Woche 8 – Übersicht & Tricks

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu Serie 6: Allgemeines

- Invariantenbeweise: Bitte auf korrekte Form achten. Falsche oder ungenaue Induktionshypothesen führen quasi immer zum Verlust von (in der Klausur im worst case allen) Punkten. Base Cases müssen kurz gezeigt werden. Vorsicht beim Verwenden von "for all" in der Induktionshypothese wenn ihr starke Induktion verwenden wollt. Bei starker Induktion fixiert ihr einen beliebigen Wert, bspw. ein bel. $n \in \mathbb{N}$ und nehmt dann an, dass irgendwas für alle $k < n$ gilt, bspw. $\forall k \in \mathbb{N}. 0 \leq k < n \Rightarrow A(k)$.
- Die Berechnungsvorschrift ist das **Wichtigste**. Alles andere gibt in der Klausur nur Teilpunkte. Also bemüht euch darum, die ordentlich zu machen, Orientierung gibt ML.

Stack:

Der Stack ist die erste abstrakte Datenstruktur, die ihr kennenlernt: Er funktioniert nach dem LIFO Prinzip: **L**ast **I**n **F**irst **O**ut. Bildlich kann man sich das sehr gut wie einen Stapel vorstellen: Ihr legt Dinge auf einen Stapel und wenn ihr etwas von dem Stapel zurückhaben wollt, bekommt ihr das oberste Element: Genau das, dass ihr ja als letztes auf den Stapel obendrauf gelegt habt. Wir können einen Stack problemlos selbst implementieren (Es gibt aber sogar eine fertige Stack Classe in Java, die u.U. importiert werden kann):

```java
class Stack {

    int length;
    Node head;

    public Stack(){
      this.length = 0;
      this.head = null;
    }

    public int peek() {
        if (isEmpty()) {
            return -1;
        }
        return head.val;
    }

    public int pop() {
        if (isEmpty()) {
            return -1;
        }
        int headVal = head.val;
        this.head = head.next;
        this.length--;
        return headVal;
    }
    public void push(int val) {
        Node nextHead = new Node(val, this.head);
        this.head = nextHead;
        this.length++;
    }
    public int size() {
        return this.length;
    }
    public boolean isEmpty() {
        return this.length == 0;
    }
}
```

```java
class Node {
  int val;
  Node next;

  public Node(int val, Node next){
    this.val = val;
    this.next = next;
  }
}

public class Test {
    public static void main(String args[]) {
      Stack s = new Stack();
      s.push(1);
      s.push(2);
      s.push(3);
      s.pop();
      System.out.println(s.pop());
      System.out.println(s.peek());
    }
}
```

Queue:

Die Queue funktioniert im Grunde genommen genau wie ein Stack, nur das wir nicht immer das neuste – sondern das älteste (also am längsten in der Datenstruktur verweilende) – Element aus der Datenstruktur extrahieren. Das Prinzip heisst **FIFO: F**irst **I**n **F**irst **O**ut. Unter Umständen wird dasPrinzip der Queue noch leicht modifiziert, sodass sie zu einer sogenannten **Priority Queue** wird, dieimmer das Wichtigste (normalerweise das grösste oder kleinste) Element zurückgibt. Dafür verwenden wir das Prinzip des Heaps (siehe Zusammenfassung 5 für eine ausführliche Erklärung).

Allgemeine Datenstrukturen:

Um abstrakte Datenstrukturen wie Queues, Stacks und weitere zu implementieren, verwenden wir Datenstrukturen, die ihr bereits kennt:

(Sortierte/Unsortierte) Arrays, Linked Lists, Doubly Linked Lists, Heaps.

Dazu kommen nun noch zwei weitere Datenstrukturen: Binary Search Trees (BSTs) und AVL Trees.

Binary Search Tree (BST):

Zunächst führen wir diese neue Datenstruktur formal ein: Ein BST besteht aus Knoten, die baumartig miteinander verbunden sind. Dabei ist jeder Knoten entweder ein Blatt (hat keine Nachfolger) oder ein innerer Knoten und hat bis zu 2 Nachfolgerbäume. Elemente, die wir in unserem BST abspeichern wollen, speichern wir nur in inneren Knoten. Also speichern wir in jedem inneren Knoten den Wert des zu speichernden Elements und einen Zeiger auf jeden seiner Nachfolgerbäume. Ein Zeiger auf null ist äquivalent zu einem Zeiger auf ein Blatt. Es gibt nur höchstens einen Knoten, der keinen Vorgänger hat, wir nennen ihn Wurzel – oder üblicherweise: **root**.

Damit ein beliebiger Baum als BST klassifiziert werden kann, muss er die **Suchbaumeigenschaft** erfüllen:

Für jeden Knoten im Baum gilt: Der Knoten speichert einen Wert k, alle im linken Nachfolger-Teilbaum gespeicherten Knoten sind kleiner als k, alle im rechten Nachfolger-Teilbaum gespeicherten Knoten sind grösser als k.

Hier nun eine mögliche Java-Implementation:

```
class Tree {
  int val;
  Tree left;
  Tree right;
  Public Tree(int val, Tree left, Tree right){
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
```

Aber wie können wir ein neues Element in einen BST einfügen? Ganz einfach: Wir suchen die richtige Stelle und hängen das Element dort an den Baum. Referenz: https://leetcode.com/problems/insert-into-a-binary-search-tree/

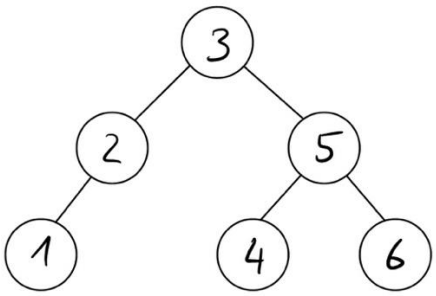
```java
public class TreeNode {
     int val;
     TreeNode left;
     TreeNode right;
     TreeNode( ) {}
     TreeNode(int val) { this.val = val; }
     TreeNode(int val, TreeNode left, TreeNode right) {
         this.val = val;
         this.left = left;
         this.right = right;
     }
}

class Solution {
    public TreeNode insertIntoBST(TreeNode root, int val) {
        if(root == null){
            root = new TreeNode(val);
            return root;
        }
        else if(val < root.val){
            if(root.left == null)
                root.left = new TreeNode(val);
            else
                insertIntoBST(root.left, val);
            return root;
        }
        else{
            if(root.right == null)
                root.right = new TreeNode(val);
            else
                insertIntoBST(root.right, val);
            return root;
        }

    }
}
```

Wir traversieren den Baum und gucken an jeder Stelle, in welchen Teilbaum wir weiter vorrücken müssen und ob es diesen Teilbaum überhaupt noch gibt. Denn falls nicht können wir das Element an die entsprechende Stelle anhängen.

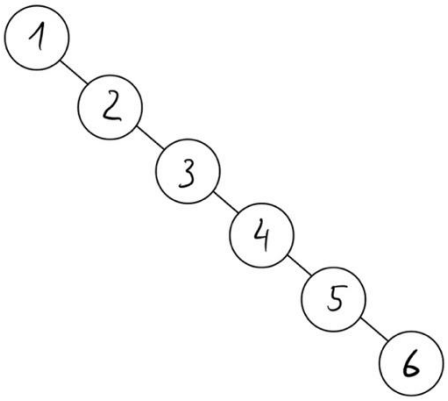
AVL-Trees:

Binary Search Trees haben ein Problem: Sie können abhängig von der Reihenfolge, in der die Elemente eingefügt werden extrem hässlich werden, sodass eine effiziente Suche nicht mehr möglich ist. Beispiel: Wir wollen die Integers {1,2,3,4,5,6} speichern. Dafür betrachten wir folgende Reihenfolgen: (3,5,2,4,6,1) und (1,2,3,4,5,6):

(3,5,2,4,6,1):



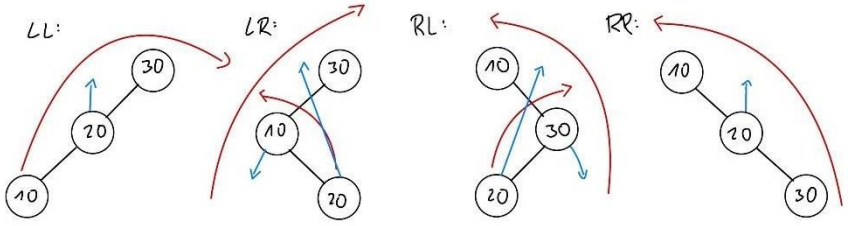
(1,2,3,4,5,6):

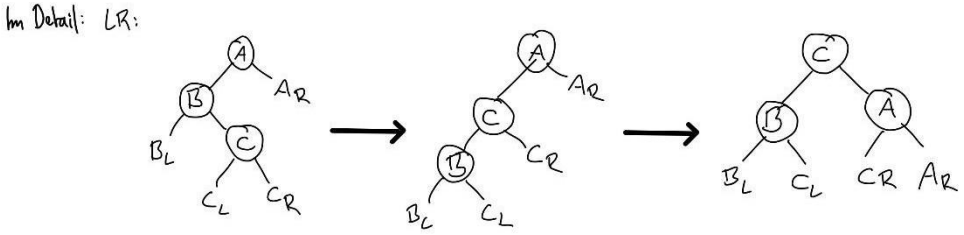


Im oberen Fall können wir in $O(\log(n))$ bestimmen, ob es ein Element k in unserem Baum gibt. Im unteren Fall nur in $O(n)$ – und das nur, weil der Baum nicht balanciert ist. Es sind exakt dieselben Knoten, nur wurden die Werte in anderer Reihenfolge abgespeichert.

Dieses Problem beheben AVL-Trees: Sie stellen eine Zusätzliche Bedingung an jeden ihrer Knoten: Der Unterschied der Höhen seiner beiden Nachfolgerknoten darf höchstens 1 sein, ansonsten muss rebalanciert werden. Die genaue Definition des Rebalancierens kann im Skript gefunden werden. Hier werde ich nur Zeigen, wie genau es immer abläuft, sodass ihr alle Aufgaben dieser Art problemlos lösen könnt.

Es gibt exakt **vier mögliche Fälle** für jeden Knoten: (Ich habe sie hier mit dem Beispiel {10,20,30} illustriert:



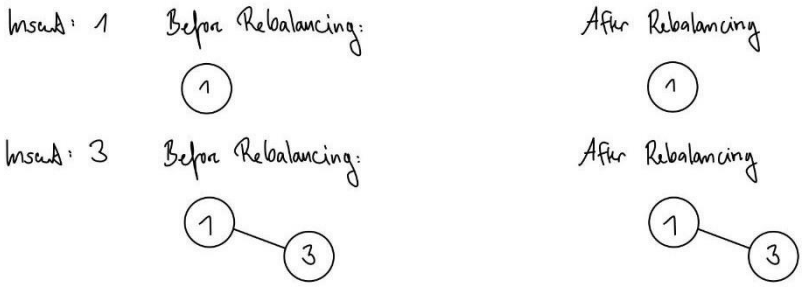


In jedem der vier Fälle ist die AVL Bedingung immer an dem «30»-Knoten verletzt. Je zweimal ist entweder der linke Teilbaum mehr als einen Knoten oder rechte Teilbaum mehr als einen Knoten tiefer als der jeweils andere Teilbaum. Für die vier Möglichkeiten einer Imbalance gibt es vier Rotationen, die essentiell ziemlich ähnlich sind. Im Detail zeige ich die LR-Rotation.

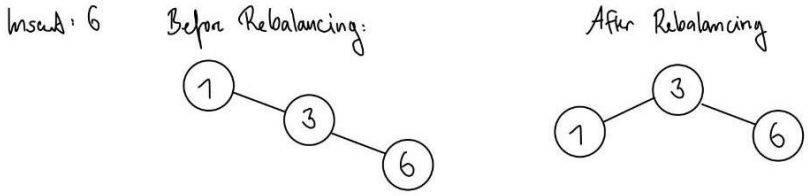
Nun wollen wir ein Beispiel betrachten:

Wir wollen die Zahlen von 1 bis 8 in dieser Reihenfolge: <1, 3, 6, 5, 4, 8, 7, 2> in einen anfangs leeren AVL-Tree einfügen:

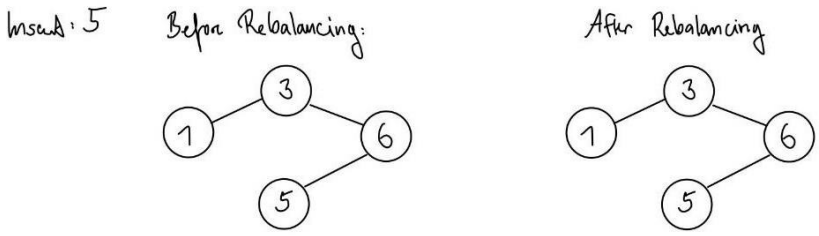
Die ersten Schritte sind ziemlich unspektakulär, da die AVL-Eigenschaft an keiner Stelle verletzt wird und es damit auch nicht zu Rotationen kommt. Wir fügen Elemente so hinzu, wie wir es vom BST kennen:



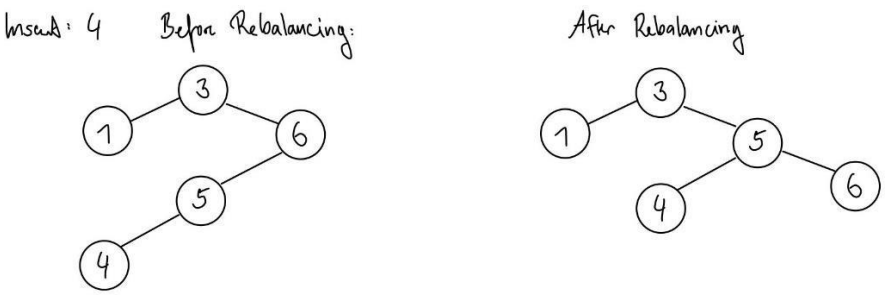
Nun fügen wir das Element 6 ein und es kommt zum ersten mal zu einer Rotation. In diesem Fall handelt es sich dabei offensichtlich um eine RR-Rotation:



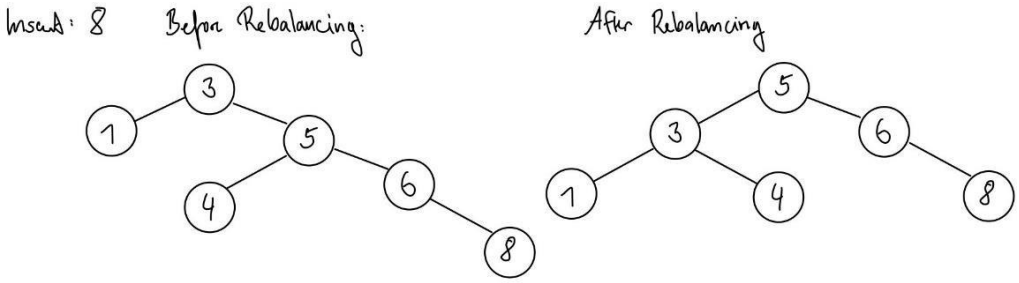
Als nächstes fügen wir das Element 5 ein, verletzen dabei allerdings nicht die AVL-Eigenschaft:



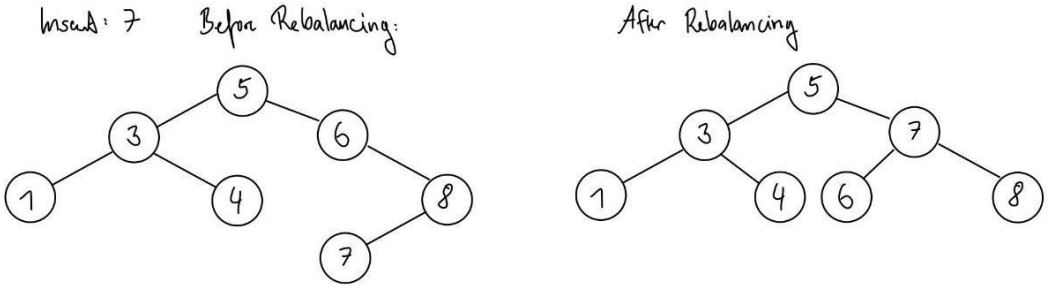
Nun fügen wir das Element 4 ein und verletzen die AVL-Eigenschaft: Wir gucken, wo wir sie **zuerst** verletzen, dabei schauen wir im Baum erst die unteren Knoten an. Wir verletzen sie am Knoten 6 und führen also eine LL-Rotation durch. Danach wird die AVL-Eigenschaft nirgendwo mehr verletzt.



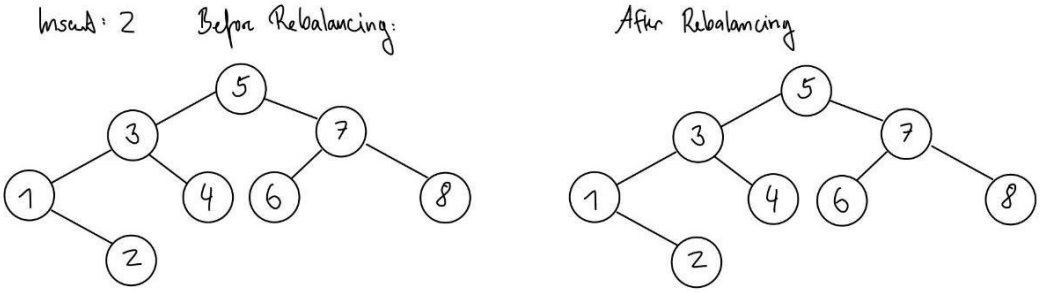
Mit dem Einfügen des Elementes 8 verletzen wir die AVL-Eigenschaft erneut: Dieses Mal am Knoten 3. Sein linker Teilbaum hat bloss Tiefe = 1, während der rechte Teilbaum Tiefe = 3 hat. Es handelt sich wieder um eine RR-Rotation.



Nun fügen wir die 7 ein: Es gibt eine Imbalance am Knoten 6: Sein linker Sub-Tree hat Tiefe = 0, während der rechte Tiefe = 2 hat. Eine RL-Rotation löst das Problem.



Abschliessend fügen wir noch das Element 2 ein: Es gibt keine Imbalance und damit auch keine Rotationen.



Nun wissen wir, wie man Elemente in einen AVL-Tree einfügt und dabei Rotationen einsetzt, um die Balance, also die AVL-Eigenschaft zu erhalten. Es bleibt noch eine wichtige Frage: Wie genau löscht man Elemente aus der Datenstruktur?

Für Knoten, die nur Blätter als Nachfolger haben, ist es trivial. Wir entfernen sie einfach und – falls nötig – führen wir Rotationen durch, um den Baum zu reparieren (Im obigen letzten Schritt bspw., wenn man anschliessend die 4 löschen würde: Es käme zu einer Imbalance am Knoten 3).

Für innere Knoten ist es ein klein wenig komplizierter: Wir können sie nicht einfach entfernen, da uns das unter Umständen den gesamten Baum zerreissen könnte. Haben sie nur einen linken bzw. rechten Subtree (und nicht beide) können wir einfach diese an ihre Vorgängerknoten anstelle des zu löschenden Elements hängen. Haben sie aber zwei Nachfolger-Trees, geht das nicht zu leicht: Wir müssen ein Element hintauschen und danach unser Element aus dem getauschten Platz löschen. Dabei ist folgendes wichtig: Wir müssen nicht nur auf die AVL-Bedingung acht geben, sondern auch

aufpassen, dass der Tree immer noch ein BST bleibt. Wir machen das, indem wir das Element mit dem **symmetrischen Vorgänger (/Nachfolger)** austauschen: Dieses ist das grösste Element des linken Subtrees des zu löschenden Elements. (Das ist der Subtree in dem alle Elemente kleiner als unser zu löschendes Element sind). Denn so ist wieder garantiert, dass alle Elemente im linken Subtree kleinerals das Element an dieser Stelle sind (denn wir haben ja das grösste Element aus dem linken Subtree gewählt). Und trivial sind alle Elemente aus dem rechten Subtree grösser als das neu hingetauschte Element. Ausserdem wissen wir, dass dieser Symmetrische Vorgänger (der Nachfolger-Fall würde einfach umgekehrt ablaufen) sicherlich keinen rechten Nachfolger-Tree hat (da diese Elemente ja grösser sein müssten). Also lässt es sich dieser Knoten leicht aus dem Baum entfernen.

Hier ein Beispiel: (Bei Delete 2 verwenden wir den symmetrischen Nachfolger anstelle des symmetrischen Vorgängers).