Informatik II: Algorithmen und Datenstrukturen SS 2013

Vorlesung 8a, Dienstag, 11. Juni 2013
(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 Ü7 (Listen, Blockoperationen)
- Binäre Suchbäume (binary search trees)
  - Wofür man das braucht
  - Wie man es macht
  - Vergleich mit den Datenstrukturen, die wir bisher schon kennengelernt haben
  - Ü8, Aufgabe 1: Implementieren Sie einen BinarySearchTree

Erfahrungen mit dem Ü7 (Listen, Blockop.)

UNI FREIBURG

- Zusammenfassung / Auszüge   Stand 11. Juni 15:30
  - Einfaches Blatt + zeitlich gut machbar
  - LinkedList war schon aus Info I bekannt
  - Mehr Beispiele für Analyse Blockoperationen
  - Wie Rekursionsgleichung exakt auflösen ... nächste Folie
  - Gut die Auswirkungen von Cache-Effekte zu verstehen
  - Wofür braucht man dann überhaupt noch LinkedList ?

    Für Einfügen an beliebiger Stelle in O(1) Zeit, siehe VL 7a Folie 11
  - Forum hilfreich ... oder auch nicht, je nachdem wer antwortet
  - Hinweis, dass man nicht erst am Montag Abend anfangen soll, erst am Montag Abend gelesen
  - 20h in Kopf-gegen-Wand-Gehirnanregungsübungen investiert

Musterlösung Ü7, Aufgabe 2

- Analyse Anzahl Blockoperationen von MergeSort

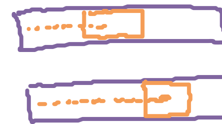
IO(n) = #Blöcke für MergeSort für Eingabegröße n

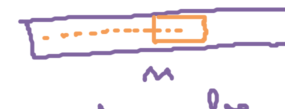
$$IO(n) \leq 2 \cdot IO(n/2) + A \cdot n/B$$

#Blöcke um zwei Felder der Gesamtgröße n zu mischen (mergen)

$$\leq 2 \cdot [2 \cdot IO(n/4) + A \cdot \tfrac{n}{2}/B] + A \cdot n/B$$

$$= 4 \cdot IO(n/4) + 2 \cdot A \cdot n/B$$





$$\vdots$$

$$\leq 2^k \cdot IO(n/2^k) + k \cdot A \cdot n/B$$

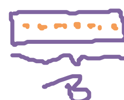
← das kann man formal mit vollständiger Induktion beweisen

$k = \log_2(n/B)$

$$= n/B \cdot IO(B) + \log_2(n/B) \cdot A \cdot n/B$$

$IO(B) = O(1)$!

$$2^{\log_2(n/B)} = n/B$$



Sortierte Folgen 1/2

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

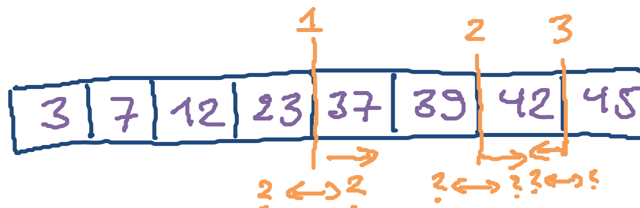
Sortierte Folgen 2/2

UNI FREIBURG

- Typisches Anwendungsbeispiel: Datenbanken
  - Eine große Menge von Einträgen (Engl.: 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 Einträge hinzufügt oder alte löscht, will man nicht jedes Mal erst alles wieder neu sortieren

Lösung 1: 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(40)
⇝ 42

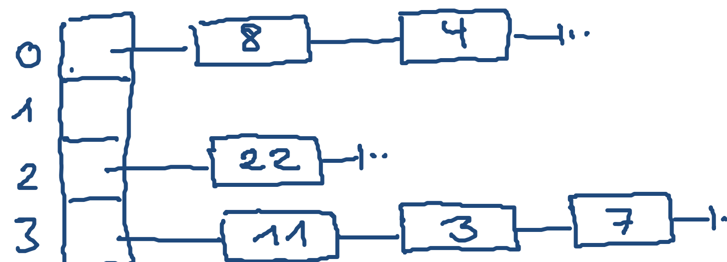
Lösung 2: Hashtabellen

ABER: umgekehrt kann man für sort. Folgen auch für das verwenden, wo man sonst Hashing benutzt
z.B. JAVA: HashMap vs. TreeMap

UNI FREIBURG

- 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!

$h(x) = x \bmod 4$



Lösung 3: Verkettete Listen

UNI FREIBURG

- 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 $\Theta(n)$
    - man könnte die Elemente sortiert halten, aber dann muss man beim Einfügen die richtige Stelle finden, dafür muss man sich aber im schlechtesten Fall alle Elemente anschauen

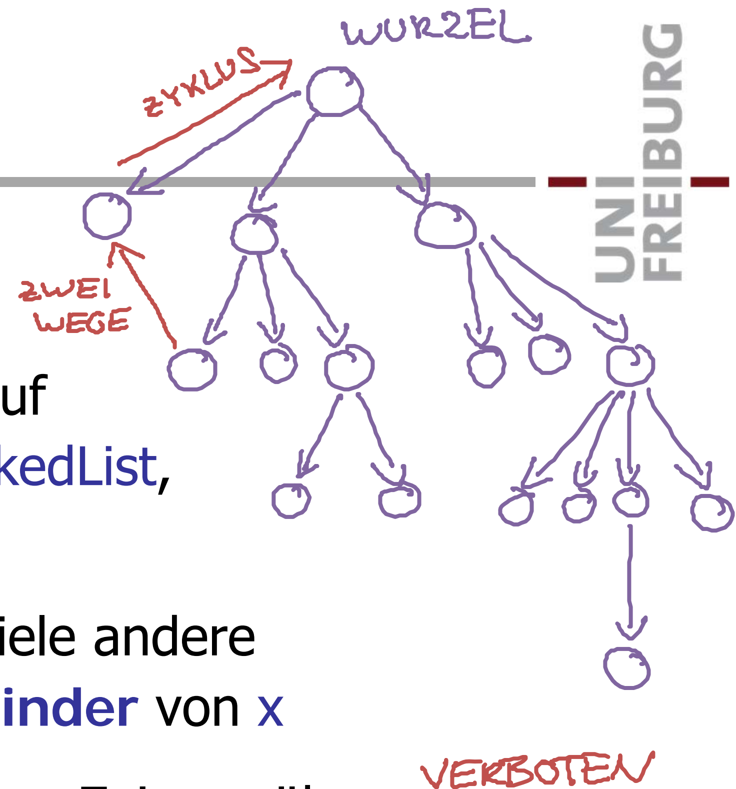
      Binäre Suche funktioniert nicht auf einer LinkedList !

Lösung 4: 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

Binärer Suchbaum 1/8



- Definition allgemeiner Baum
  - Besteht aus Elementen, die Zeiger auf andere Elemente haben, wie bei LinkedList, mit folgenden Unterschieden:

    Jedes Element x kann auf beliebig viele andere Elemente zeigen, die heißen dann **Kinder** von x

    Es gibt keinen Zyklus = eine Folge von Zeigern über die man von einem Element x wieder zu x kommt

    Es gibt ein **Wurzel**element, auf das niemand zeigt

    Von der Wurzel aus gibt es genau einen Weg, über die Zeiger zu jedem Element zu kommen

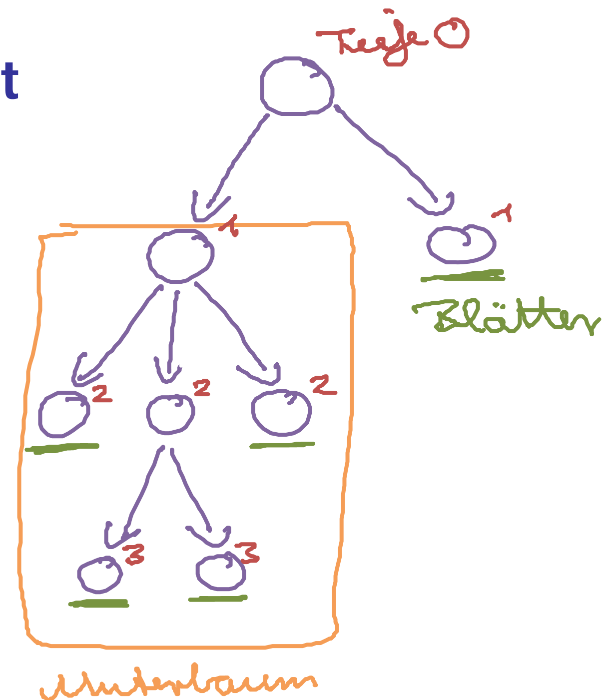
Binärer Suchbaum   2/8

UNI FREIBURG

Tiefe dieses Baumes = 3

- Terminologie allgemeiner Baum
  - Die Elemente nennt man auch **Knoten** (English: **node**)
  - Alle Elemente, die von einem Knoten x aus erreichbar sind, bilden wieder einen (Unter-)Baum
  - Einen Knoten ohne Kind nennt man **Blatt**

    die anderen nennt man **innere Knoten**
  - Die Anzahl Zeiger auf dem Weg von der Wurzel zu einem Knoten nennt man dessen **Tiefe**
  - Die Tiefe des Baumes ist die max. Tiefe eines Knotens

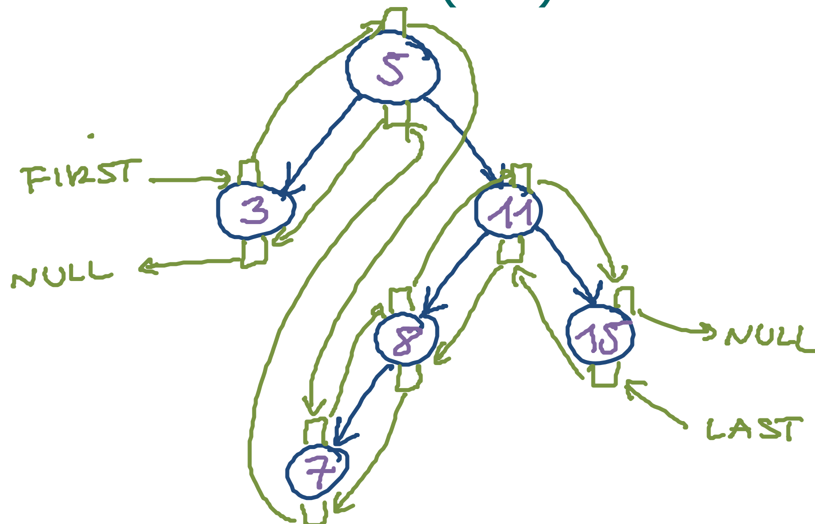


Binärer Suchbaum   3/8

- Binärer Suchbaum
  - Jeder Knoten hat **höchstens** zwei Kinder
  - 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

  Braucht man (nur) für next und previous in O(1) Zeit

bei genau zwei: vollständig linear



ich male nur die Keys hin

Baum nur in Bezug auf die blauen Zeiger

Binärer Suchbaum 4/8

UNI FREIBURG

- **Lookup(x)**
  - Von der Wurzel abwärts suchen, und an jeden Knoten node

    falls x == node.key ... gefunden!

    falls x < node.key ... nach links weiter suchen

    falls x > node.key ... nach rechts weiter suchen
  - Wenn man so an einem Blatt ankommt und x nicht gefunden hat, gibt es den Key nicht im Baum

    Dann ist der nächstgrößere Key an dem Knoten, bei dem man zum letzten Mal nach **links** gegangen ist

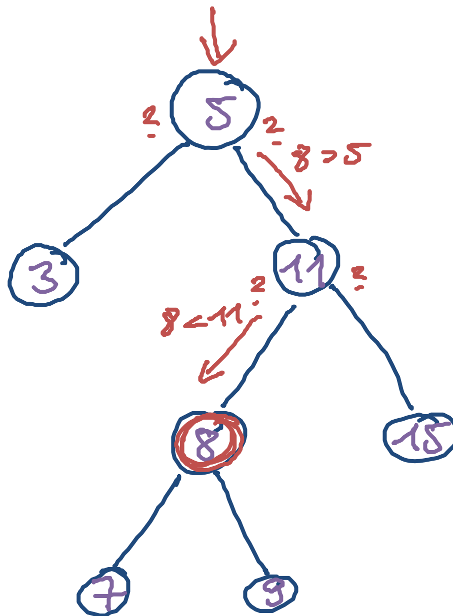
    Den muss man sich also beim Abwärtssuchen immer merken

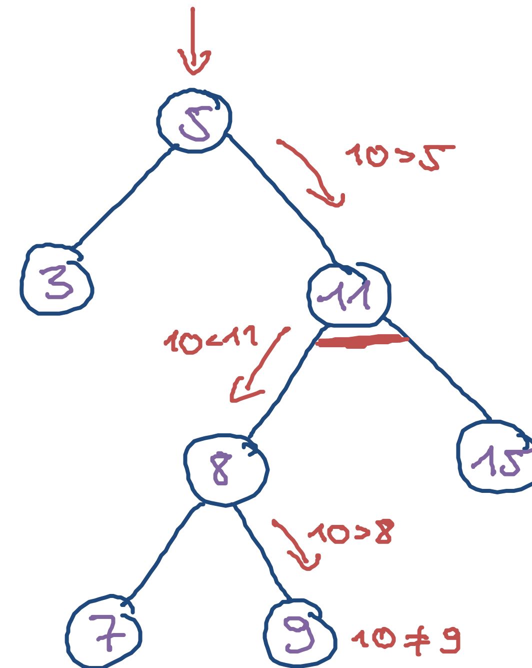
    wenn man nie nach links geht und x nicht findet, dann:
    x größer als alle Schlüssel im Baum

Binärer Suchbaum 5/8

UNI FREIBURG

- **Lookup(x)** … Beispiele:





lookup (8)
→ gefunden

lookup (10)
→ nicht gefunden
nächstgrößere = 11
= letzte Stelle, wo man nach links gegangen ist

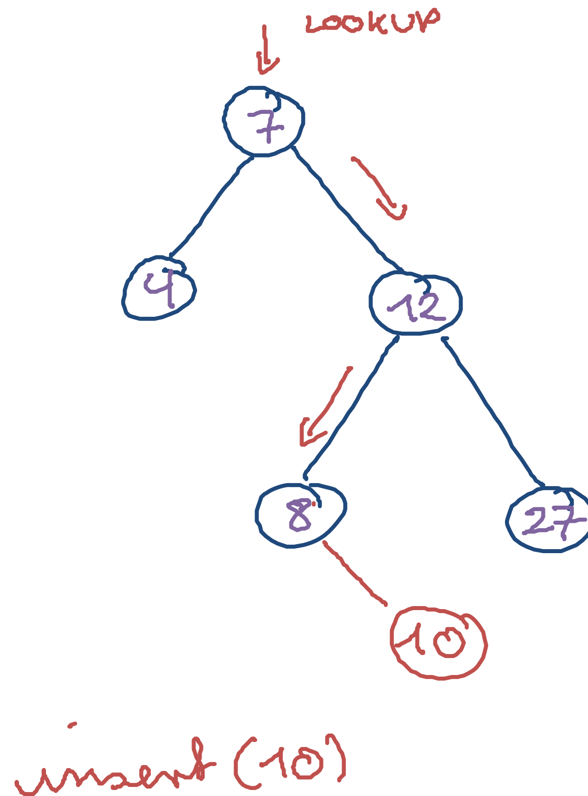
Binärer Suchbaum 6/8

UNI FREIBURG

- **Insert(x, value)**
  - Erst mal ein lookup(x)
  - Wenn es x im Baum schon gibt, überschreiben wir einfach das Value an dem Knoten
  - Sonst endet die Abwärtssuche von lookup(x) an einem node, der **höchstens** ein Kind hat und node.key != x
  - Falls x < node.key, dann hat node kein linkes Kind (sonst wäre die Suche da weiter gegangen) und das neue Element wird das linke Kind
  - Fall x > node.key, dann hat node kein rechtes Kind (sonst wäre die Suche da weiter gegangen) und das neue Element wird das rechte Kind

Binärer Suchbaum 7/8

- **Insert(x, value)** … Beispiele:

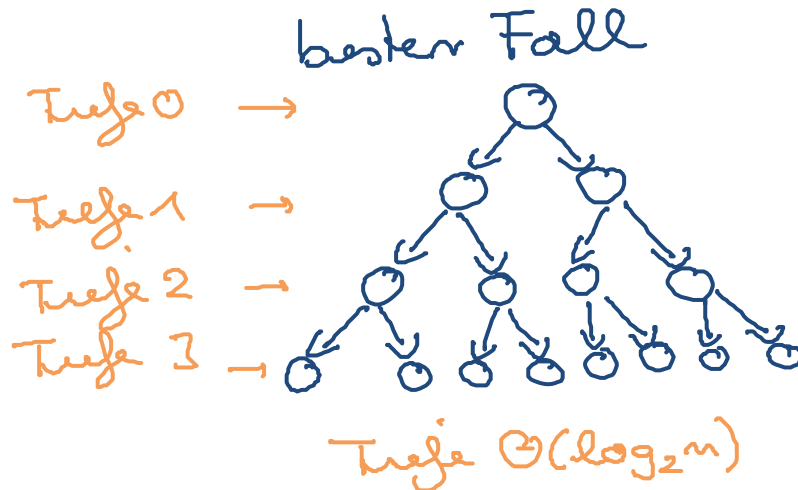


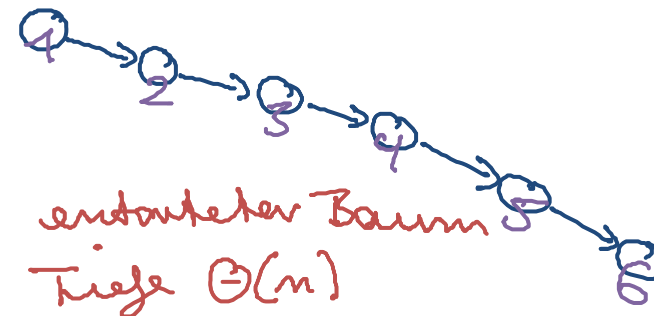
Binärer Suchbaum 8/8

UNI FREIBURG

- 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

#Knoten auf Tiefe $i = 2^i$
#Knoten insgesamt bei Tiefe $d = 2^0 + 2^1 + 2^2 + \dots + 2^d = 2^{d+1} - 1 = n$
$2^{d+1} = n+1$
$\Rightarrow d+1 = \log_2(n+1)$
$\Rightarrow d = \Theta(\log n)$





Übungsblatt 8, Aufgabe 1

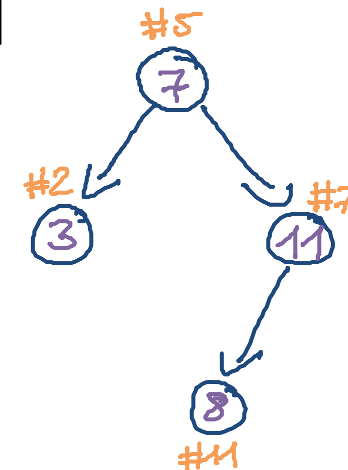
UNI FREIBURG

- Implementierung eines BinarySearchTree
  - Die Methoden insert und lookup
  - Die toString() Methode haben wir schon für Sie geschrieben (sowohl in Java, als auch in C++)

#5, #2, #7, #11 sind beliebige aber eindeutige IDs

```
  [this = #2, left = #0, right = #0, key = 3]
[this = #5, left = #2, right = #7, key = 7]
    [this = #11, left = #0, right = #0, key = 8]
  [this = #7, left = #11, right = #0, key = 11]
```

Einrückungstiefe = Tiefe im Baum



Literatur / Links

UNI FREIBURG

- 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