Algorithmen und Datenstrukturen (für ESE) WS 2011 / 2012

Vorlesung 9, Dienstag, 10. Januar 2012

(Verkettete Listen, Binäre Suchbäume)

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 8. Übungsblatt
  - Treffen mit Ihrem Tutor / Ihrer Tutorin
- Binäre Suchbäume (binary search trees)
  - Was ist das?
  - Wofür braucht man das?
  - Übungsaufgabe: schreiben Sie eine Klasse BinarySearchTree

Ihre Erfahrungen mit dem Ü8 (Cache)

UNI FREIBURG

- Zusammenfassung von Ihrem Feedback **Stand 10.1 15:20**
  - Thema (Cache) fanden die meisten interessant / lehrreich
  - Interessant, dass bessere Laufzeit trotz mehr Operationen
  - Viel Unsicherheit was genau gefordert war
  - Manche sehr lange an der Knobelaufgabe (2c) gesessen
  - War das ÜB für drei Wochen gedacht? Wir haben doch Ferien
  - Lieber ganz präzise Aufgaben
  - Erklärungen etwas zu oberflächlich zur Beantwortung des ÜBs
  - Aussage, dass Studenten für Korrektheit mitverantwortlich, ist seltsam, wir sind doch nicht die Professoren
  - Vorlagen in Java → wer C++ benutzt hat mehr Aufwand
  - Sind die erfahrungen.txt eigentlich anonym?
  - 100ns für RAM und 1ns für Cache, stimmt das wirklich?

Motivation: Sortierte Folgen

UNI FREIBURG

- Problem
  - Wir wollen wieder (key, value) Paare / Elemente verwalten
  - Wir haben wieder eine Ordnung **<** auf den Keys
  - Diesmal wollen wir folgende Operationen unterstützen
    - insert(key, value): füge das gegebene Paar ein
    - remove(key): entferne das Paar mit dem gegebenen Key
    - lookup(key): finde das Element mit dem gegebenen Key; falls es das nicht gibt, finde das Elemente mit dem kleinsten Key der > key ist
    - next / previous: für ein gegebenes Element, finde das mit dem nächstgrößeren / nächstkleineren Schlüssel; damit lässt sich insbesondere über alle Elemente iterieren

Wo braucht man das?

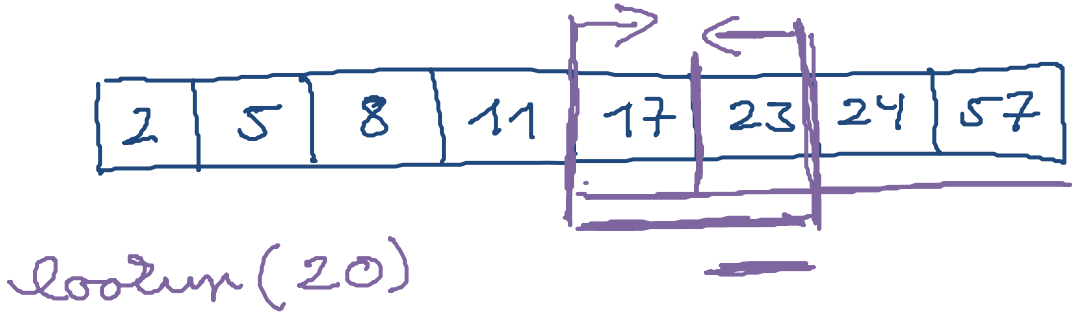
UNI FREIBURG

- Typisches Anwendungsbeispiel: Datenbanken
  - Eine große Menge von Records
  - Zum Beispiele Bücher, Produkte, Wohnungen, ...
  - Typische Suchanfrage: alle Wohnungen zwischen 400 und 600 Euro Monatsmiete
    - Ein sogenannter **range query**
    - Das bekommt man mit lookup und next
    - Man beachte: es ist dafür nicht wichtig, dass es eine Wohnung gibt, die **genau** 400 Euro kostet
  - Wenn man ein paar records hinzufügt oder alte löscht, will man nicht jedes Mal erst alles wieder neu sortieren

Lösung 1 (schlecht): Einfache Arrays

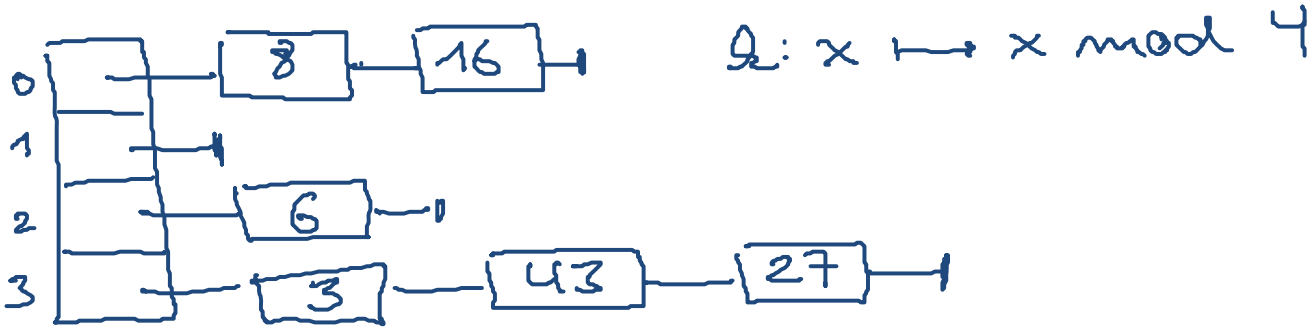
- Mit einem einfachen Array bekommen wir
  - lookup in Zeit $O(\log n)$
    - das geht mit **binärer Suche**, siehe unten
  - next und previous in Zeit $O(1)$
    - klar, sie stehen ja direkt nebeneinander
  - insert und remove in Zeit bis zu $\Theta(n)$
    - bis zu $\Theta(n)$ Elemente müssen umkopiert werden

viel mals mal wieder nur die Keys Dein!



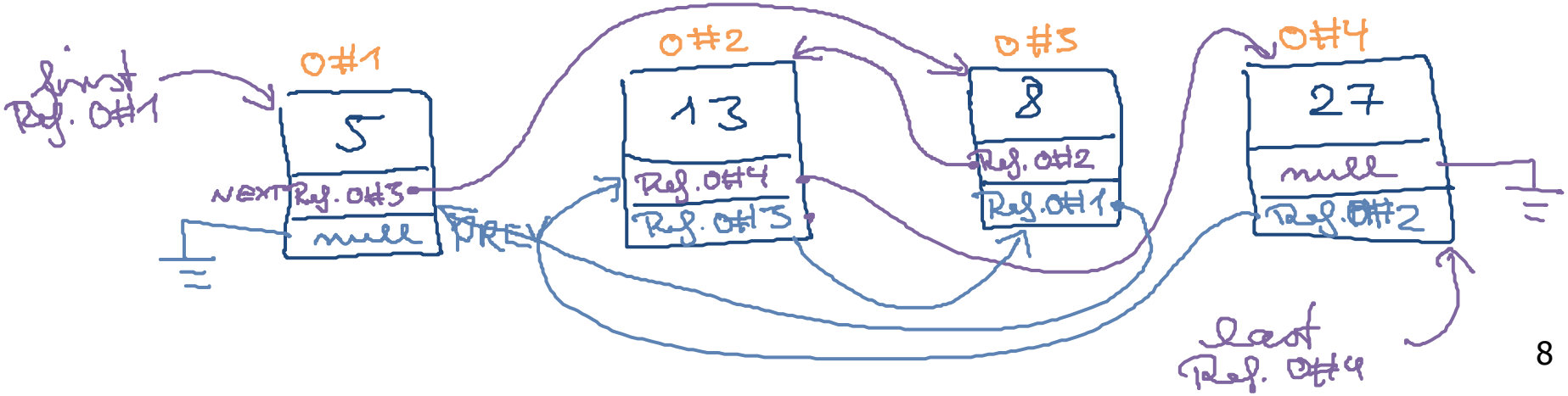
Lösung 2 (schlecht): Hashtabellen

- Mit einer Hashtabelle bekommt man
  - insert und remove in erwarteter Zeit O(1)
    - bei genügend großer Hashtabelle und guter Hashfunktion
  - lookup in erwarteter Zeit O(1)
    - aber nur wenn es ein Element mit **exakt** dem gegebenen Key gibt, sonst bekommt man gar nichts
  - next und previous in Zeit bis zu Θ(n)
    - die Reihenfolge, in der die Elemente in einer Hashtabelle stehen hat nichts mit der Reihenfolge der Keys zu tun!



Lösung 3 (besser): Verkettete Listen

- Mit einer doppelt verketteten Liste bekommt man
  - next und previous in Zeit O(1)
    - jedes Element hat einen Zeiger zum Vorgänger / Nachfolger
  - insert und remove in Zeit O(1)
    - es müssen nur konstant viele Zeiger umgesetzt werden
  - lookup in Zeit bis zu Θ(n)
    - die Elemente stehen jetzt nicht mehr sortiert in einem Feld; man muss sie sich im schlechtesten Fall alle anschauen



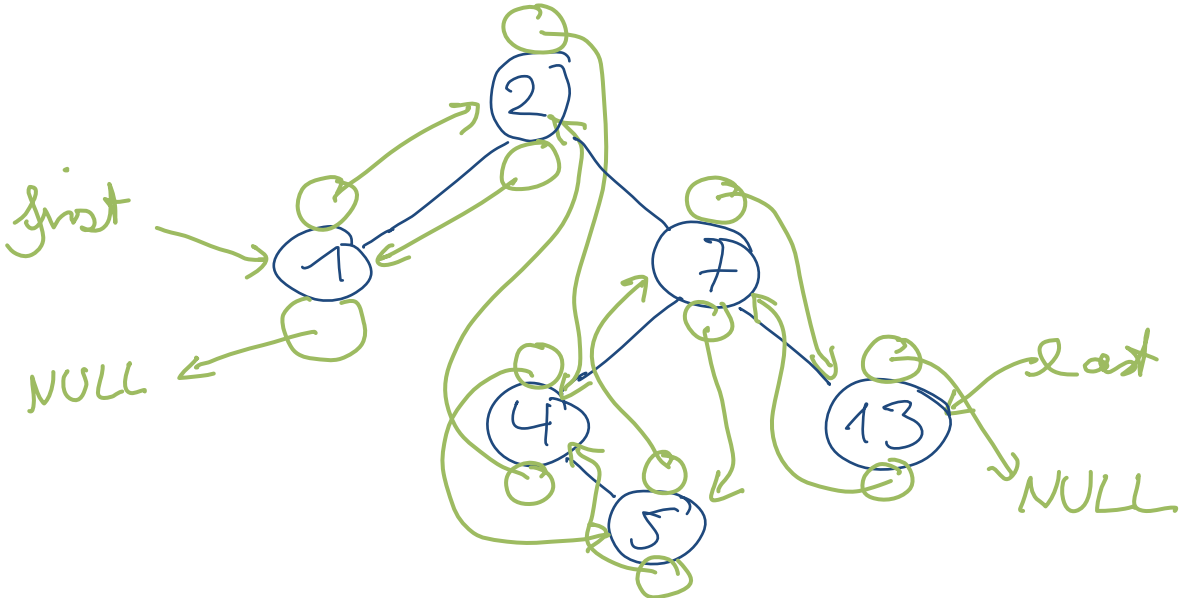
Lösung 4 (gut): Suchbäume

UNI FREIBURG

- Mit einem geeigneten Suchbaum bekommt man
  - next und previous in Zeit O(1)
    - entsprechende Zeiger wie bei der verketteten Liste
  - insert und remove in Zeit O(1)
    - ebenfalls wie bei der verketteten Liste
  - lookup in Zeit O(log n)
    - eine Baumstruktur hilft jetzt beim effizienten Suchen

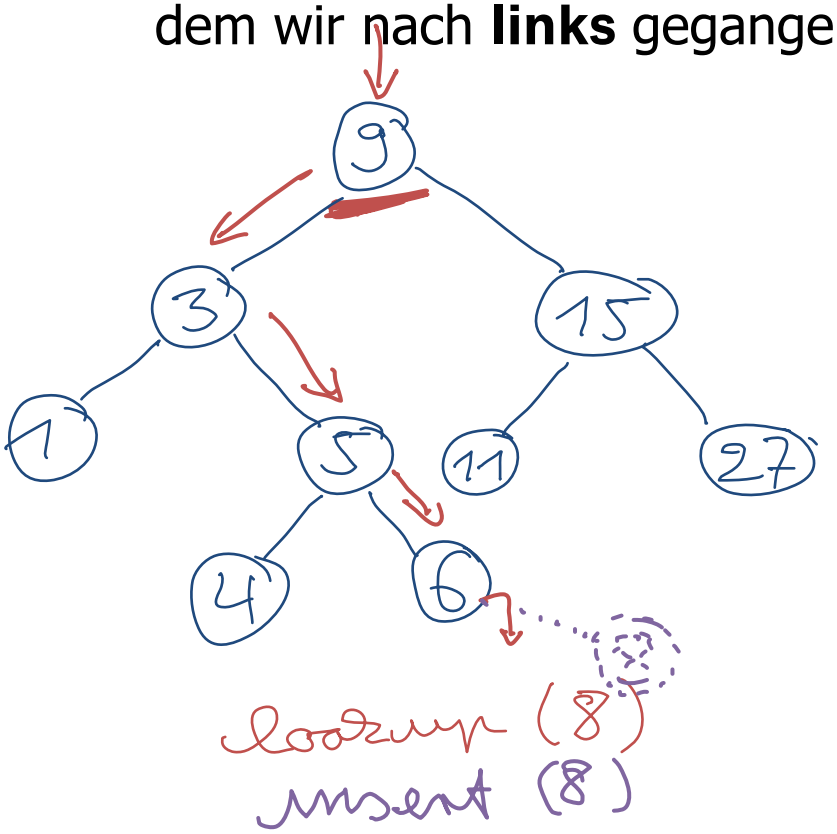
Binäre Suchbäume — Idee

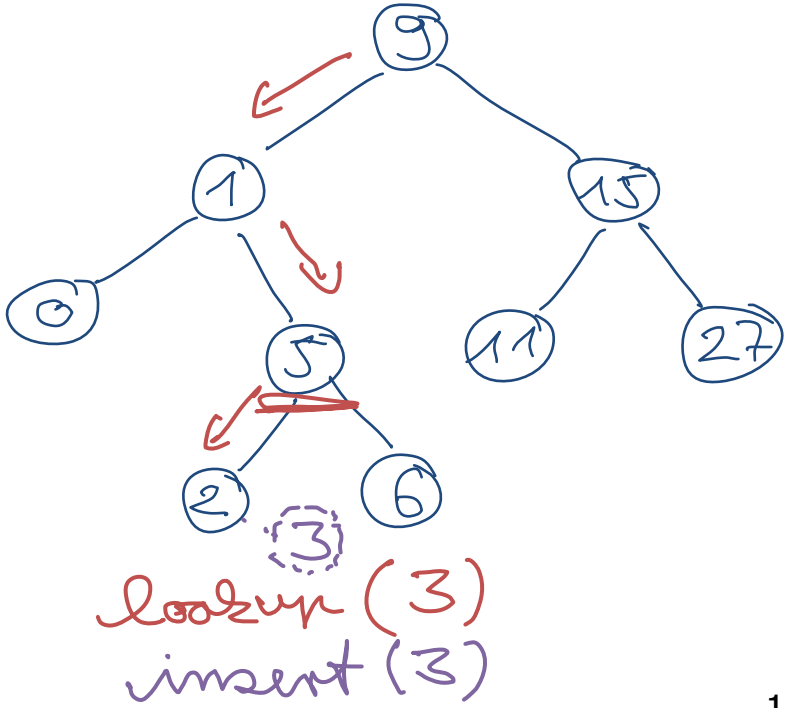
- Anordung ähnlich wie bei der Prioritätswarteschlange
  - Aber jetzt ganz sortiert!
  - Für jeden Knoten gilt: alle Elemente im linken Unterbaum haben einen kleineren Key + alle Elemente im rechten Unterbaum haben einen größeren Key
  - Und **gleichzeitig** eine doppelt verkettete Liste der Elemente
    - die können Sie für's Übungsblatt aber weglassen!



Binäre Suchbäume — Lookup

- Wir suchen einfach von der Wurzel abwärts
  - und gehen je nach Key links oder rechts
  - und merken uns dabei immer den **letzten** Knoten, an dem wir nach **links** gegangen sind **… warum?**





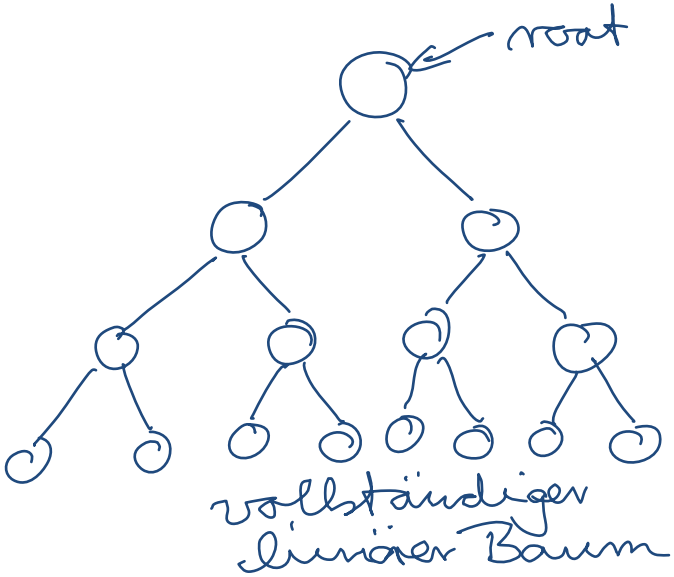
Binäre Suchbäume — Einfügen

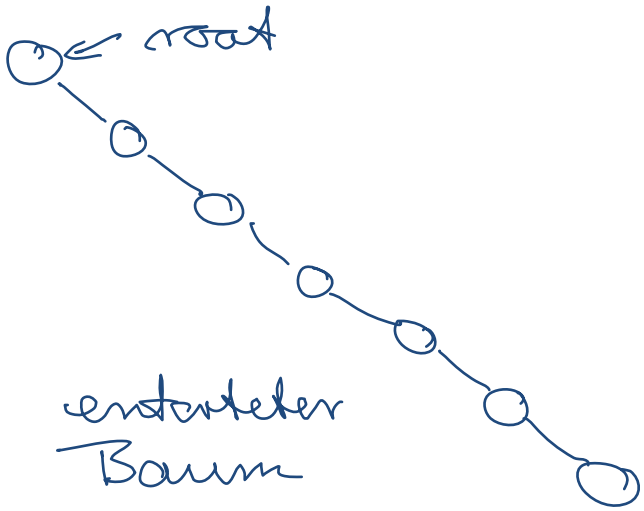
UNI FREIBURG

- Wir suchen erstmal den gegebenen Key
  - Wenn wir ihn gefunden haben, überschreiben wir einfach das Value an dem Knoten
  - Sonst fügen wir an geeigneter Stelle einen neuen Knoten ein
    - Wenn es den Key im Baum noch nicht gab, kommen wir immer bis an einen Knoten v mit max. 1 Kind ... **warum?**
    - Wenn der neue Key kleiner ist als der Key von v, dann hat v kein linkes Kind und das neue Element wird linkes Kind
    - Wenn der neue Key größer ist, entsprechend rechtes Kind

Binäre Suchbäume — Komplexität

- Wie lange dauern insert und lookup ?
  - Bis zu Zeit Θ(d), wobei d die Tiefe des Baumes ist
    = die größte Tiefe eines Blattes
  - Im besten Fall ist das Θ(log n), im schlechtesten Fall aber Θ(n), wobei n die Anzahl der Knoten im Baum ist
  - Wenn man **immer** Θ(log n) will, muss man den Baum gelegentlich **rebalancieren** → nächste Vorlesung





Literatur / Links

- Suchbäume
  - In Mehlhorn/Sanders:

    7 Sorted Sequences
  - In Cormen/Leiserson/Rivest

    13 Binary Search Trees
  - In Wikipedia

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

    http://en.wikipedia.org/wiki/Binary_search_tree
  - In Java / C++
    - die java.util.TreeMap und die std::map sind typischerweise mit (balancierten) Suchbäumen implementiert