Informatik II: Algorithmen und Datenstrukturen SS 2015

Vorlesung 8a, Dienstag, 16. Juni 2015

(Sortierte Folgen, 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)
- Inhalt
  - Sortierte Folgen … noch eine nützliche Datenstruktur
  - Binäre Suchbäume … eine mögliche Realisierung davon
  - Ü8, Aufgabe 1: ein praktisches Problem, zu dessen Lösung ein Suchbaum nützlich ist (Präfixsuche in Städtenamen)

Erfahrungen mit dem Ü7 1/2

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 16. Juni 12:00
  - Aufgabe 1 hat den meisten gut gefallen

    LinkedList um lookup, remove und size erweitern
  - Bei Aufgabe 2 waren sich viele unsicher

    Wenig hinzuschreiben, Verständnis war wichtig
  - Viele nutzen die Zusatzaufgabe als "mal schauen, ob ich dafür auch noch ein paar Punkte bekomme"

    Oder weil sie besser gefällt als die Pflichtaufgaben
  - Programmieren fällt vielen leichter als Beweise, ob das was für die Klausur nützt?
  - Wunsch nach Fragestunde, insbesondere Laufzeitbest.

Erfahrungen mit dem Ü7 2/2

- Lösungsskizze Aufgabe 2
  - Anzahl Blockoperationen bei Cuckoo Hashing

    lookup: Θ(1) Blockoperationen

    Man muss an höchstens zwei Stellen schauen

    remove: Θ(1) Blockoperationen

    Aus genau demselben Grund

    insert: Θ(s) Blockoperationen ... s = Anzahl "Rauswürfe"

    Die s Rauswürfe können im worst case jeder in einem anderen Block stehen (für n ≫ M und damit m ≫ M)

    rehash: Θ(n) Blockoperationen

    Die n Elemente können im worst case jeder in einem anderem Block stehen (für n ≫ M und damit m ≫ M)

Sortierte Folgen 1/6

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 keins gibt, finde Elemente mit kleinsten Key > key

    next / previous(element): finde das Element mit dem nächstgrößeren / nächstkleineren Schlüssel, falls es existiert

Sortierte Folgen 2/6

UNI FREIBURG

- Typisches Anwendungsbeispiel: Bereichssuche
  - Ein große Menge von Objekten

    Zum Beispiel Bücher, Wohnungen, sonstige Produkte
  - Typische Suchanfrage: alle Wohnungen zwischen 400 und 600 Euro Monatsmiete

    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 Objekte hinzufügt oder alte löscht, will man nicht jedes Mal erst alles wieder neu sortieren

Sortierte Folgen 3/6

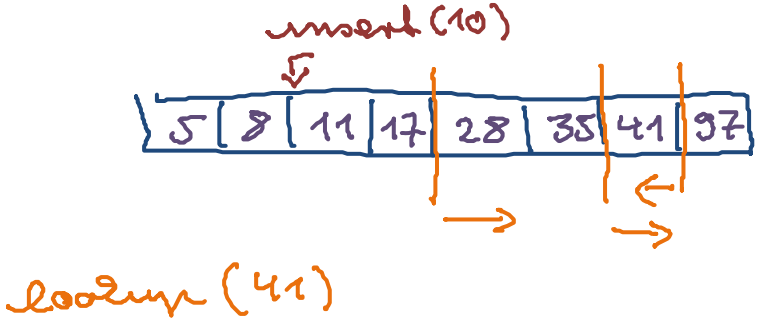
Elemente werden in sortierter Reihenfolge gehalten

- Lösung 1: Einfaches (dynamisches) Feld
  - lookup in Zeit $O(\log n)$

    Das geht mit binärer Suche, siehe unten
  - next und previous in Zeit $O(1)$

    Benachbarte Elemente stehen direkt nebeneinander
  - insert und remove in Zeit bis zu $\Theta(n)$

    Bis zu $\Theta(n)$ Elemente müssen umkopiert werden



Sortierte Folgen 4/6

UNI FREIBURG

- Lösung 2: Hashtabellen
  - insert und remove in erwarteter Zeit O(1)

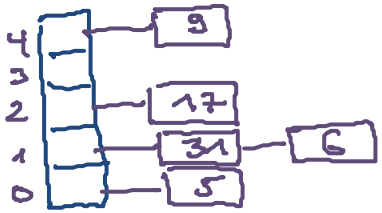
    Bei genügend großer Hashtabelle und guter Hashfunktion
  - lookup in erwarteter Zeit O(1)

    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!

$m = \Omega(n)$
m = Größe Hashtabelle
n = # Elemente

$h(x) = x \bmod 5$



Sortierte Folgen 5/6

UNI FREIBURG

- Lösung 3: (Doppelt) Verkettete Listen
  - next und previous in Zeit O(1)

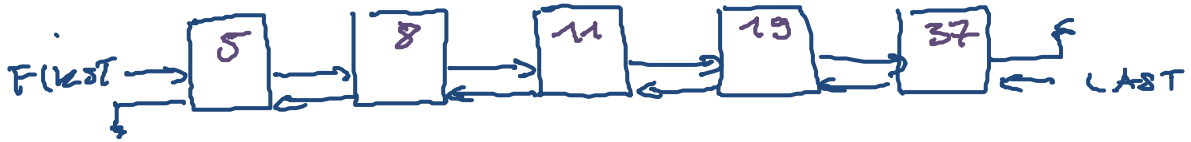
    Jedes Element hat einen Zeiger zum Vorgänger / Nachfolger
  - insert und remove in Zeit O(1)

    Es müssen nur konstant viele Zeiger umgesetzt werden
  - lookup in Zeit bis zu Θ(n)

    siehe ü7.1

    Man könnte die Liste sortiert halten, aber um die richtige Einfügestelle zu finden, muss man sich "durchhangeln"

    Binäre Suche geht nicht auf einer verketteten Liste, weil man nicht einfach (wie im Feld) an Position i springen kann



Sortierte Folgen 6/6

UNI FREIBURG

- Lösung 4: Suchbäume
  - next und previous in Zeit O(1)

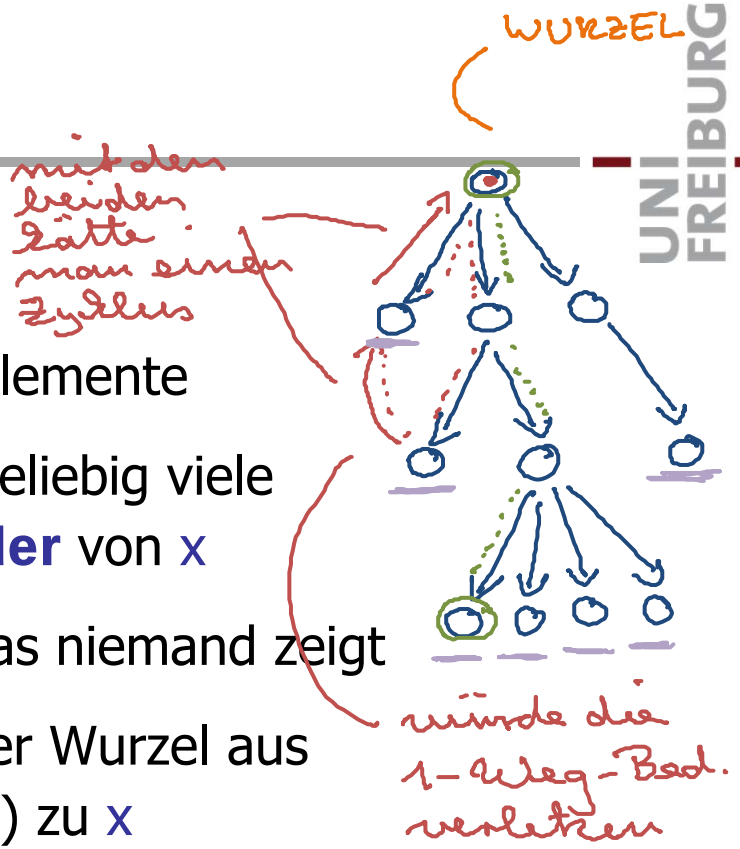
    Entsprechende Zeiger wie bei der verketteten Liste
  - insert und remove in Zeit O(1)

    Ebenfalls wie bei der verketteten Liste
  - lookup in Zeit O(log n)

    Eine Baumstruktur hilft jetzt beim effizienten Suchen

    Wie genau, schauen wir uns im Rest der Vorlesung heute und weiter morgen an

Binärer Suchbaum 1/11



- Allgemeiner Baum, Definition
  - Elemente, mit Zeiger auf andere Elemente

    Von jedem Element x Zeiger auf beliebig viele andere Elemente, die heißen **Kinder** von x

    Es gibt <u>ein</u> **Wurzel**element, auf das niemand zeigt

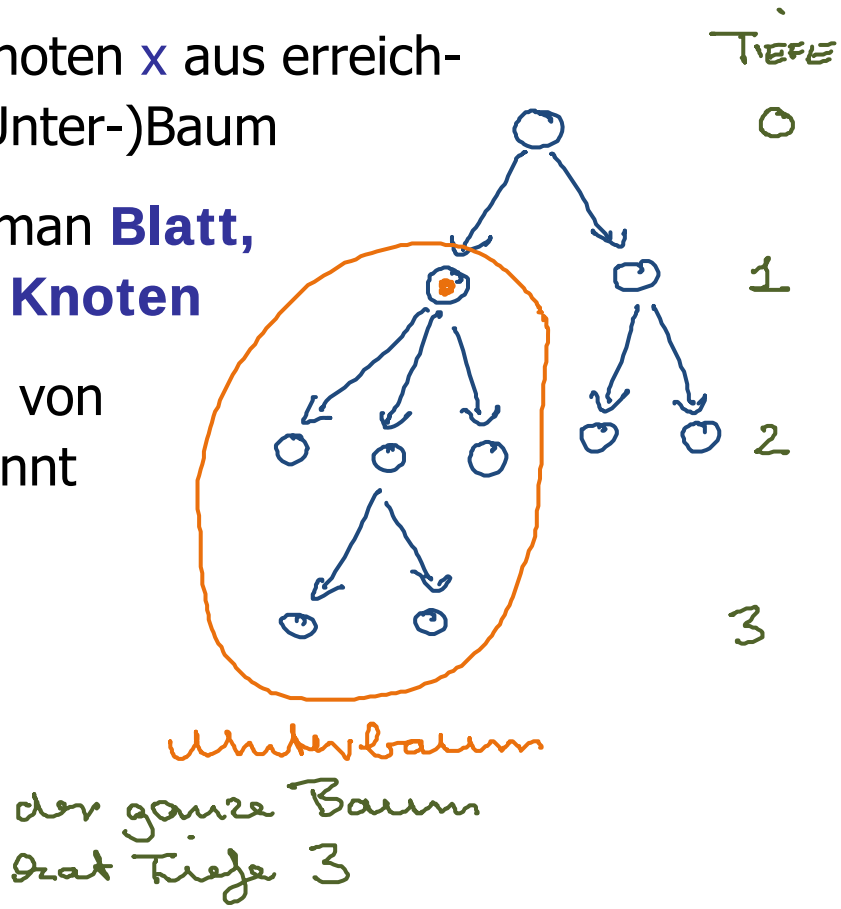
    Für jedes Element x gibt es von der Wurzel aus <u>genau einen</u> Weg (über die Zeiger) zu x

    Es gibt keinen **Zyklus** = eine Folge von Zeigern, die von einem Element x wieder zu x führt

Knoten ohne Kinder heißen Blätter

Binärer Suchbaum 2/11

- Allgemeiner Baum, Terminologie
  - 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 maximale Tiefe eines Knotens



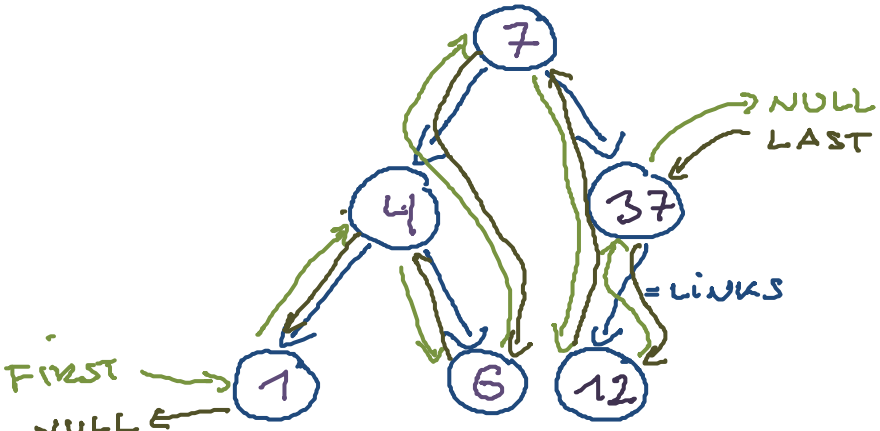
Binärer Suchbaum 3/11

- Binärer Suchbaum, Definition

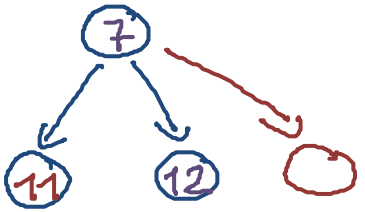
  Für's erste: alle KEYS sind verschieden

  - 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



Ich male nur die KEYS rein, nicht die VALUES



VERBOTEN

Binärer Suchbaum 4/11

UNI FREIBURG

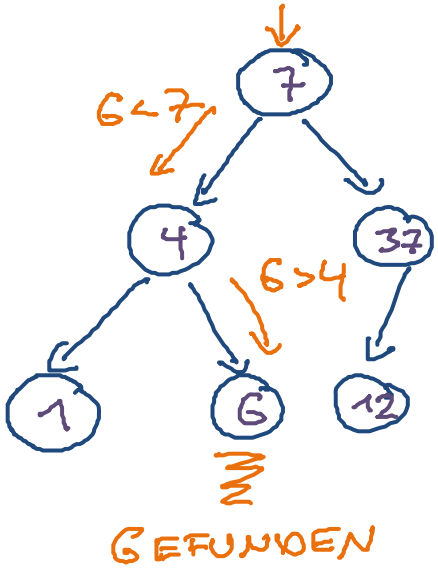
- Die Operation **lookup(x)**
  - Von der Wurzel abwärts suchen, und an jeden Knoten node

    falls x == node.key … gefunden!

    falls x < node.key … nach links weiter suchen

    falls x > node.key … nach rechts weiter suchen

    Wenn es den Key im Baum gibt, findet man ihn so sicher



lookup(6)

Binärer Suchbaum 5/11

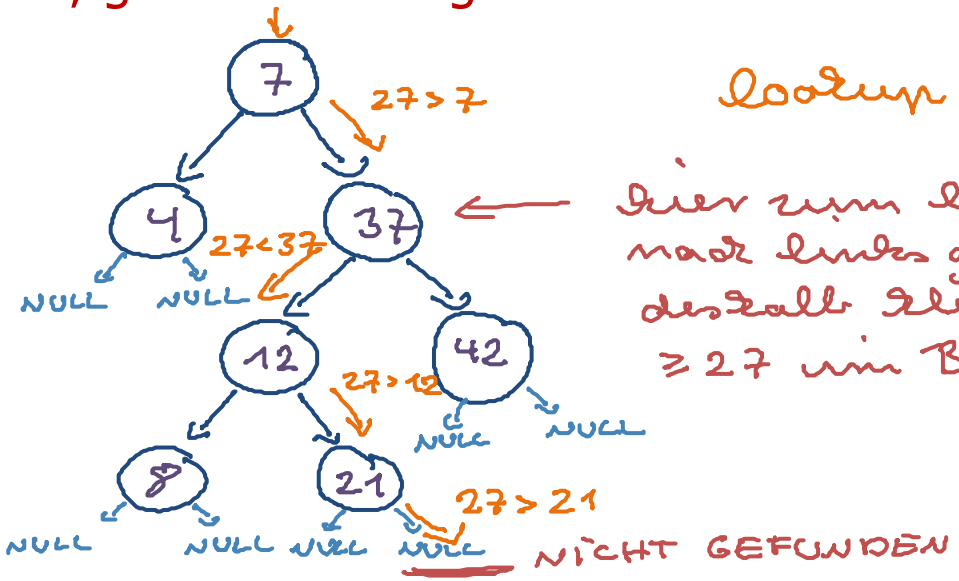
- Die Operation **lookup(x)**
  - Wenn es den Key im Baum **nicht** gibt:

    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 immer nur nach rechts geht und den Key nie findet, gibt es keinen größeren Schlüssel im Baum

Wenn man NEXT hat, kann man auch das benutzen. Hat man aber aus Effizienzgründen manchmal nicht.



lookup(27)

hier zum letzten Mal nach links gegangen, deshalb kleinster KEY ≥ 27 im Baum

Binärer Suchbaum 6/11

UNI FREIBURG

- Die Operation **insert(x, value)**
  - Erst mal ein lookup(x)
  - Wenn es x im Baum schon gibt, überschreiben wir einfach das Element an dem Knoten und sind fertig
  - Wenn es x im Baum **nicht** gibt, können wir so lange nach unten gehen, wie gilt

    Entweder: $x < node.key$, und es gibt ein linkes Kind

    Oder: $x > node.key$, und es gibt ein rechtes Kind

Binärer Suchbaum 7/11

```
class Node {
  ...
  Node leftChild;
  Node rightChild;
}
```

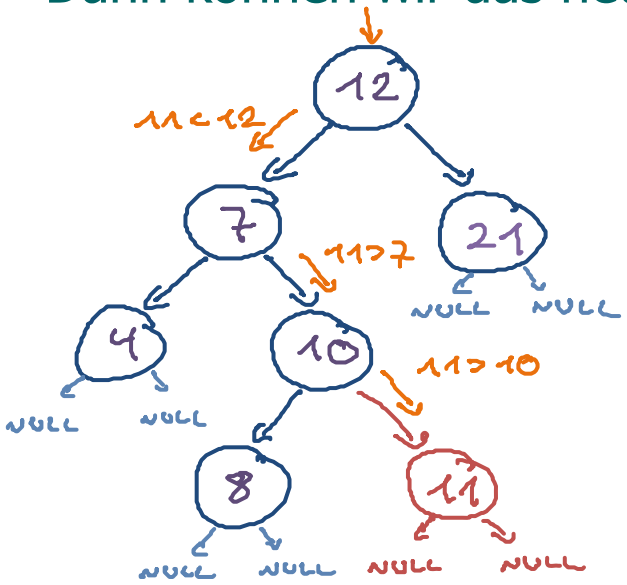
- Die Operation **insert(x, value)**
  - Wenn es an einem Knoten nicht mehr weitergeht, ist also

    entweder: x < node.key aber es gibt kein linkes Kind

    Dann können wir das neue Element links einfügen !

    oder: x > node.key aber es gibt kein rechtes Kind

    Dann können wir das neue Element rechts einfügen !



insert(11, ...)

Binärer Suchbaum 8/11

UNI FREIBURG

- Laufzeit von insert und lookup
  - In Zeit O(d), wobei d die Tiefe des Baumes ist

    Man geht ja in jedem Schritt eins nach unten

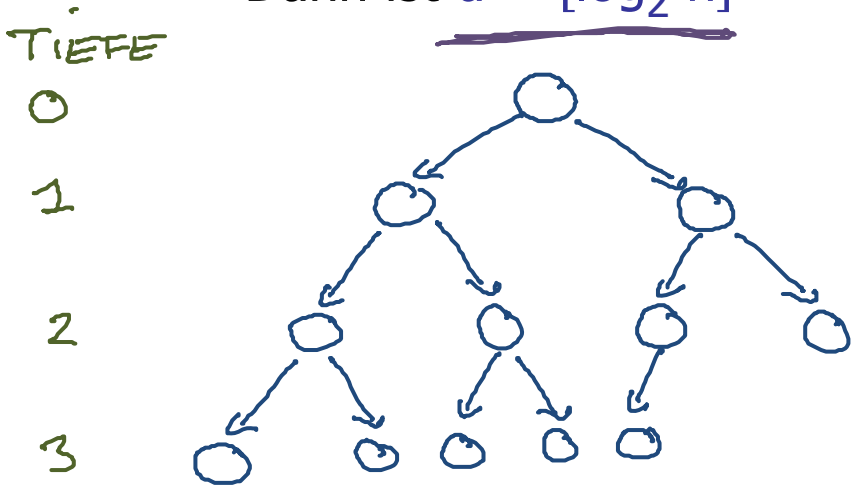
    Und wenn es nicht mehr nach unten geht, ist man fertig

    Wenn man den Schlüssel schon weiter oben im Baum findet, kann es auch schneller gehen
  - Wir hätten gerne eine Abhängigkeit von der Anzahl n der Elemente ... wie hängt die mit d zusammen ?

Binärer Suchbaum 9/11

- Tiefe des Baumes, best case
  - Die Tiefe des Baumes (siehe Folie 12) ist am niedrigsten, wenn jeder innere Knoten zwei Kinder hat

    Außer vielleicht einige Knoten der "vorletzten" Tiefe
  - Dann ist $d = \lfloor \log_2 n \rfloor$



Beweis →

$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$5 \leq 2^3$

$= 2 \dots$

$n = 7 \;,\; \lfloor \log_2 n \rfloor = 2$

$n = 8 \;,\; \lfloor \log_2 n \rfloor = 3$ ($= 3$)

$n > 2^0 + 2^1 + \dots + 2^{d-1}$

$= 2^d - 1$

$\Rightarrow n \geq 2^d$

$\Rightarrow d \leq \log_2 n$

$n \leq 2^0 + 2^1 + \dots + 2^d$

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

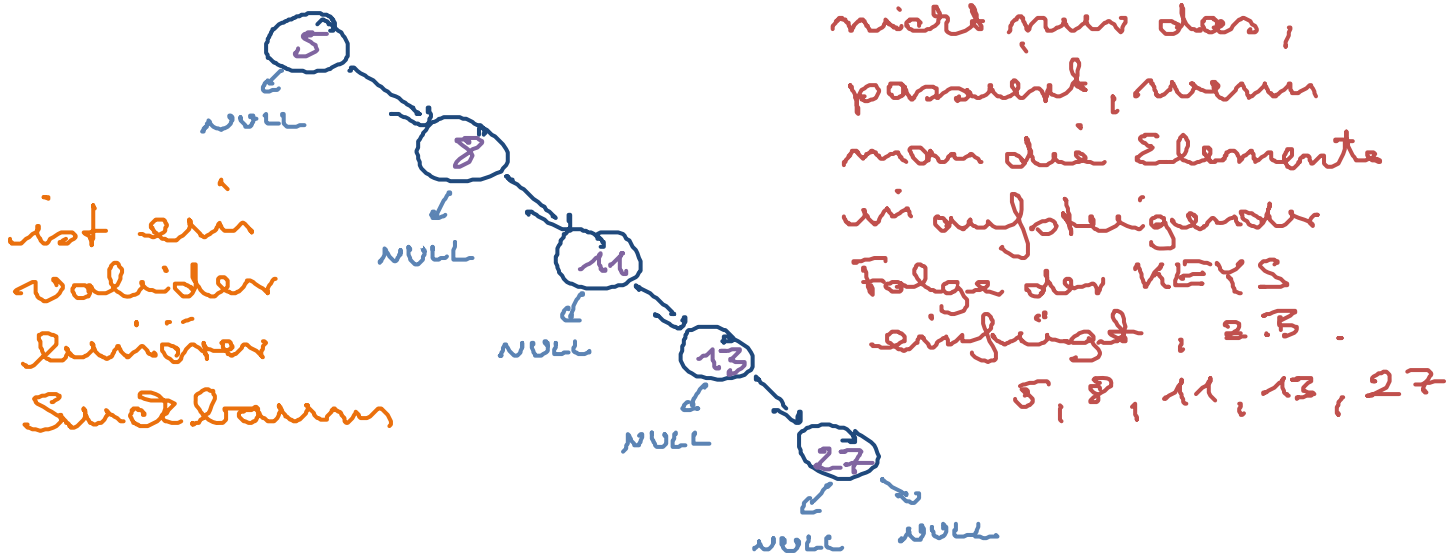
$\Rightarrow d + 1 \geq \log_2 n$

$\Rightarrow d \geq \log_2 n - 1$

Binärer Suchbaum 10/11

- Tiefe des Baumes, worst case
  - Die Tiefe des Baumes (siehe Folie 12) ist am höchsten, wenn jeder innere Knoten nur ein Kind hat
  - Dann ist $d = n - 1$

  Wenn man immer $\Theta(\log n)$ will, muss man den Baum gelegentlich rebalancieren … das machen wir morgen



Binärer Suchbaum 11/11

UNI FREIBURG

- Verwendung in Java, C++ und Python
  - **Java:** java.util.TreeMap<KeyType, ValueType>
  - **C++:** std::map<KeyType, ValueType>
  - **Python:** bintrees.BinaryTree

    bintrees ist nicht Teil der Standardsprache und muss von Hand nachinstalliert werden, am einfachsten mit

    wget https://pypi.python.org/.../bintrees-2.0.2.zip
    unzip bintrees-2.0.2.zip
    cd bintrees.2.0.2
    python3 setup.py install --user

    Die vollständige URL finden Sie auf dem Wiki

Literatur / Links

UNI FREIBURG

- Suchbäume
  - In Mehlhorn/Sanders:

    7 Sorted Sequences
  - In Wikipedia

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

    http://en.wikipedia.org/wiki/Binary_search_tree