Informatik II: Algorithmen und Datenstrukturen SS 2017

Vorlesung 8a, Dienstag, 20. Juni 2017

(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
  - Erfahrungen mit dem ÜB7 — Listen, Cache-Effizienz
  - Was ist ein Plagiat — Erinnerung + Beispiel
- Inhalt
  - Sortierte Folgen — Definition + Beispiel
  - Binäre Suchbäume
  - ÜB8: ein praktisches Problem, und Sie müssen selber schauen, welche Datenstrukturen geeignet sind

    (Bewusst weniger Vorgaben als sonst)

    Achtung: stellen wir heute gegen 17 Uhr erst online

Erfahrungen mit dem ÜB7 1/2

UNI FREIBURG

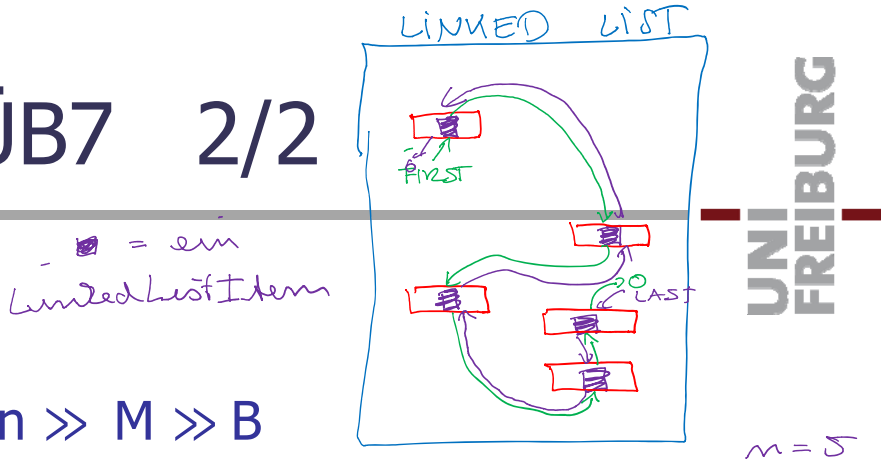
- Zusammenfassung / Auszüge
  - Aufgabe 1 gut machbar + hat vielen viel Spaß gemacht
  - Verständnisschwierigkeiten bei Aufgabe 2 (Block-Ops)

    Häufige Rückmeldung: "Waren die Aufgaben diesmal einfacher, oder habe ich etwas falsch gemacht?"

    Wichtig: das zu erkennen ist ein wichtiger Verständnistest
  - "Sobald die Elemente auf dem Heap liegen, ist es mit der Cache-Effizienz eh dahin"
  - Können sie bei Gelegenheit den Livestream an die Tafel projizieren, um Endlosrekursion zu erzeugen?

    Gute Idee! Geht auch zu Hause mit Handy+TV+SmartView

Erfahrungen mit dem ÜB7 2/2



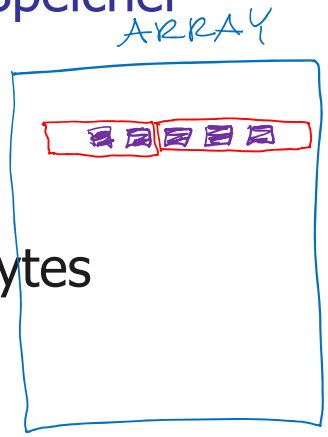
- Lösungsskizze Aufgabe 2
  - Sie konnten annehmen, dass n ≫ M ≫ B

    Stand nicht explizit auf dem ÜB, wurde im Forum geklärt
  - Anzahl Blockoperationen von **reverse()**

    Ohne Weiteres muss man für reverse() alle **n** Elemente der Liste anschauen und verändern

    Die Elemente stehen an beliebigen Stellen im Speicher

    Also im worst case **Θ(n)** Blockoperationen



  - Anzahl Blockoperationen von **splice()**

    Man muss nur an zwei Stellen konstant viele Bytes anschauen bzw. verändern

    Also immer **Θ(1)** Blockoperationen

Was ist ein Plagiat

- Erinnerung + Beispiel
  - Im Rahmen von Lehrveranstaltungen geht es vor allem um das Übernehmen fremder Texte (inklusive Code)
  - Ab wann gilt ein Text als übernommen vs. selbst erdacht?

    Einzelne Worte darf man offenbar wieder verwenden

    Ganze Sätze sind in der Regel schon einzigartig

    Schon die Aneinanderreihung von k Worten ist oft so charakteristisch, dass man damit ein Dokument eindeutig identifizieren kann, für relativ kleine k

    Beispiele von unseren Webseiten:

    "Das bedeutet, dass wir uns komplexe"

    "research interest is aptly described"

Sortierte Folgen 1/6

muss nicht == key sein

- 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 ein Element mit kleinsten Key ≥ key

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

    (so, dass Iteration über alle Elemente möglich ist)

Sortierte Folgen 2/6

- 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

wo die Elemente in sortierter Reihenfolge stehen

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

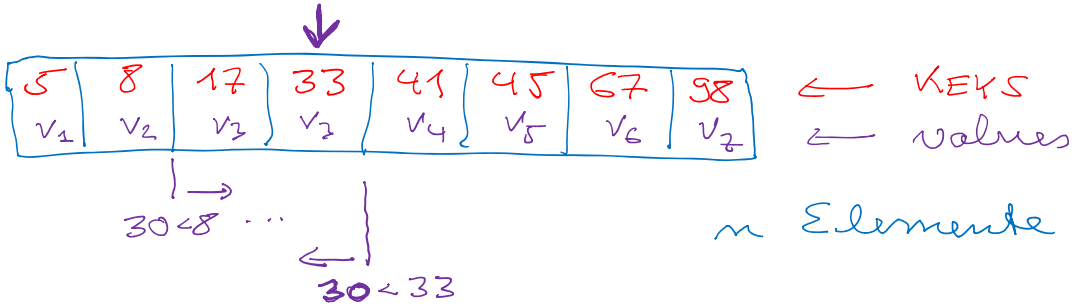
    → wenn man bei vielen gleichen KEYS das „linkeste" finden will

    lookup(30)

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

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

    Bis zu Θ(n) Elemente müssen umkopiert werden



Sortierte Folgen 4/6

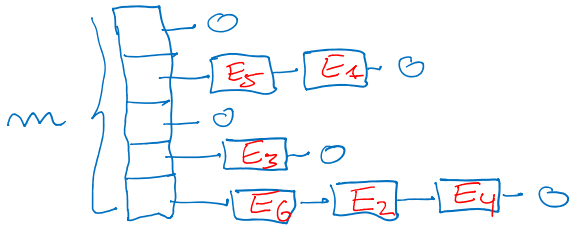
lookup(key): kleinster Schlüssel ≥ key

- Lösung 2: Hashtabellen
  - insert und remove in Zeit: $O(1)$ im Erwartungsfall wenn $m = \Theta(n)$

    Bei genügend großer Hashtabelle und guter Hashfunktion
  - lookup in ~~erwarteter~~ Zeit: $O(1)$ oder bis zu $\Theta(n)$

    Aber nur wenn es ein Element mit **exakt** dem gegebenen Key gibt, sonst bekommt man **gar nichts**
  - next und previous in Zeit: bis zu $\Theta(n)$

    Die Reihenfolge, in der die Elemente in einer Hashtabelle stehen hat nichts mit der Reihenfolge der Keys zu tun!



n Elemente

Sortierte Folgen 5/6

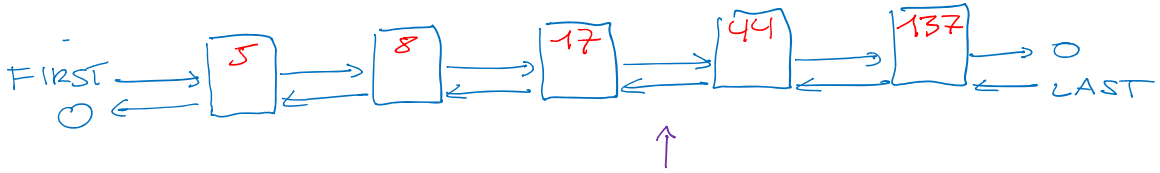
- Lösung 3: (Doppelt) Verkettete Listen
  - next und previous in Zeit: $O(1)$

    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 $O(n)$ Loolup (20)

    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

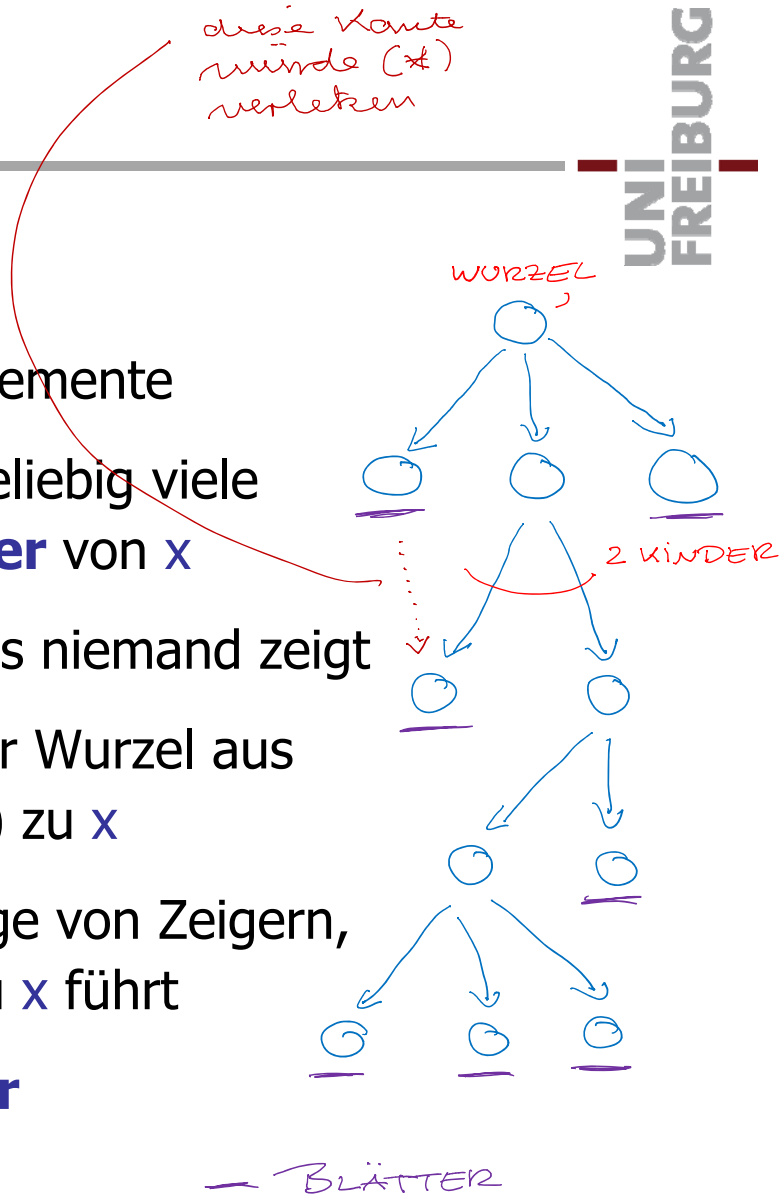
    Von jedem Element x Zeiger auf beliebig viele andere Elemente, die heißen **Kinder** von x

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

  (*) Für jedes Element x gibt es von der Wurzel aus genau einen Weg (über die Zeiger) zu x

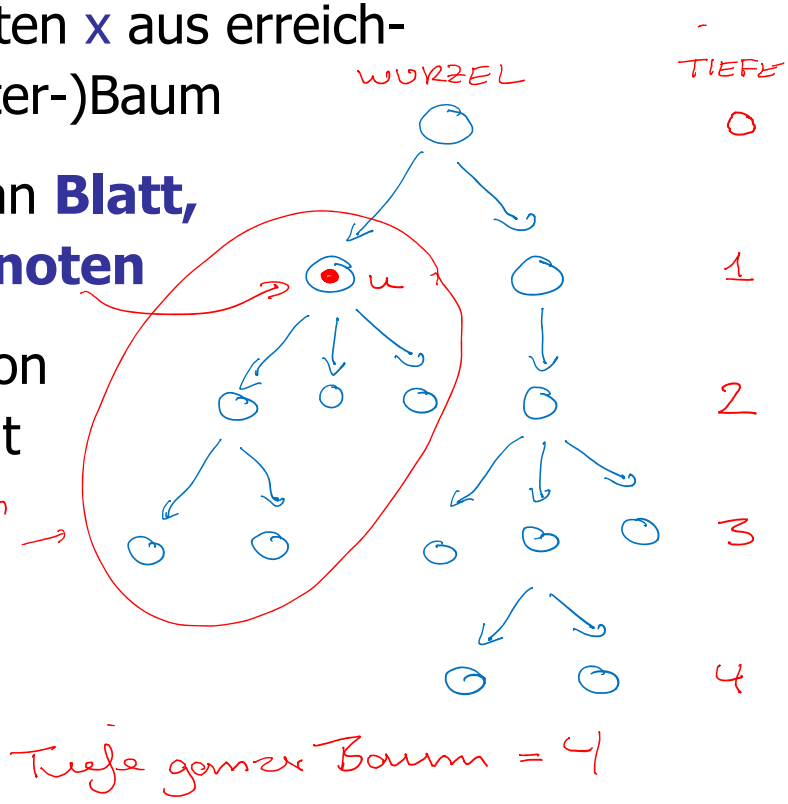
    Es gibt keinen **Zyklus** = keine Folge von Zeigern, die von einem Element x wieder zu x führt

    Knoten ohne Kinder heißen **Blätter**



Binärer Suchbaum 2/11

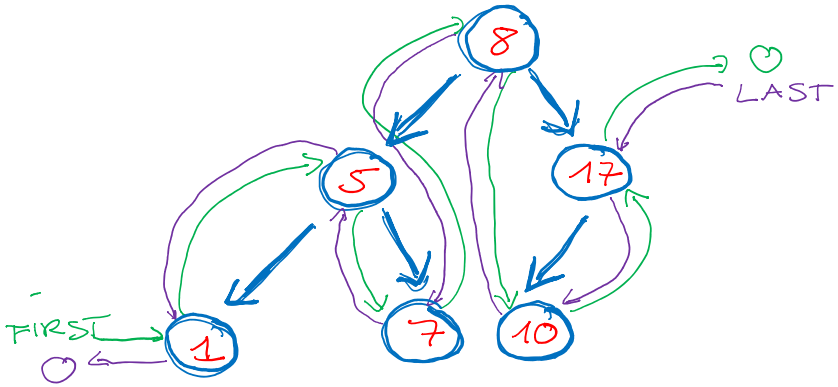
- Allgemeiner Baum, Terminologie
  - Die Elemente nennt man auch **Knoten** (English: **node**)
  - Alle Elemente, die von einem Knoten x aus erreichbar sind, bilden wieder einen (Unter-)Baum
  - Einen Knoten ohne Kind nennt man **Blatt,** die anderen nennt man **innere Knoten**
  - Die Anzahl Zeiger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



Binärer Suchbaum 4/11

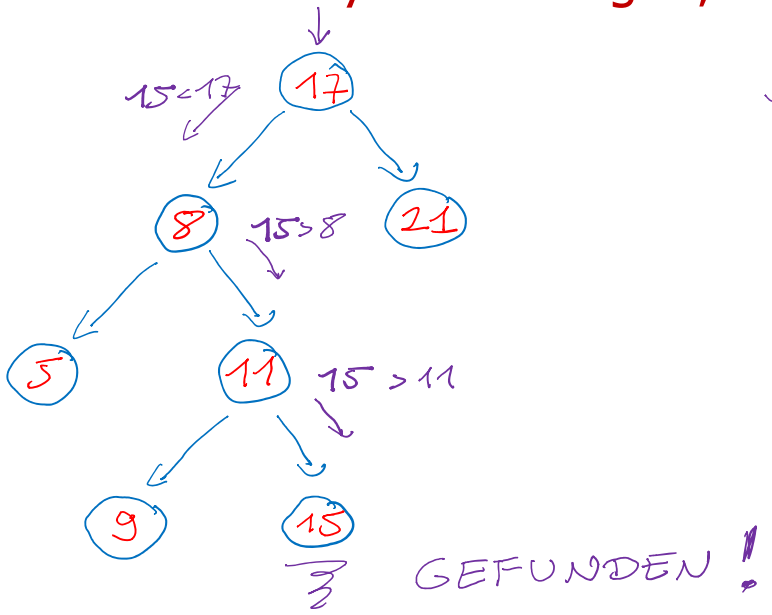
- Die Operation **lookup(x)**
  - Von der Wurzel abwärts suchen, und an jeden Knoten node

    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



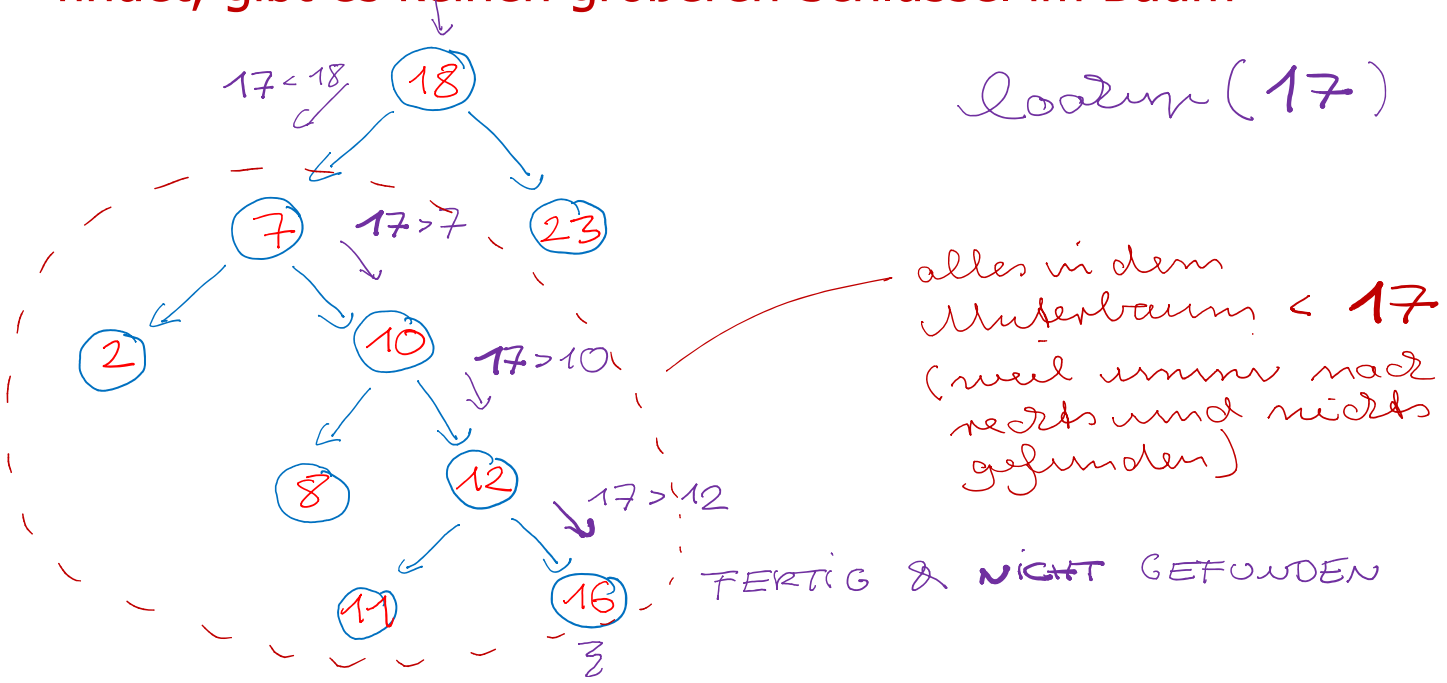
lookup(15)

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

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



Binärer Suchbaum 6/11

- 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

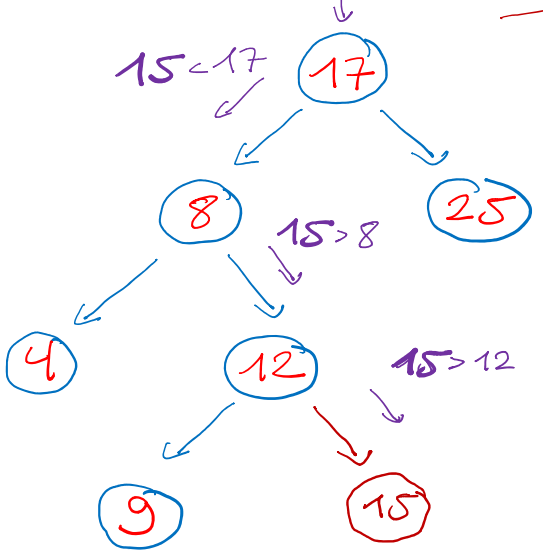
- Die Operation **insert(x, value)**
  - Wenn es an einem Knoten nicht mehr weitergeht, ist also

    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(15, value)

Binärer Suchbaum 8/11

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

    Es geht in jedem Schritt eins nach unten, nie nach oben

    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

$n > 1 + 2 + 4 + \cdots + 2^{d-1} = 2^d - 1$

$\Rightarrow n \geq 2^d \Rightarrow d \leq \log_2 n$

$n \leq 1 + 2 + 4 + \cdots + 2^d$

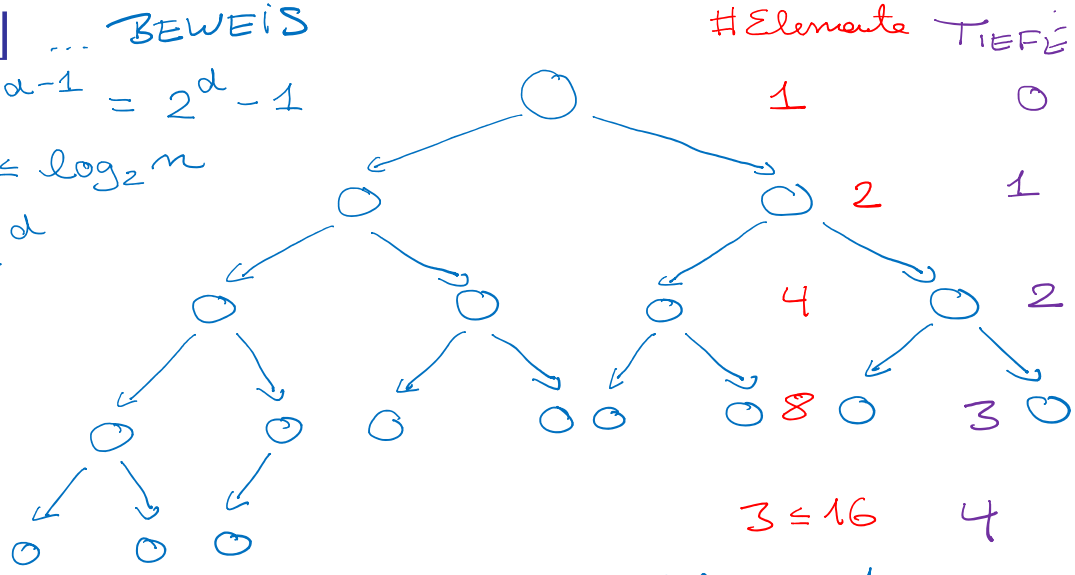
$= 2^{d+1} - 1$

$n < 2^{d+1}$

$\Rightarrow d + 1 > \log_2 n$

$d > \log_2 n - 1$

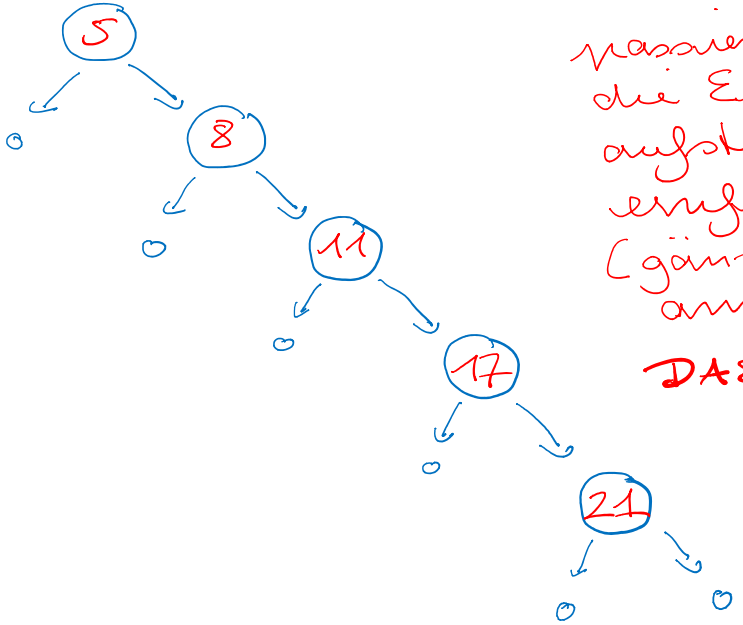
$\Rightarrow d = \lfloor \log_2 n \rfloor$



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 Θ(log n) will, muss man den Baum gelegentlich rebalancieren … das machen wir morgen



passiert, wenn man die Elemente in gänzlich aufsteigender Reihenfolge einfügt (gänzlich absteigend anna-log)

DAS PASSIERT SCHON MAL

Binärer Suchbaum 11/11

die HashMap heißt std::unordered_map

die HashMap heißt java.util.HashMap

- Verwendung in Java, C++ und Python
  - **Java:** java.util.TreeMap<KeyType, ValueType>
  - **C++:** std::map<KeyType, ValueType>
  - **Python:** bintrees.BinaryTree
  - In Python ist bintrees ist nicht Teil der Standardsprache und muss von Hand nachinstalliert werden, z.B. so:

```
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
```

Axel sagt, es geht auch:

```
pip install bintrees
```

Siehe https://pypi.python.org/pypi/bintrees/2.0.2

Literatur / Links

- Suchbäume
  - In Mehlhorn/Sanders:

    7 Sorted Sequences
  - In Wikipedia

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

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