Algorithmen und Datenstrukturen (ESE)
Entwurf, Analyse und Umsetzung von Algorithmen (IEMS)
WS 2012 / 2013

Vorlesung 6, Dienstag 27. November 2012
(Prioritätswarteschlangen)

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 Ü5 (Universelles Hashing)
  - Nochmal die Pointe von universellem Hashing
  - Schauen Sie in's Forum, es lohnt sich !
- Prioritätswarteschlangen (Englisch: priority queues)
  - Ebenfalls eine Datenstruktur, die man sehr häufig braucht
  - Operationen: insert, getMin, deleteMin, changeKey
  - Anwendungsbeispiel
  - Benutzung in C++ und in Java
  - Implementierung mittels eines binären Heaps
  - Übungsblatt: Implementierung einer PriorityQuery mit den genannten Methoden

Erfahrungen mit dem Ü5 (Hashing)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 27. November 16:00
  - Gute praktische Aufgabe um die Theorie zu verstehen
  - Was genau bedeuten die Perzentile?
  - Programmieren kostet viele nach wie vor viel Zeit
  - Die Ergebnisse machen keinen Sinn, sie ergeben Sinn !
  - Verwirrung, ob Duplikate in Schlüsselmenge erlaubt ... nein
  - Ob Hashfunktion gut, hängt nicht von b ab ... genau !
  - Wie testet man Zufall?
  - Musterlösung in C++ ?

Schauen Sie in's Forum

UNI FREIBURG

- Es lohnt sich
  - Klärung von Unklarheiten auf dem Übungsblatt
  - Zusätzliche interessante Hintergrundinformationen
  - Diverse praktische Tipps
- Sie müssen ja nicht immer alles lesen
  - Sondern lernen Sie, schnell interessante von uninteressanten Information zu unterscheiden
  - Das ist eh ein wichtiger "soft skill"

Universelles Hashing Reprise 1/2

UNI FREIBURG

- Nochmal die Pointe
  - Keine Hashfunktion ist gut für **alle** Schlüsselmengen
    - das kann gar nicht gehen, weil ein großes Universum auf einen kleinen Bereich abgebildet wird
  - Für **zufällige** Schlüsselmengen tun es auch einfache Hashfunktionen wie $h(x) = x \bmod m$
    - dann sorgen die zufälligen Schlüssel dafür, dass es sich gut verteilt
  - Wenn man für **jede** Schlüsselmenge gute Hashfunktionen finden will, braucht man universelles Hashing
    - dann ist aber, für eine feste Schlüsselmenge, nicht jede Hashfunktion gut, sondern nur viele / die meisten

Universelles Hashing Reprise 2/2

UNI FREIBURG

- Rehash
  - Auch mit universellem Hashing kann man mal eine schlechte Hashfunktion erwischen, wenn auch unwahrscheinlich
  - Das kann man aber leicht feststellen, in dem man die maximale Bucketgröße misst
  - Wenn die einen vorgegeben Wert überschreit macht man einen sogenannten **Rehash**
    - Neue Hashtabelle mit neuer zufälliger Hashfunktion
    - Elemente von der alten in die neue Tabelle kopieren
    - Das ist teuer, wird aber selten passieren
    - Deshalb durschnittliche Kosten gering
    - Siehe Thema amortisierte Analye in der nächsten Vorlesung

Prioritätswarteschlangen 1/4

UNI FREIBURG

- Definition
  - Eine **Prioritätswarteschlange** (PW) speichert eine Menge von Elementen, von denen jedes einen Schlüssel hat von
    - Also Key-Value Paare, wie in einer Map auch
  - Es gibt eine totale Ordnung ≤ 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

bei uns einfach immer ≤ auf int.

Prioritätswarteschlangen 2/4

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 3/4

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 4/4

UNI FREIBURG

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

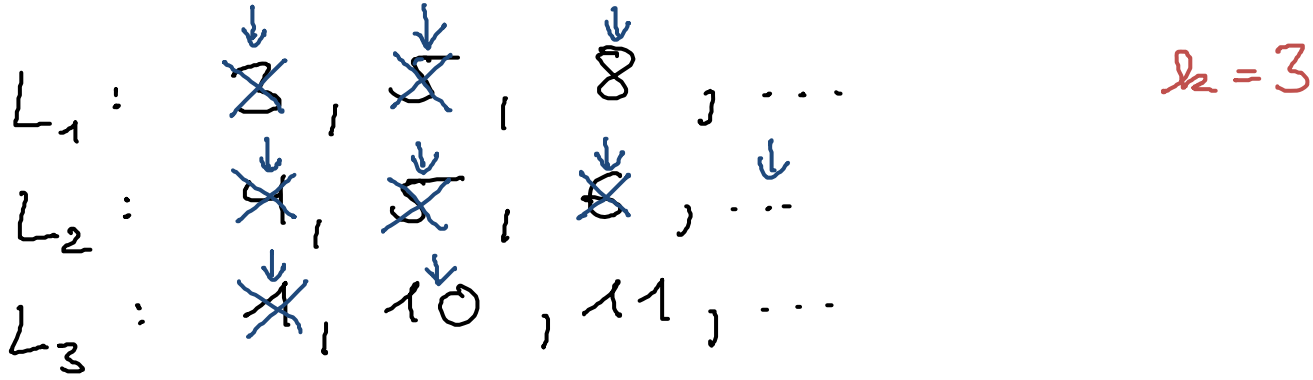
    std::priority_queue<T> pq;
  - Es wird die Ordnung ≥ auf T genommen, und nicht ≤
  - Beliebige Vergleichsfunktion wie bei std::sort
  - Operationen: insert = push, getMin = top, deleteMin = pop
  - Es gibt kein changeKey und auch kein beliebiges remove

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

PWs — Anwendungen 1/2

UNI FREIBURG

- Anwendungsbeispiel 1
  - Berechnung der Vereinigungsmenge von k sortierten Listen (sogenannter multi-way merge oder k-way merge)



PWs — Anwendungen 2/2

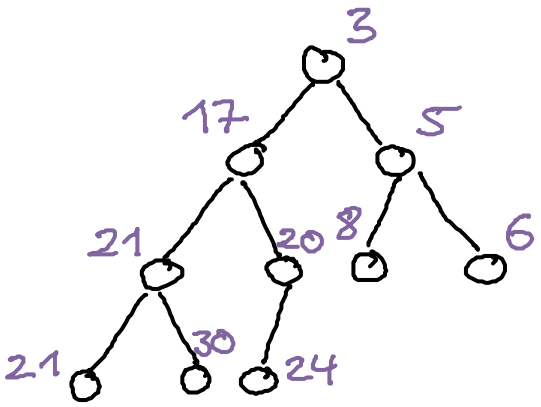
UNI FREIBURG

- Anwendungsbeispiel 2
  - Zum Beispiel für Dijkstra's Algorithmus zur Berechnung kürzester Wege → spätere Vorlesung
  - Unter anderem kann man damit auch einfach Sortieren

Implementierung 1/7

UNI FREIBURG

- Grundidee
  - Elemente in einem binären Heap speichern
  - Wiederholung aus der 1. Vorlesung (HeapSort):
    - vollständiger binärer Baum (bis evtl. "unten rechts")
    - es gilt die Heap-Eigenschaft = der Key jedes Knotens ist ≤ die Keys von den beiden Kindern

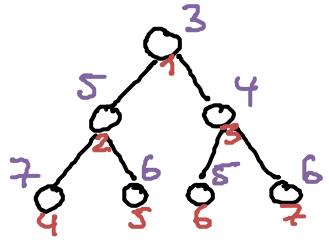


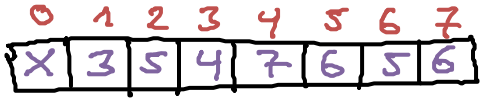
Implementierung 2/7

- Wie speichert man einen binären Heap
  - Ebenfalls Wiederholung aus Vorlesung 1:
  - 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)
  - Elemente stehen dann einfach in einem Array:

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

rot: Indizes
lila: Keys





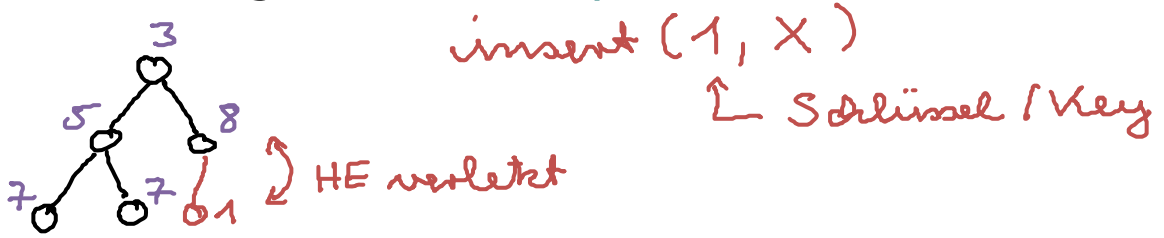
Implementierung 3/7

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

    heap.add(keyValuePair); // Java.

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

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



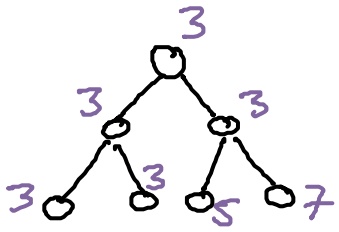
Implementierung 4/7

UNI FREIBURG

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

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

  - Achtung falls Heap leer, dann null zurückgeben



Implementierung 5/7

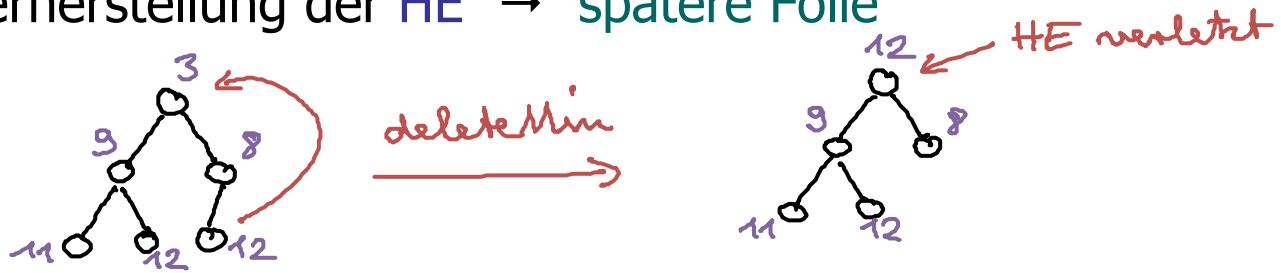
UNI FREIBURG

- 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 (HE) verletzt sein

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



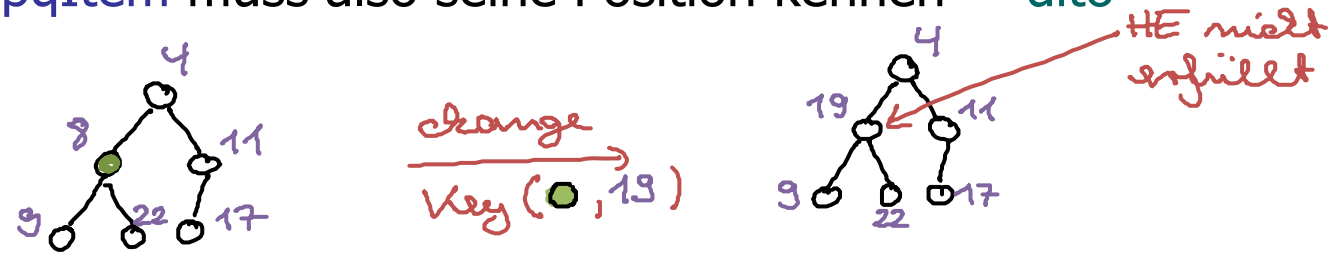
Implementierung 6/7

UNI FREIBURG

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

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

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



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 (HE) verletzt sein

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

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

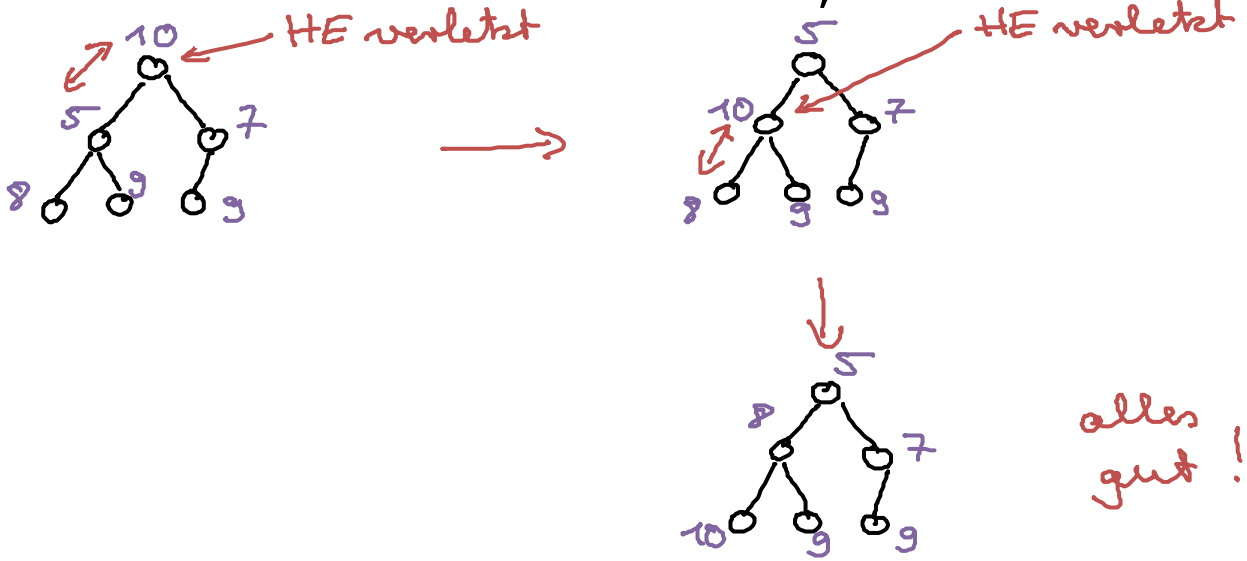
    repairHeapDownwards

    repairHeapUpwards
  - Siehe die nächsten drei Folien ...

Reparieren der Heapeigenschaft 2/4

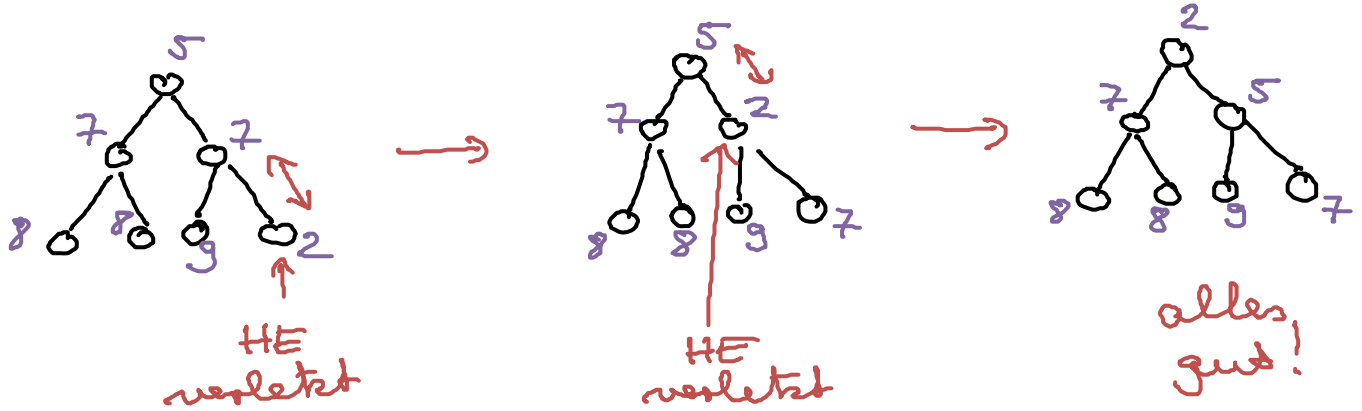
UNI FREIBURG

- Methode repairHeapDownwards
  - Knoten mit dem Kind tauschen, das den kleineren Key von den beiden Kindern hat
  - Jetzt ist bei diesem Kind evtl. die HE verletzt
  - Wenn, dann Key > der von den Kindern ... warum?
  - In dem Fall einfach da dasselbe nochmal, usw.



Reparieren der Heapeigenschaft 3/4

- Methode repairHeapUpwards
  - Knoten mit dem Elternknoten tauschen
  - Jetzt ist bei dem Elternknoten evtl. die HE verletzt
  - Wenn, dann Key < der von dessen Elternkn. ... warum?
  - 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 !

Komplexität 1/2

UNI FREIBURG

- Wiederholung Vorlesung 1
  - Ein vollständiger binärer Baum (bis evtl. "unten rechts") mit n Elementen hat Tiefe O(log n)
  - D.h. 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)
- Kosten (Laufzeit) für unsere diversen Methoden
  - Für repairHeapDownwards und repairHeapUpwards daher O(log n)
  - Für insert, deleteMin, changeKey, remove daher ebenfalls O(log n)
  - Für getMin offensichtlich O(1)

Komplexität 2/2

UNI FREIBURG

- Geht es noch besser?
  - Ja, mit sogenannten Fibonacci Heaps bekommt man

    getMin in Zeit O(1)

    insert in Zeit O(1)

    descreaseKey in amortisierter Zeit O(1)

    deleteMin in amortisierter Zeit O(log n)

    amortisiert = durchschnittlich ... nächste Vorlesung
  - In der Praxis ist der binäre heap aufgrund seiner Einfachheit aber schwer zu schlagen ... vor allem wenn die Anzahl der Elemente nicht riesig ist
  - Beachte: für $n = 2^{10} \approx 1.000$ ist $\log_2 n$ nur 10

    und selbst für $n = 2^{20} \approx 1.000.000$ ist $\log_2 n$ nur 20

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