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

Vorlesung 10, Dienstag 8. Januar 2013
(Verkettete Listen, Binäre Suchbäume)

Björn Buchhold
i.V. für Prof. Dr. Hannah Bast
Lehrstuhl für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

- Organisatorisches
  - Ihre Erfahrungen mit dem Ü9 (Performance Tuning)
- Binäre Suchbäume (binary search trees)
  - Was ist das?
  - Wofür braucht man die?
  - Übungsblatt 10: Einfache Implementierung (nur insert und lookup, kein remove) + ein paar einfache Laufzeittests

Erfahrungen mit dem Ü9 (Perf. Tuning)

- Zusammenfassung / Auszüge   Stand 8. Januar 15:30
  - Vorlesung / Thema war sehr interessant
  - Meiste Zeit ging für Probieren / Tuning drauf ... so war's gedacht
  - Erst nicht klar, was ein HashSet ist ... dann auf Forum geklärt
  - In Java, mit int[][] viel schneller als Standard-Implementierung
  - Mit ArrayList<ArrayList<Integer>> dagegen viel langsamer
  - Tests wirklich hilfreich, weil wie angekündigt nach manchen "Optimierungen" der Code nicht mehr funktioniert hat
  - Verhältnisse StaticHashSet / Standardimpl. in der Ergebnis-tabelle teilweise arg niedrig

Motivation: Sortierte Folgen

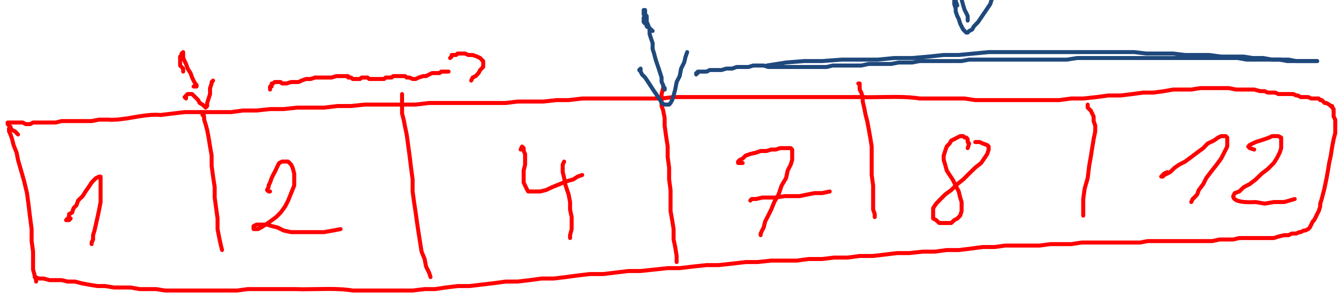
- Problem
  - Wir wollen wieder (key, value) Paare / Elemente verwalten
  - Wir haben wieder eine Ordnung **<** auf den Keys
  - Diesmal wollen wir folgende Operationen unterstützen
    - insert(key, value): füge das gegebene Paar ein
    - remove(key): entferne das Paar mit dem gegebenen Key
    - lookup(key): 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?

- 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 (nicht gut): Einfache Arrays

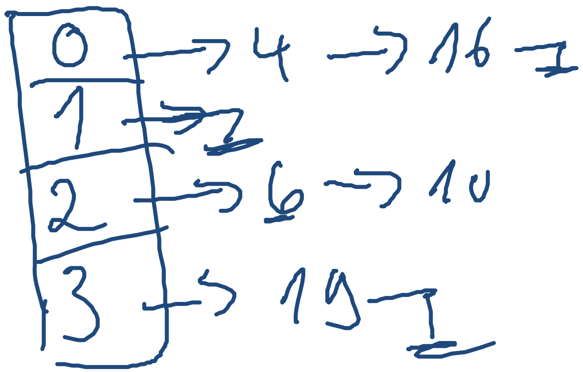
- Mit einem einfachen Array bekommen wir
  - lookup in Zeit O(log n)
    - das geht mit **binärer Suche**, siehe unten
  - next und previous in Zeit O(1)
    - klar, sie stehen ja direkt nebeneinander
  - insert und remove in Zeit bis zu Θ(n)
    - bis zu Θ(n) Elemente müssen umkopiert werden



lookup(8)

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!

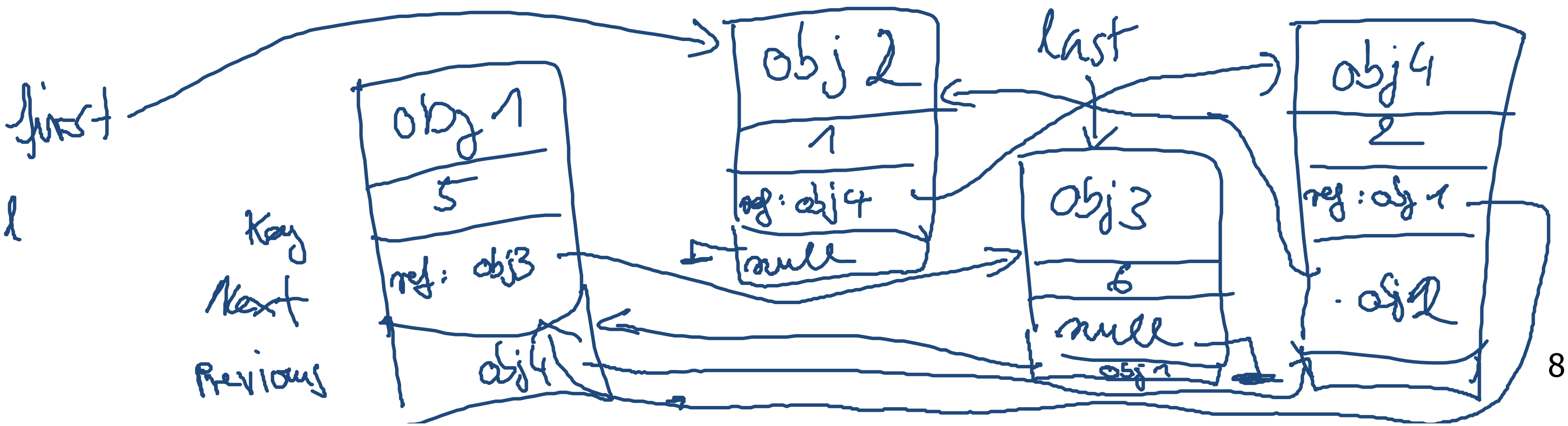


$k = n \bmod 4$

lookup(6)
lookup(7)

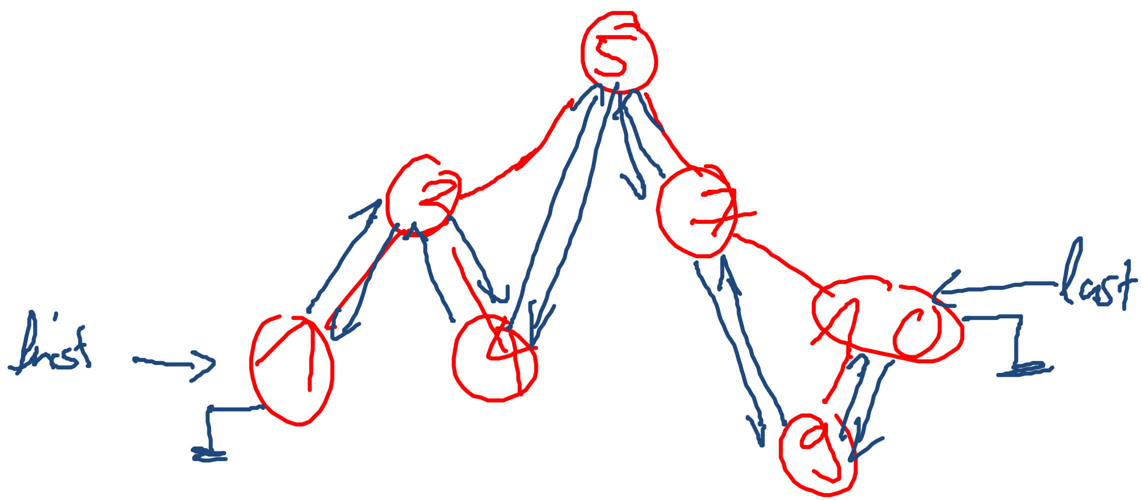
Lösung 3 (nicht gut): Verkettete Listen

- 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

- 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

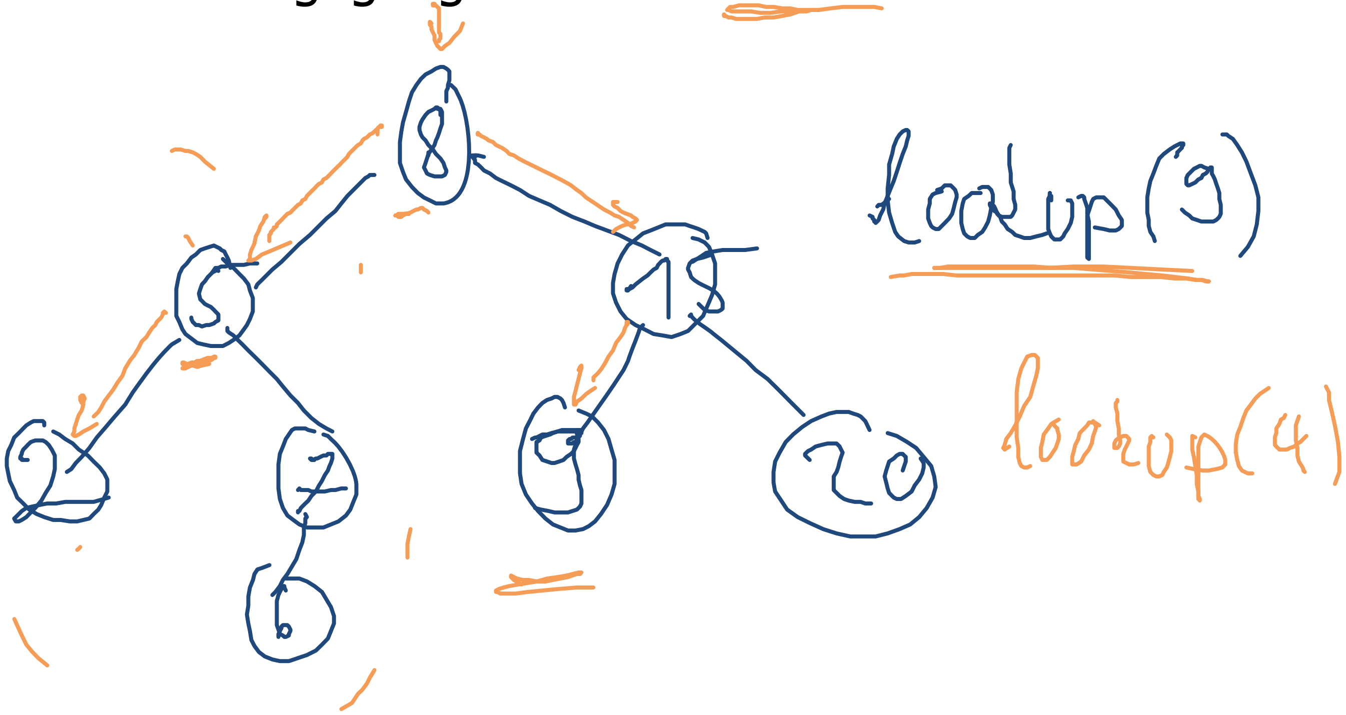


Binäre Suchbäume — Idee

- 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

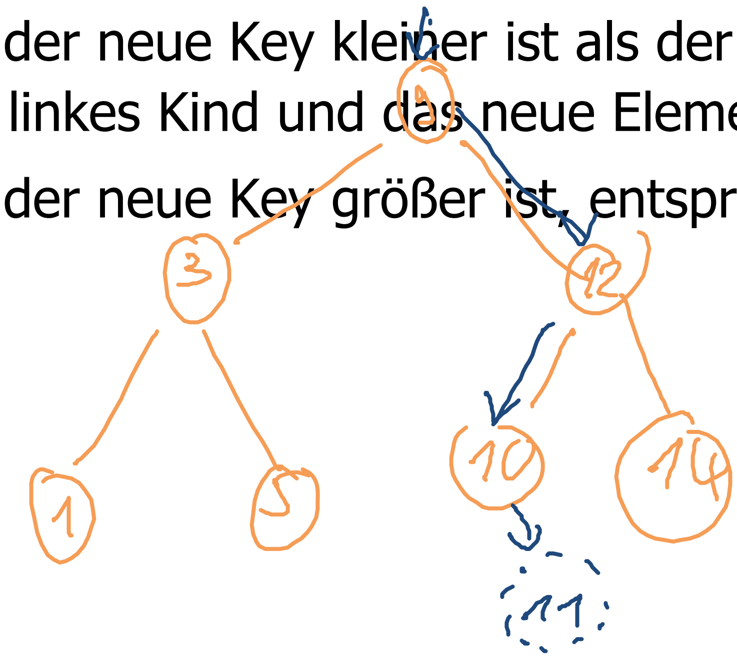
- Wir suchen einfach von der Wurzel abwärts
  - und gehen je nach Key links oder rechts
  - und merken uns dabei immer den **letzten** Knoten, an dem wir nach **links** gegangen sind ... warum?



Binäre Suchbäume — Einfügen

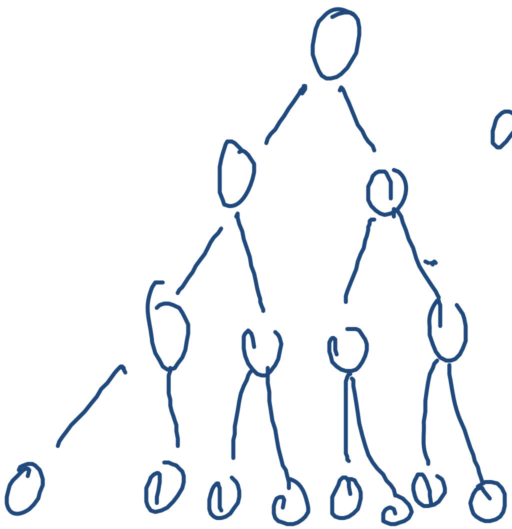
- Wir suchen erstmal den gegebenen Key
  - Wenn wir ihn gefunden haben, überschreiben wir einfach das Value an dem Knoten
  - Sonst fügen wir an geeigneter Stelle einen neuen Knoten ein
    - Wenn es den Key im Baum noch nicht gab, kommen wir immer bis an einen Knoten v mit max. 1 Kind ... warum?
    - 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

insert(11)

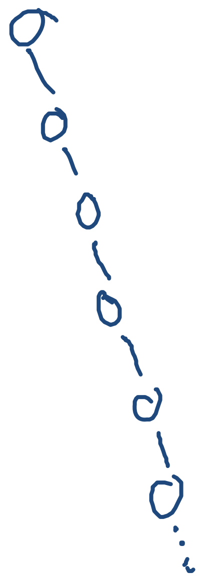


Binäre Suchbäume — Komplexität

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



$d = O(\log n)$



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