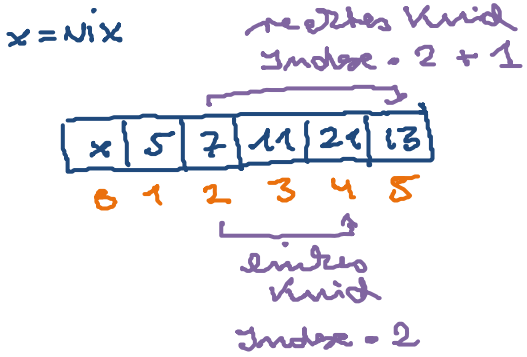


- Wie speichert man einen binären Heap
  - Anders als beim BinarySearchTree geht das **ohne Zeiger**
  - Wir nummerieren die Knoten von oben nach unten und von links nach rechts durch, beginnend mit **1**
  - Wir können die Elemente dann in einem Feld speichern, und leicht zu Kinder- bzw. Elternknoten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

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

    Der Elternknoten von einem Knoten i ist Knoten ⌊i/2⌋





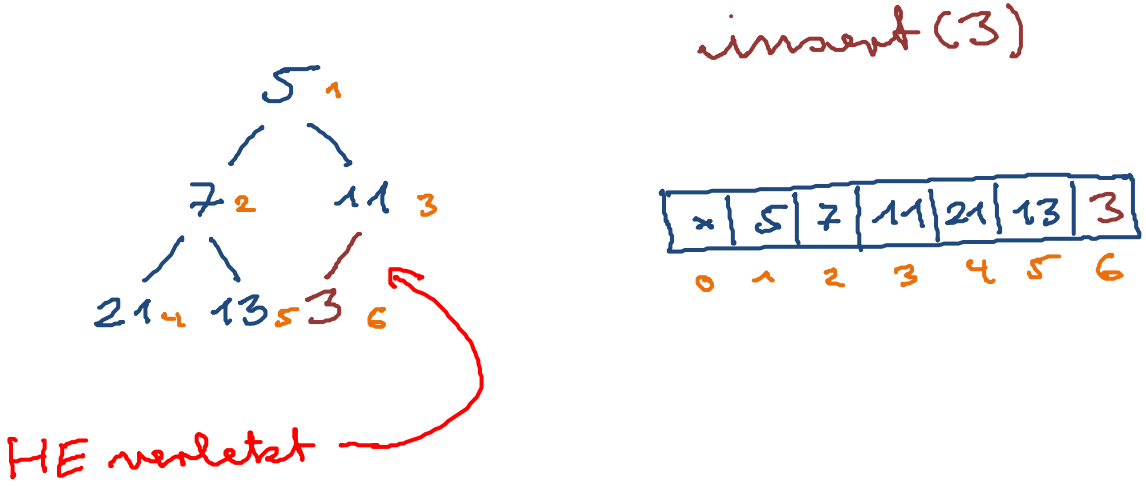
PW – Implementierung 3/15

UNI FREIBURG

- 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



PW – Implementierung 4/15

UNI FREIBURG

- Die Operation getMin
  - Einfach das oberste Element zurückgeben

    Im Feld ist das einfach das Element an Position 1

    Falls Heap leer, einfach null zurückgeben

PW – Implementierung 5/15

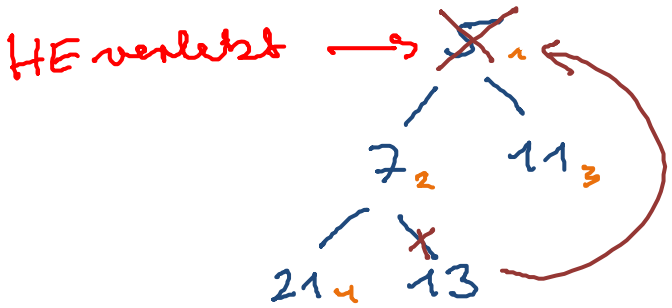
UNI FREIBURG

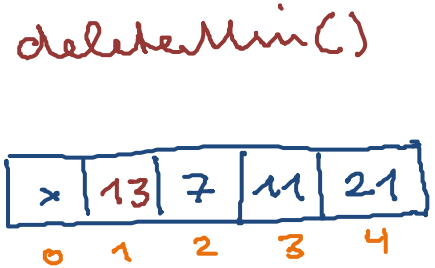
- Die Operation deleteMin
  - Einfach das Element von der letzten Position an die erste Stelle setzen (falls Heap nicht leer)

    Element an der ersten Stelle wird dabei überschrieben
  - Danach kann die Heapeigenschaft (HE) verletzt sein

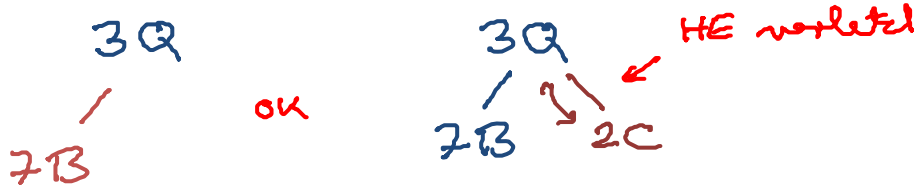
    Aber wieder nur genau an dieser (ersten) Position

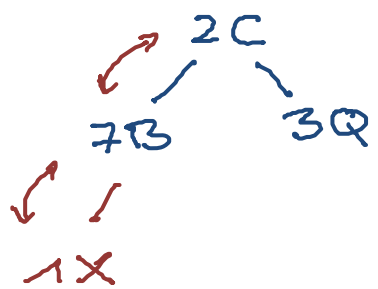
    Wiederherstellung der HE siehe Folie 21

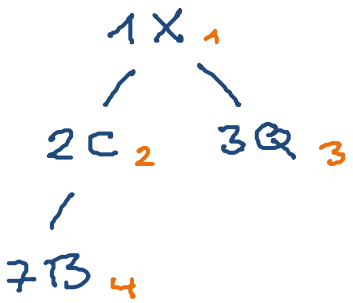


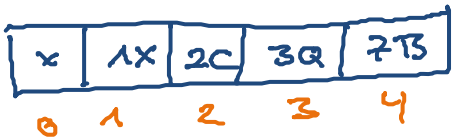


insert(3Q)
insert(7B)
insert(2C)
insert(1X)









PW – Implementierung 6/15

UNI FREIBURG

- 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

PW – Implementierung 7/15

UNI FREIBURG

- 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

PW – Implementierung 8/15

UNI FREIBURG

- 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 ≤ der seiner Kinder

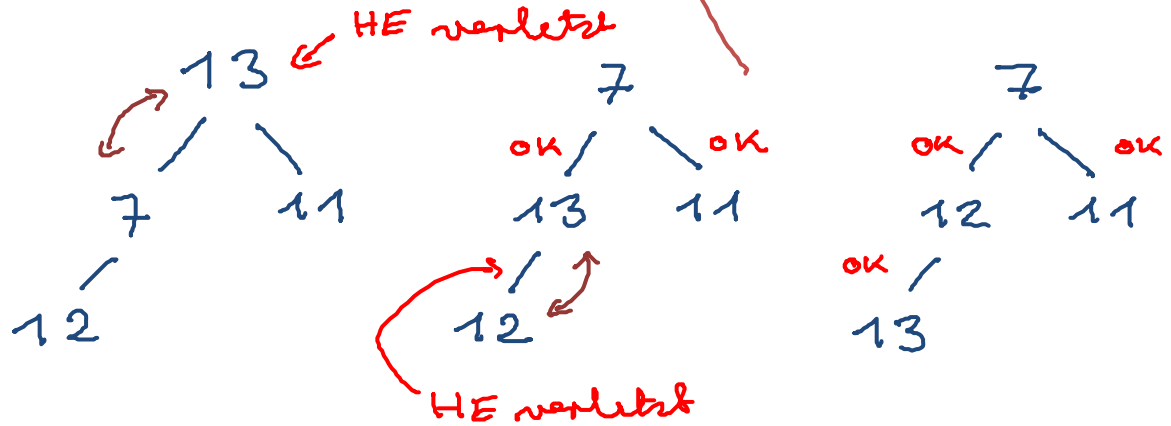
    Schlüssel an Position i ist nicht ≥ der vom Elternkn.
  - Entsprechend brauchen wir zwei Reperaturmethoden

    repairHeapDownwards und repairHeapUpwards

PW – Implementierung 9/15

- 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.

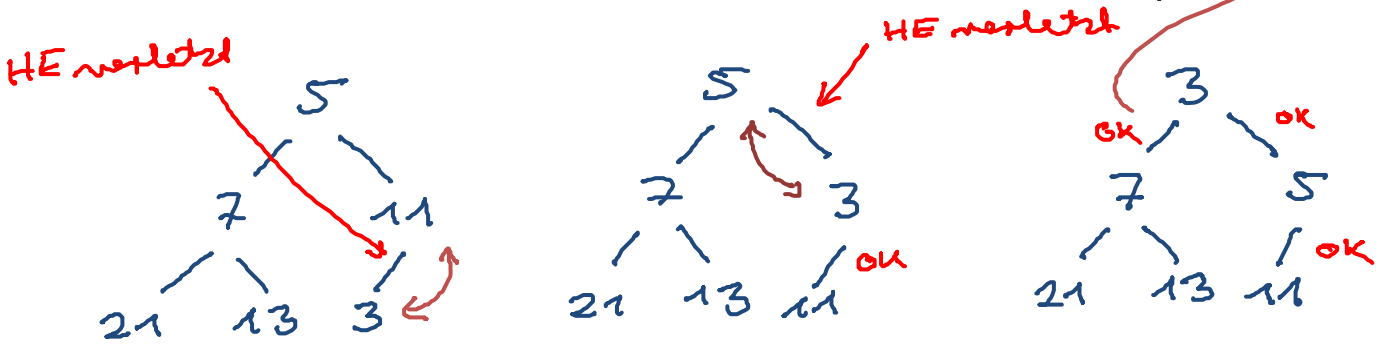


PW – Implementierung 10/15

UNI FREIBURG

- 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.



PW – Implementierung 11/15

5c@3 ↔ 1x@7
1x@3 5c@7

- Index eines Elementes in der PW
  - **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

PW – Implementierung 12/15

UNI FREIBURG

- 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

PW – Implementierung 13/15

UNI FREIBURG

- Benutzung in **Java** 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
  - Operationen: insert = add, getMin = peek, deleteMin = poll
  - Die Operation changeKey gibt es nicht, dafür remove

    Mit remove und insert kann man changeKey leicht simulieren

PW – Implementierung 14/15

UNI FREIBURG

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

    `std::priority_queue<T> pq;`
  - Es wird die Ordnung ≥ auf T genommen, und nicht ≤
  - 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

PW – Implementierung 15/15

UNI FREIBURG

- 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

Literatur / Links

UNI FREIBURG

- 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