Algorithmen und Datenstrukturen (für ESE) WS 2011 / 2012

Vorlesung 5, Dienstag, 29. November 2011

(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 4. Übungsblatt
  - Schauen Sie in's Forum
  - Nochmal die bottom line von universellem Hashing
- Prioritätswarteschlangen (Englisch: priority queues)
  - Ebenfalls eine Datenstruktur, die man sehr häufig braucht
  - Wir werden ein Beispiel sehen, wo man sie braucht
  - Und dann erklären, wie man selber eine bauen kann
  - Die Übungsaufgabe ist es dann, auf der Grundlage dieser Erklärungen, eine Klasse PriorityQueue zu schreiben
  - Am Ende noch kurz etwas zur C++ std::priority_queue und zur java.util.PriorityQueue

Ihre Erfahrungen mit dem 4. Ü-Blatt

- Zusammenfassung von Ihrem Feedback Stand 29.11 15:22
  - Universelles Hashing war nicht leicht zu verstehen
  - Gerüst für die Java Klasse war sehr hilfreich, bitte weiter so
  - Aufgabe 2 hat gedauert bis man die verstanden hat
    - insbesondere: was sollte zufällig sein und was nicht
  - Zeitaufwand für die meisten 4 – 8 Stunden
  - Vorlesung war sehr hilfreich für das Übungsblatt
  - Vorlesung war gar nicht hilfreich für das Übungsblatt
  - Test habe die meiste Zeit gekostet ... aber sinnvoll
  - Programmieren macht mehr Spaß und lehrreicher als Beweise
    - "Leider kommen jetzt wohl wieder mehr Beweise ..."
  - Es geht von Übung zu Übung besser!
  - "Viele Grüße"
  - Warum Punktabzug wenn in den erfahrungen.txt nichts steht?

Ihre Erfahrungen mit dem 4. Ü-Blatt

UNI FREIBURG

- Technische Probleme / Fragen:
  - Probleme mit Integer Overflow haben viel Zeit gekostet
    - **Problem**: die Produkte in PB1, PB2, PB3 können größer werden als der größte Integer, was insbesondere bei PB1 und PB2 zu Fehlern führt
      - **Lösung 1:** Mit BigInteger rechnen
      - **Lösung 2:** Universum kleiner machen
    - Dazu eine Korrektur: für die Hashfunktion PB1 muss man a und b zufällig aus {0, ..., p-1} wählen
    - Es reicht **nicht** zufällig aus {0, ..., N-1} für N < p, wie ich fälschlicherweise erklärt hatte
    - Gibt aber natürlich keinen Punktabzug dafür
  - Unterschied zwischen hash map und hash table?
  - Was tun wenn man vorher die Größe nicht weiß?

Schauen Sie in's Forum

- 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

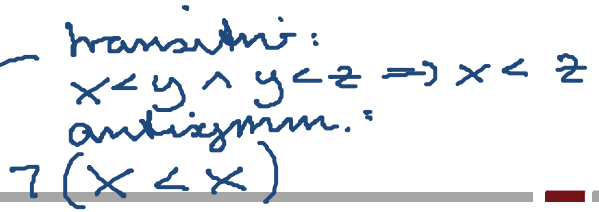
- Nochmal die "bottom line"
  - Keine Hashfunktion ist gut für alle Schlüsselmengen
    - das kann gar nicht gehen, weil ein großes Universum auf einen kleinen Bereich abgebildet wird
  - Für zufällige Schlüsselmengen tun es auch einfache Hashfunktionen wie $h(x) = x \bmod m$
    - dann sorgen die zufälligen 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

- Wenn man eine schlechte Hashfunktion erwischt
  - ... was einem auch mit universellem Hashing passieren kann, ja unvermeidlich ist, siehe Folie vorher
  - Man kann es aber leicht feststellen, nämlich wenn auf einer Position viel mehr Elemente stehen als im Idealfall der Gleichverteilung stehen sollten
  - Dann macht man einen sogenannten **"Rehash"**
    - Dabei wird eine neue Hashfunktion zufällig gewählt und **alle** Elemente von der alten in eine neue Hashtabelle kopiert
    - Weil bei universellem Hashing die meisten Hashfkt. für eine feste Schlüsselmenge gut sind, muss man das nicht sehr oft machen und die Kosten sind im Durchschnitt gering

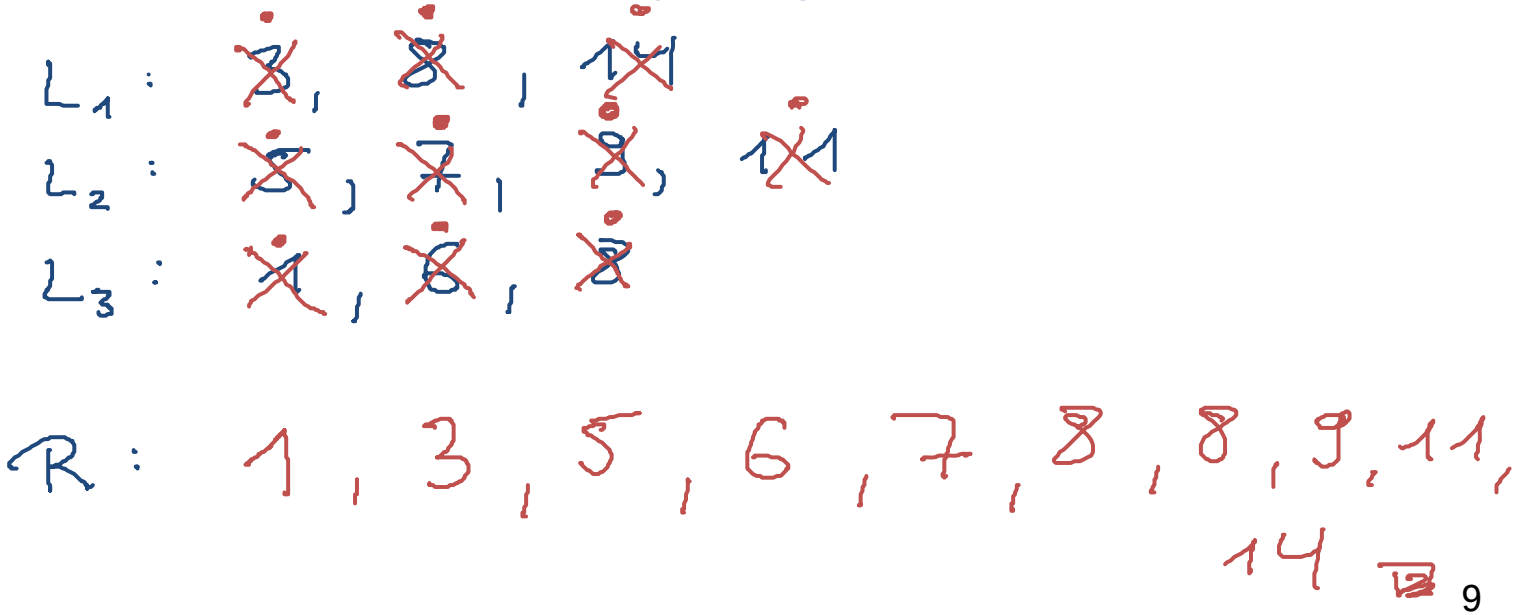
Prioritätswarteschlangen



- Definition
  - Eine **Prioritätswarteschlange** (PW) speichert eine Menge von (key, value) Paaren (wie ein assoziatives Array auch)
  - Es gibt eine transitive, antisymm. Ordnung **<** auf den Keys
    - Bei uns sind die Keys immer Zahlen mit dem normalen **<**
  - Die PW unterstützt auf dieser Menge folgende Operationen
    - getMin(): liefert das Paar mit dem kleinsten Key
    - deleteMin(): entferne das Paar mit dem kleinsten Key
    - insert(key, value): füge das gegebene Paar ein
  - Bemerkung: mehrere Paare mit demselben Key möglich; gibt es mehrere mit dem kleinsten Key, gibt getMin irgendeins davon zurück (und deleteMin löscht eben das)

PWs — Anwendungen 1/2

- Wo braucht man PWs? Ein Beispiel
  - Berechnung der Vereinigungsmenge von k sortierten Listen (sogenannter multi-way merge oder k-way merge)
  - Die implementieren wir jetzt mit unserer eigenen PriorityQueue (die Sie für das 5. Übungsblatt implementieren sollen)
  - Vorab die Grundidee des k-way merge:

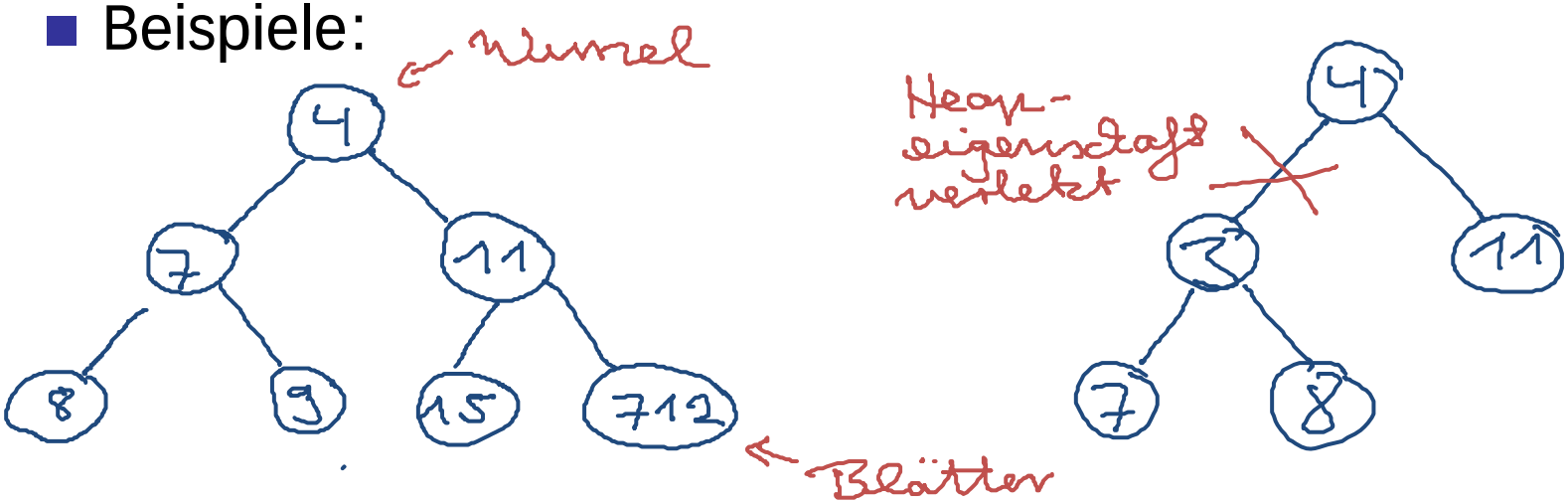


PWs — Anwendungen 2/2

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

PWs — Realisierung 1/4

- Grundidee
  - Elemente in einem **binären Baum** speichern
    - Ein binärer Baum ist ein Baum, bei dem jeder Knoten höchtens zwei Kinder hat
  - Es gilt die sogenannte **Heap-Eigenschaft**
    - Der Key eines Knotens ist ≤ die Keys seiner Kinder
  - Entsprechend nennt man das ganze **binären Heap**
- Beispiele:

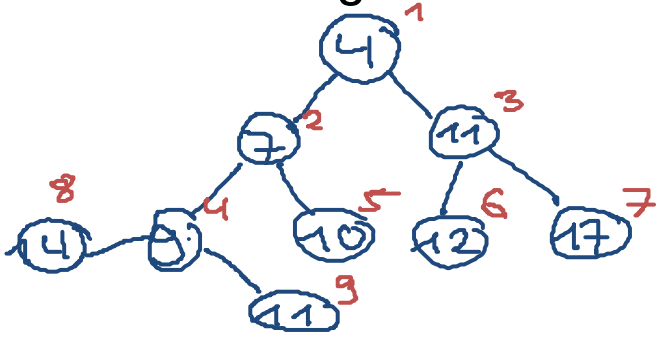


PWs — Realisierung 2/4

- Wie speichert man einen binären Heap
  - Wir numerieren die Knoten von oben nach unten und links nach rechts durch, beginnend mit **1** (wir sehen gleich warum!)
  - Dann sind die Kinder von Knoten i die Knoten 2i und 2i + 1
  - Und der Elternknoten von einem Knoten i ist floor(i/2)
  - Wir können die Paare dann einfach in einem normalen Feld speichern:

    ArrayList<KeyValuePair> heap; // Java.

    std::vector<KeyValuePair> heap; // C++.
  - Zugriff auf den Knoten i einfach mit heap.get(i) bzw. heap[i]

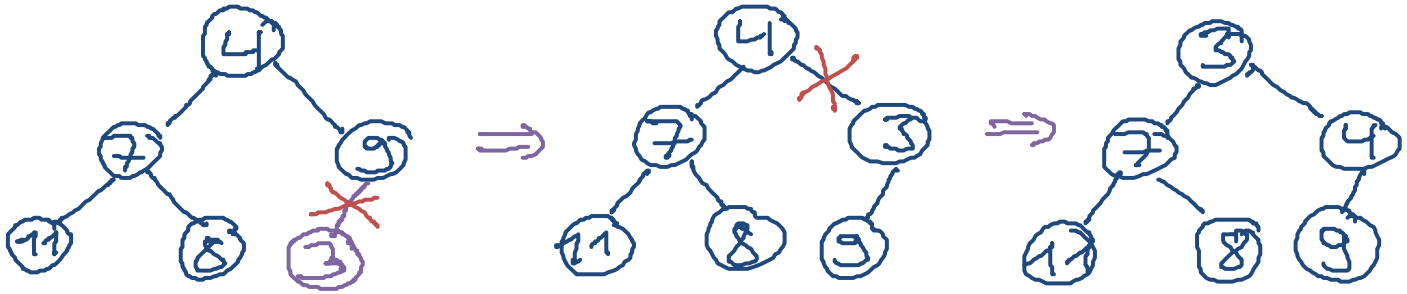


PWs — Realisierung 3/4

- Einfügen eines Elementes
  - Beim insert von einem neuen keyValuePair, fügen wir es einfach am Ende des Arrays ein

    heap.add(keyValuePair); // Java.

    heap.push_back(keyValuePair); // C++.
  - Danach ist die Heapeigenschaft wahrscheinlich verletzt
  - Wie stellen wir Sie dann (effizient) wieder her?
  - Wir folgen dem Pfad vom neu eingefügten Blatt zur Wurzel und "vertauschen" bis alles wieder stimmt:

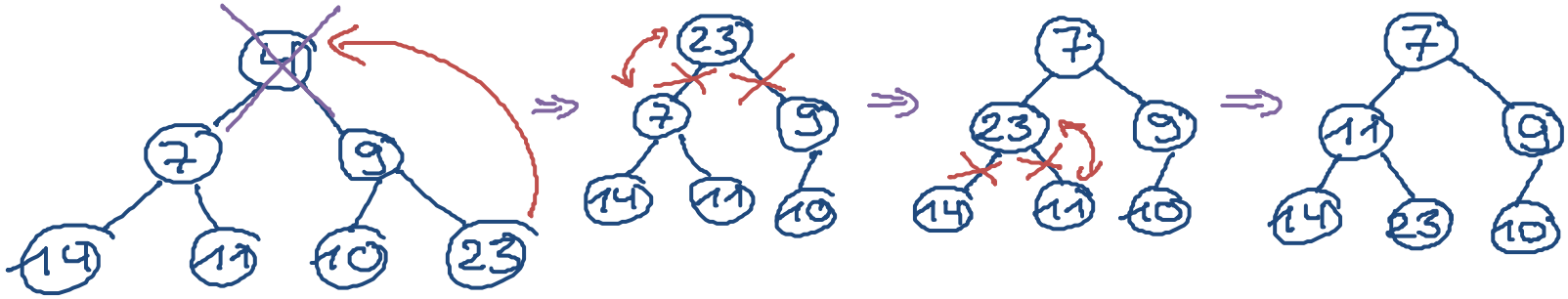


PWs — Realisierung 4/4

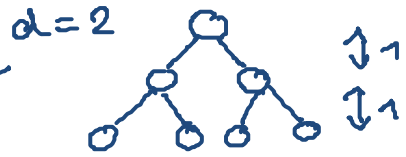
- Löschen eines Elementes
  - Nach deleteMin setzen wir einfach das letzte Element an die erste Stelle

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

    heap[1] = heap.back(); heap.pop_back(); // C++.
  - Auch danach ist die Heapeigenschaft wahrscheinlich verletzt
  - Um sie wiederherzustellen "vertauschen" wir wieder geeignet
  - Diesmal auf einem Pfad von der Wurzel nach unten:



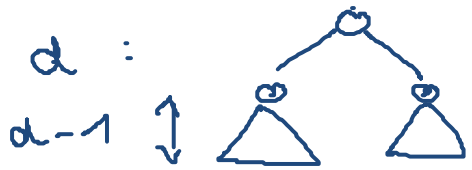
PWs — Komplexität 1/2



- **Satz:** ein vollständiger binärer Baum der Tiefe d …
  - … hat $2^{d+1} - 1$ Knoten
  - Tiefe = Anzahl Kanten auf Pfad von der Wurzel zu einem Blatt
  - Zum Beispiel: Tiefe 2 → 7 Knoten
  - Beweis über vollständige Induktion:

$d = 0$: #Knoten $= 1 = 2^{0+1} - 1$ ☐

$d-1 \to d$:



#Knoten für Tiefe d

$2 \cdot (2^d - 1) + 1$ (IV)

$= 2^{d+1} - 2 + 1$

$= 2^{d+1} - 1$

- **Korollar:** Sei n die Anzahl der Elemente in der PW …
  - … dann ist die Anzahl der Elemente auf einem Pfad von einem beliebigen Element zur Wurzel O(log n)

PWs — Komplexität 2/2

- Damit sind die Kosten unserer PW-Operationen:
  - getMin : Kosten O(1)
    - einfach das Element von heap an Stelle 1 zurückgeben
  - deleteMin : Kosten O(log n)
    - Vertauschungen entlang eines Pfades Wurzel → Blatt
  - insert : Kosten O(log n)
    - Vertauschungen entlang eines Pfades Blatt → Wurzel
- Bemerkung: es geht noch effizienter
  - getMin und insert beide in O(1), nur deleteMin in O(log n)
  - mit sogenannten **Fibonacci Heaps**
  - die sind aber deutlich komplizierter … und durch die komplexere Implementierung in der Praxis nur bei bestimmten Anwendungen auch wirklich besser

Prioritätswarteschlangen in Java

UNI FREIBURG

- Dafür braucht man 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
  - Die Operationen heißen
    - add = Einfügen eines Elementes
    - peek = liefert kleinstes Element, aber belässt es in der PW
    - poll = liefert kleinstes Element und entfernt es aus der PW
  - Zusätzlich gibt es dort noch die (Nicht-Standard) Operation
    - remove = entferne das gegebene Element aus der PW

Prioritätswarteschlangen in der STL

UNI FREIBURG

- Dafür braucht man #include <queue>
  - Element-Typ unterscheidet nicht zwischen Key und Value

    std::priority_queue<T> pq;
  - Achtung: es wird die Ordnung **>** auf T genommen, nicht **<**
  - Beliebige Vergleichsfunktion wie bei std::sort
  - Die Operationen heißen
    - top = liefert das größte Element, ohne es zu löschen
    - pop = entfernt das größte Element
    - push = fügt ein Element hinzu
    - Es gibt (aus Effizienzgründen) **keine** Methode um ein beliebiges Element zu entfernen oder zu verändern!

Literatur / Links

- 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