Übung Algorithmen und Datenstrukturen

Sommersemester 2017

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda: Graphen, Suchbäume, AVL Bäume

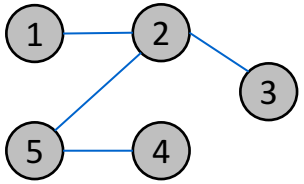
- **Heute:**
  - Graphen und Bäume
  - Binäre Suchbäume
  - AVL-Bäume

- Nächste Woche: Vorrechnen (first-come first-served)
  - Gruppe 5 13-15 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr5U5/
  - Gruppe 6 15-17 Uhr https://dudle.inf.tu-dresden.de/AlgoDatGr6U5/

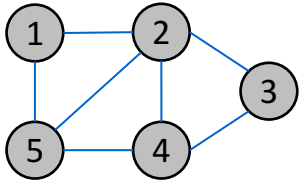
Übung: https://hu.berlin/algodat17
Vorlesung: https://hu.berlin/vl_algodat17

Graphen und Bäume

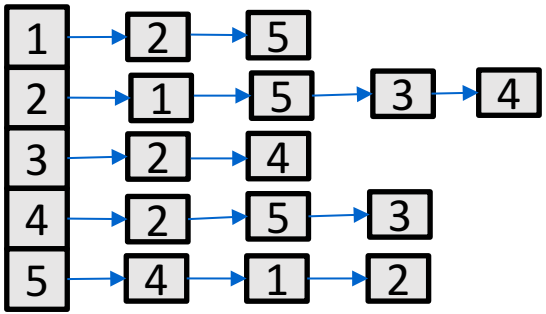
- Baum: Ein Baum ist ein ungerichteter, kreisfreier und zusammenhängender Graph
  1. Ein Graph ist ein Paar $G = (V, E)$ aus einer Menge Knoten V und Menge Kanten $E \subseteq V \times V$
  2. Für ungerichtete Graphen ist jedes Paar der Kantenmenge ungeordnet, d.h. : $(v, w) \Leftrightarrow (w, v)$
  3. Eine Pfad ist eine Folge von Knoten $v_1, v_2, \ldots, v_n$, in der für aufeinanderfolgende Knoten v_i, v_{i+1} über eine Kante verbunden sind $(v_i, v_{i+1}) \in E$.
  4. Ein Zyklus (Kreis) ist ein Pfad mit $v_1 = v_n$
  5. Ein Graph ist zusammenhängend, wenn für jedes Paar $(v, w) \in V \times V$ ein Pfad von v nach w existiert



Darstellung von Graphen



- Adjazenzliste:
  - Array A aus |V| Listen - Liste A[v] repräsentiert die zu Knoten v adjazenten Knoten
  - Bevorzugt für dünn besetzte Graphen: $|E| \ll |V|^2$
  - Zugriff auf eine bestimmte Kante: $\mathcal{O}(deg(v))$
  - Iterieren aller ausgehenden Kanten: $\mathcal{O}(deg(v))$



- Adjazenzmatrix:
  - $|V|x|V|$ Matrix A - Ein Eintrag $A[v][w] = (a_{vw})$ repräsentiert die Kante zu Knoten v und w
  - Bevorzugt für dichte Graphen: $|E| \approx |V|^2$
  - Zugriff auf eine bestimmte Kante: $\mathcal{O}(1)$
  - Iterieren aller ausgehenden Kanten: $\mathcal{O}(|V|)$

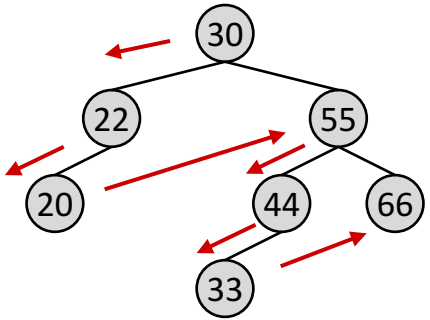
	1	**2**	**3**	**4**	**5**
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Graph-Traversierung

- Traversierung ist ein Verfahren, bei der jeder Knoten und jede Kante (einer Zusammenhangskomponente) genau einmal besucht werden
- Tiefensuche
  - Depth-first-search (DFS): traversiert in die Tiefe
  - Gefundene Knoten in LIFO-Reihenfolge (last-in first-out / Stack) verarbeiten
- Breitensuche (Dijkstra, Prim)
  - Breadth-first-search (BFS): traversiert in die Breite
  - Gefundene Knoten in FIFO-Reihenfolge (first-in first-out / Queue) verarbeiten

- Traversierung von Graphen in $\mathcal{O}(|V| + |E|)$

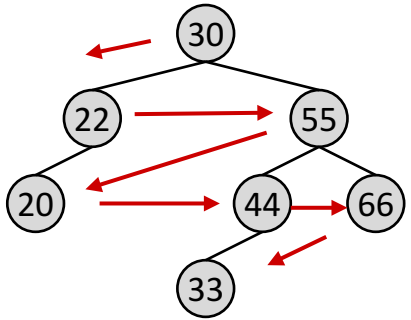
Tiefensuche und Breitensuche



Algorithmus DFS($start$)

Input: Knoten start

```
(1)   visit(start); start.visited = true;
(2)   foreach knoten in neighbor(start) do
(3)     if not knoten.visited then
(4)       DFS(knoten);
(5)     end if
(6)   end foreach
```



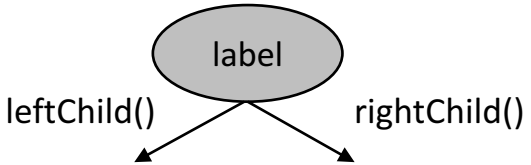
Algorithmus BFS($start$)

Input: Knoten start

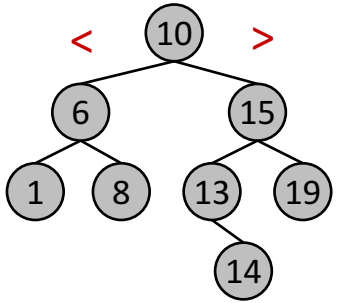
```
(1)   Queue q; q.enqueue(start);
(2)   while not q.isEmpty() do
(3)     start = q.dequeue();
(4)     visit(start );
(5)     foreach knoten in neighbor(start) do
(6)       if not knoten.visited then
(7)         knoten.visited = true;
(8)         q.enequeue(knoten);
(9)       end if
(10)    end foreach
(11)  endwhile
```

Binäre Suchbäume

- Ein binärer Suchbaum ist rekursiv definiert als:
  - „leer“ oder
  - enthält drei Knoten Mengen:
    1. Wurzel mit „Schlüssel“ (label),
    2. Linker binärer Suchbaum (leftChild) und
    3. Rechter binärer Suchbaum (rightChild)



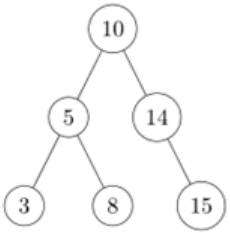
- Sortierung/Sucheigenschaft
  - Alle Schlüssel des linken Teilbaums kleiner als der Schlüssel der Wurzel
  - Alle Schlüssel des rechten Teilbaums sind größer als der Schlüssel der Wurzel



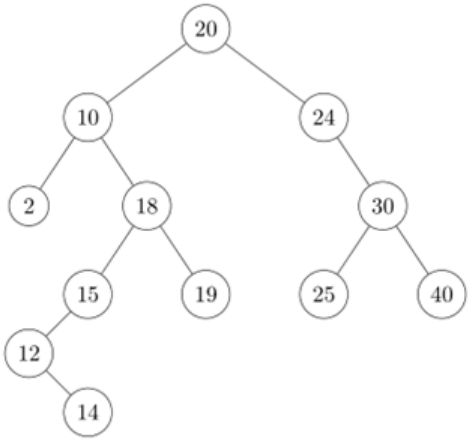
Aufgabe zu Binären Suchbäumen

In dieser Aufgabe sollen Sie einen Schreibtischtest mit binären Suchbäumen ausführen.

a) Führen Sie nun die Operationen `insert(16)`, `insert(11)` und `delete(8)` in dieser Reihenfolge aus. Geben Sie nach jeder Operation den neuen Baum an.

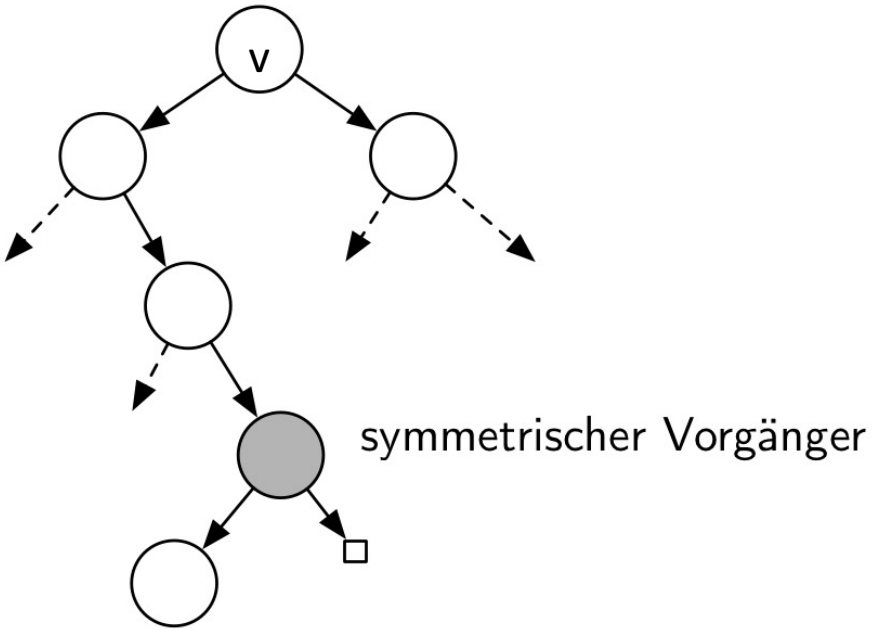


b) Führen Sie in dem folgenden Suchbaum die Operationen `delete(24)` und `delete(10)` in dieser Reihenfolge aus. Geben Sie nach jeder Operation den neuen Baum an.



Symmetrischer Vorgänger

Der symmetrische Vorgänger von Knoten v ist der Knoten mit dem größten Schlüsselwert kleiner als v. Kann ohne einen einzigen Schlüsselvergleich implementiert werden.



Symmetrischer Vorgänger: iterativ und rekursiv

Algorithmus $symmPredecessor\ (v)$

Input: Node v

```
(1)  if (v.leftChild()!= null) then
(2)    return treeMaximum(v.leftChild());
(3)  endif
(4)  return null;
```

Algorithmus $treeMaximum(v)$

Input: Node v

```
(1)  while(v.rightChild() != null) do
(2)    v = v.rightChild();
(3)  endwhile
(4)  return label(v);
```

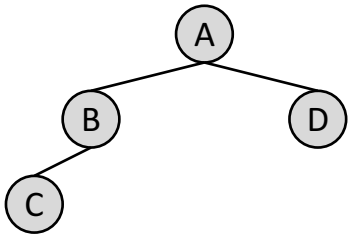
Algorithmus $treeMaximum(v)$

Input: Node v

```
(1)  if (v.rightChild() != null) then
(2)    return treeMaxmimum(v.rightChild());
(3)  endif
(4)  return label(v);
```

Tiefensuche auf binären Suchbäumen

- Es gibt drei Varianten der Tiefensuche auf binären Suchbäumen:
  - Preorder: Wurzel, Linker TB, Rechter TB
  - Inorder: Linker TB, Wurzel, Rechter TB
  - Postorder: Linker TB, Rechter TB, Wurzel



Preorder: **A (B C) (D)**
Inorder: **(C B) A (D)**
Postorder: **(C B) (D) A**

- Komplexität: $\mathcal{O}(|V|)$

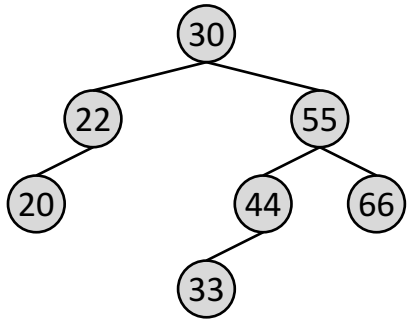
Algorithmus travers(v)

Input: Node v

```
(1)  if (v != null) then
(2)    print label(v);  // preorder
(3)    travers(v.leftChild());
(4)    print label(v);  // inorder
(5)    travers(v.rightChild());
(6)    print label(v);  // postorder
(7)  endif
```

Baumtraversierung

Gegeben sei der nachfolgende Baum. Geben Sie die Schlüssel in Inorder- und Preorder-Reihenfolge an.



Preorder: **30, 22, 20, 55, 44, 33, 66**

Inorder: **20, 22, 30, 33, 44, 55, 66**

- Inorder-Traversierung eines binären Suchbaums liefert die Schlüssel in aufsteigender (sortierter) Reihenfolge!

Programmieraufgabe: BinTree

- Die Klasse *BinTree* enthält einen Zeiger auf den Wurzelknoten *root*.
- Jeder Knoten *Node* enthält das zu speichernde Element *element* sowie Zeiger *parent*, *left* und *right* auf den Elternknoten, den linken Kindknoten und den rechten Kindknoten.
- Die Klasse verwendet generische Typen, die das Comparable-Interface implementieren (z.B. String, Integer).

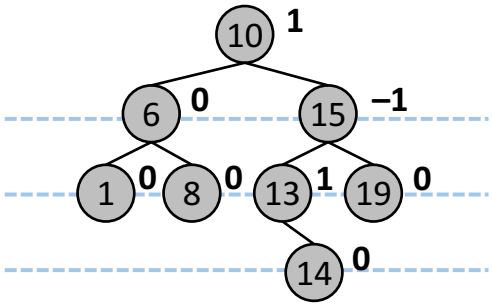
```
public class BinTree
          <E extends Comparable<E>> {
  // Die Wurzel des binären Suchbaums
  private Node root;

  public class Node
          implements Comparable<E> {
    E element;
    Node left;
    Node right;
    Node parent;
  }

  public E maximum() throws EmptyTreeException {
    return maximum(this.root).element;
  }
  private Node maximum(Node node) {
    if (node.right != null) {
      return maximum(node.right);
    }
    return node;
  }
```

AVL-Bäume

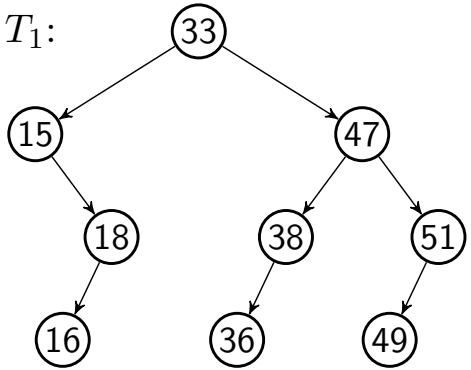
- Problem: Komplexität von Such-/Einfüge-/Lösch-Operationen abhängig von der Höhe des binären Suchbaums – entarteter Baum maximal $\mathcal{O}(|V|)$
- Lösung: Höhen-Balancierte Suchbäume garantieren logarithmische Höhe und damit Komplexität $\mathcal{O}(\log|V|)$
- AVL-Baum: In jedem Knoten unterscheidet sich die Höhe der beiden Teilbäume um höchstens Eins.
- Dafür ist Rebalancierung (Rotation) beim Einfügen und Löschen notwendig
- Definition:
  - Sei u Knoten in binärem Baum.
  - bal(u) : Differenz zwischen Höhe des rechten Teilbaums von u und Höhe des linkes Teilbaums von u
  - Ein binärer Baum heißt AVL-Baum, falls für alle Knoten u gilt: $|bal(u)| \leq 1$



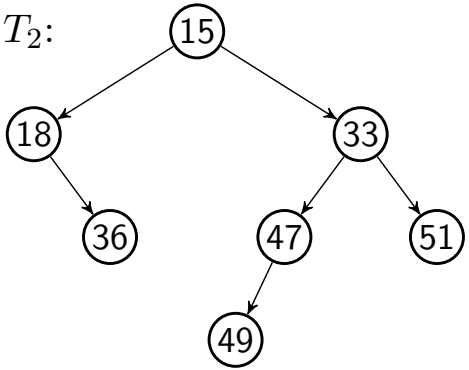
AVL Bäume, Heaps, Binäre Suchbäume

Geben Sie zu jedem der folgenden drei Bäume T_1, T_2, T_3 an, ob es sich um einen binären Min-Heap, binären Max-Heap, binären Suchbaum oder AVL-Baum handelt (mehreres kann zutreffen). Füllen Sie dazu die folgende Tabelle aus, indem Sie „ja“ oder „nein“ eintragen.

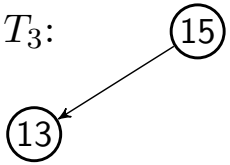
T_1:



T_2:

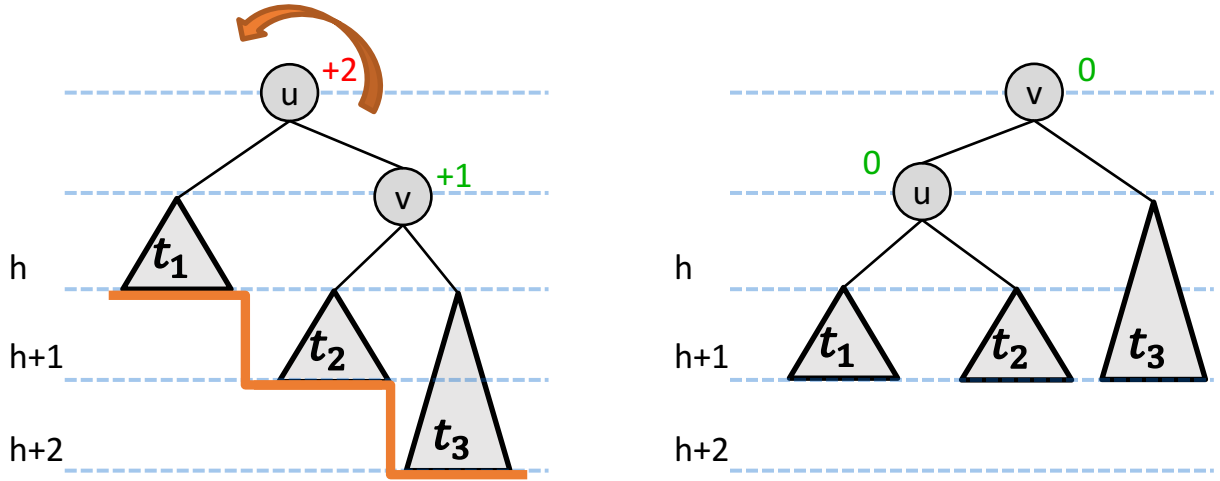


T_3:



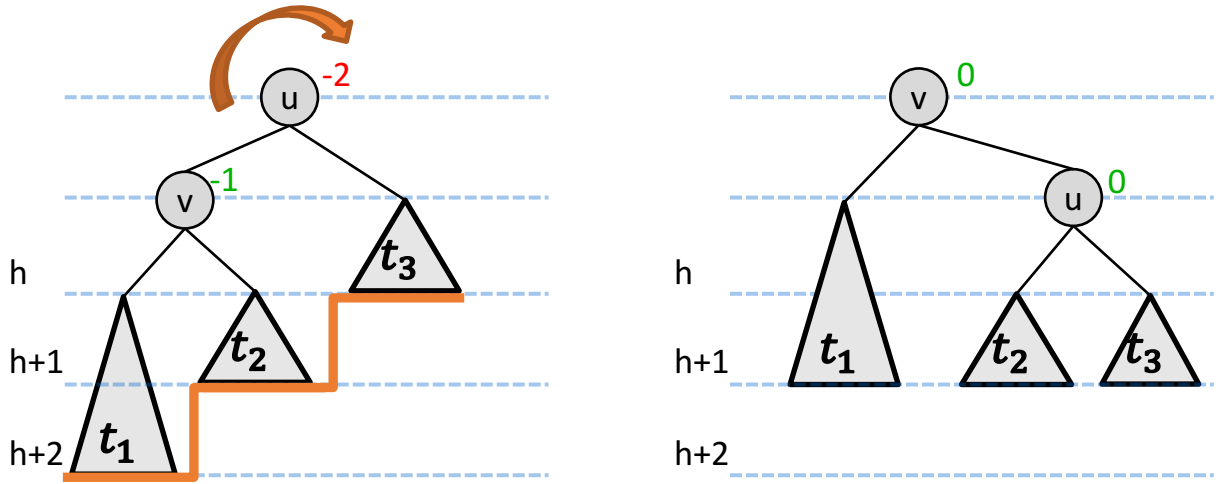
Rebalancierung von AVL-Bäumen

- Sei *u* Knoten, *v* Kind von *u* im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(*u*) = 2, bal(*v*) = 1: Einfachrotation Links(*u*)



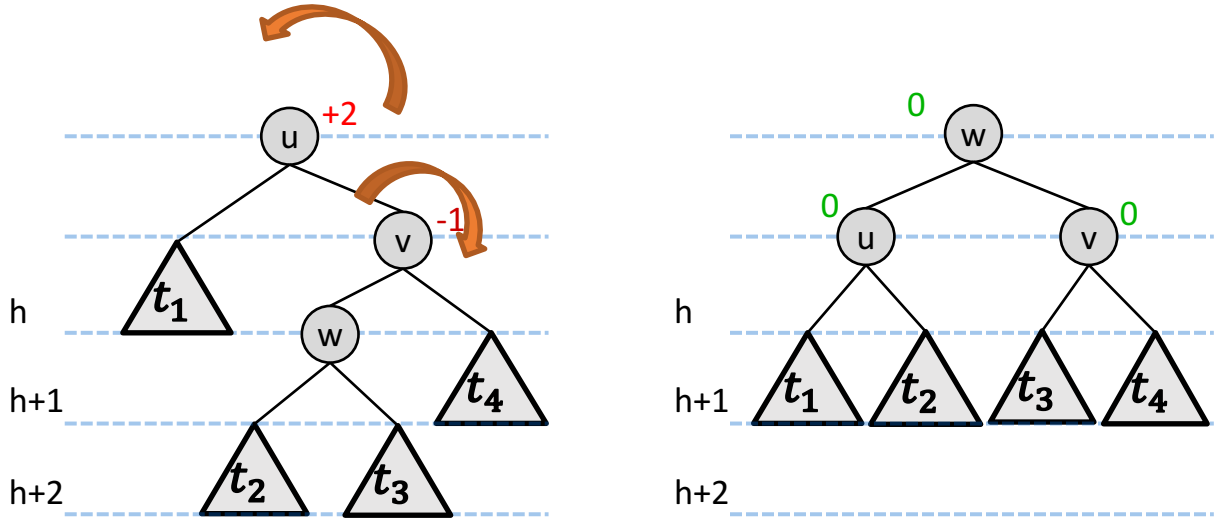
Rebalancierung von AVL-Bäumen

- Sei *u* Knoten, *v* Kind von *u* im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(*u*) = 2, bal(*v*) = 1: Einfachrotation Links(*u*)
  2. bal(*u*) = -2, bal(*v*) = -1: Einfachrotation Rechts(*u*)



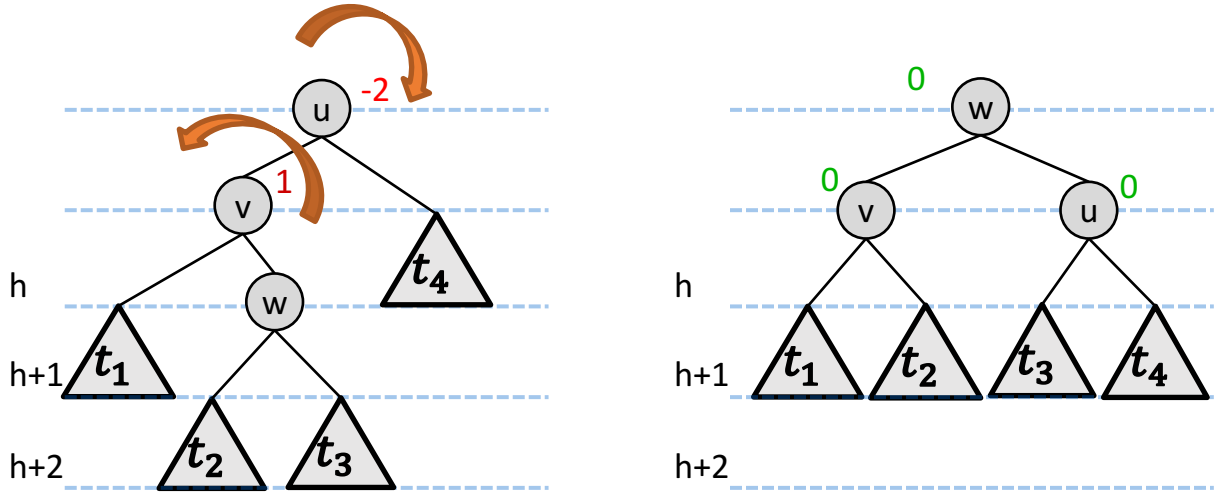
Rebalancierung von AVL-Bäumen

- Sei *u* Knoten, *v* Kind von *u* im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(*u*) = 2, bal(*v*) = 1: Einfachrotation Links(*u*)
  2. bal(*u*) = -2, bal(*v*) = -1: Einfachrotation Rechts(*u*)
  3. bal(*u*) = 2, bal(*v*) = -1: Doppelrotation Rechts(*v*) + Links(*u*)



Rebalancierung von AVL-Bäumen

- Sei *u* Knoten, *v* Kind von *u* im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(*u*) = 2, bal(*v*) = 1: Einfachrotation Links(*u*)
  2. bal(*u*) = -2, bal(*v*) = -1: Einfachrotation Rechts(*u*)
  3. bal(*u*) = 2, bal(*v*) = -1: Doppelrotation Rechts(*v*) + Links(*u*)
  4. bal(*u*) = -2, bal(*v*) = 1: Doppelrotation Links(*v*) + Rechts(*u*)



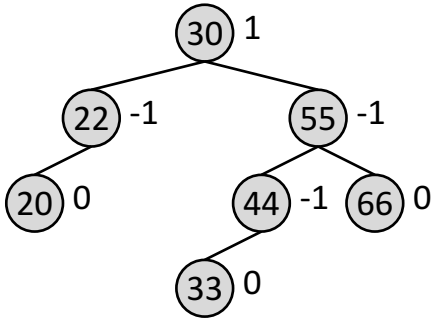
Rebalancierung von AVL-Bäumen

- Sei u Knoten, v Kind von u im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(u) = 2, bal(v) = 1: Einfachrotation Links(u)
  2. bal(u) = -2, bal(v) = -1: Einfachrotation Rechts(u)
  3. bal(u) = 2, bal(v) = -1: Doppelrotation Rechts(v) + Links(u)
  4. bal(u) = -2, bal(v) = 1: Doppelrotation Links(v) + Rechts(u)

- Komplexität: Beim Einfügen sind die Rotationen lokale Operationen, die nur Umsetzen einiger Zeiger erfordern, und in Zeit $\mathcal{O}(1)$ erfolgen.
- Aus logarithmischer Höhe des Baums ergibt sich Laufzeit $\mathcal{O}(\log n)$ für das Einfügen, Suchen und Löschen

Aufgabe: Einfügen in AVL-Bäumen

Sei *T* folgender AVL-Baum:



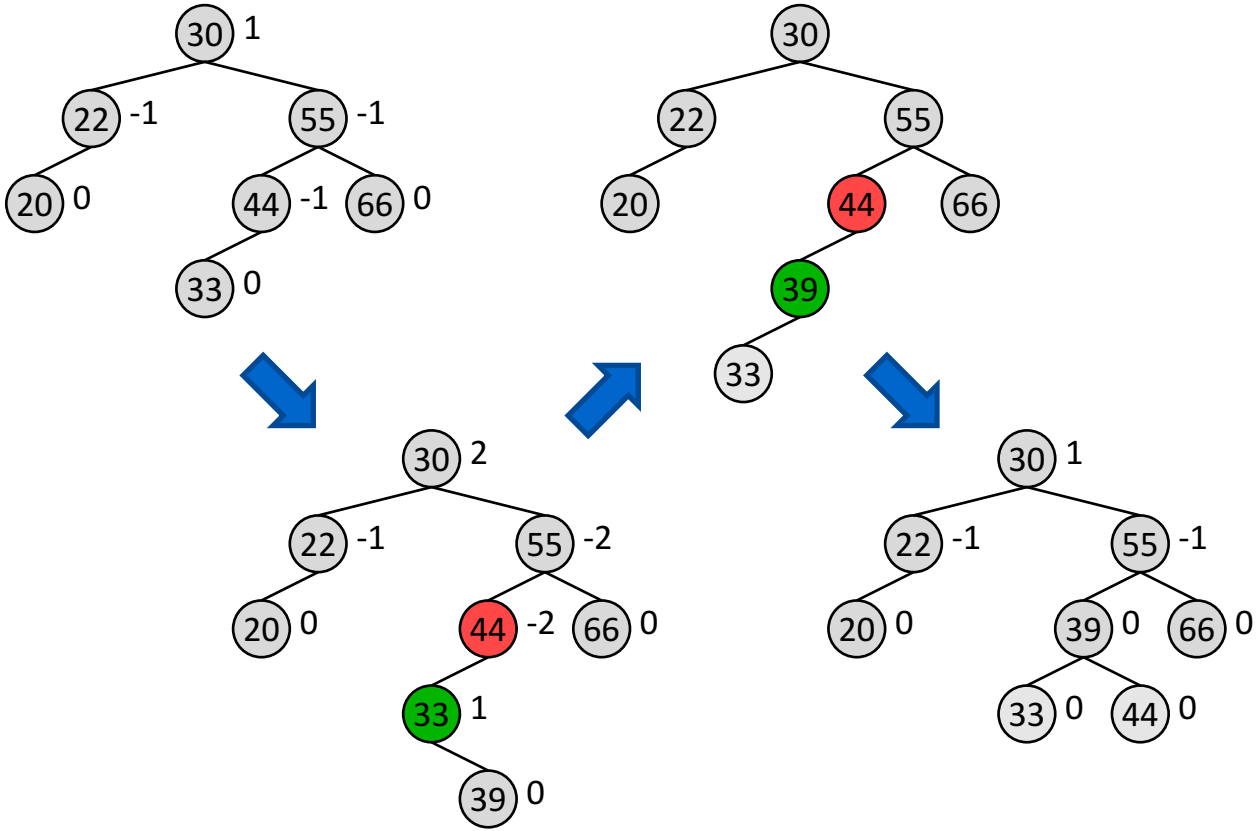
Fügen Sie nacheinander die Schlüssel

39, 42

in *T* ein und zeichnen Sie den jeweiligen AVL-Baum nach jeder insert-Operation.

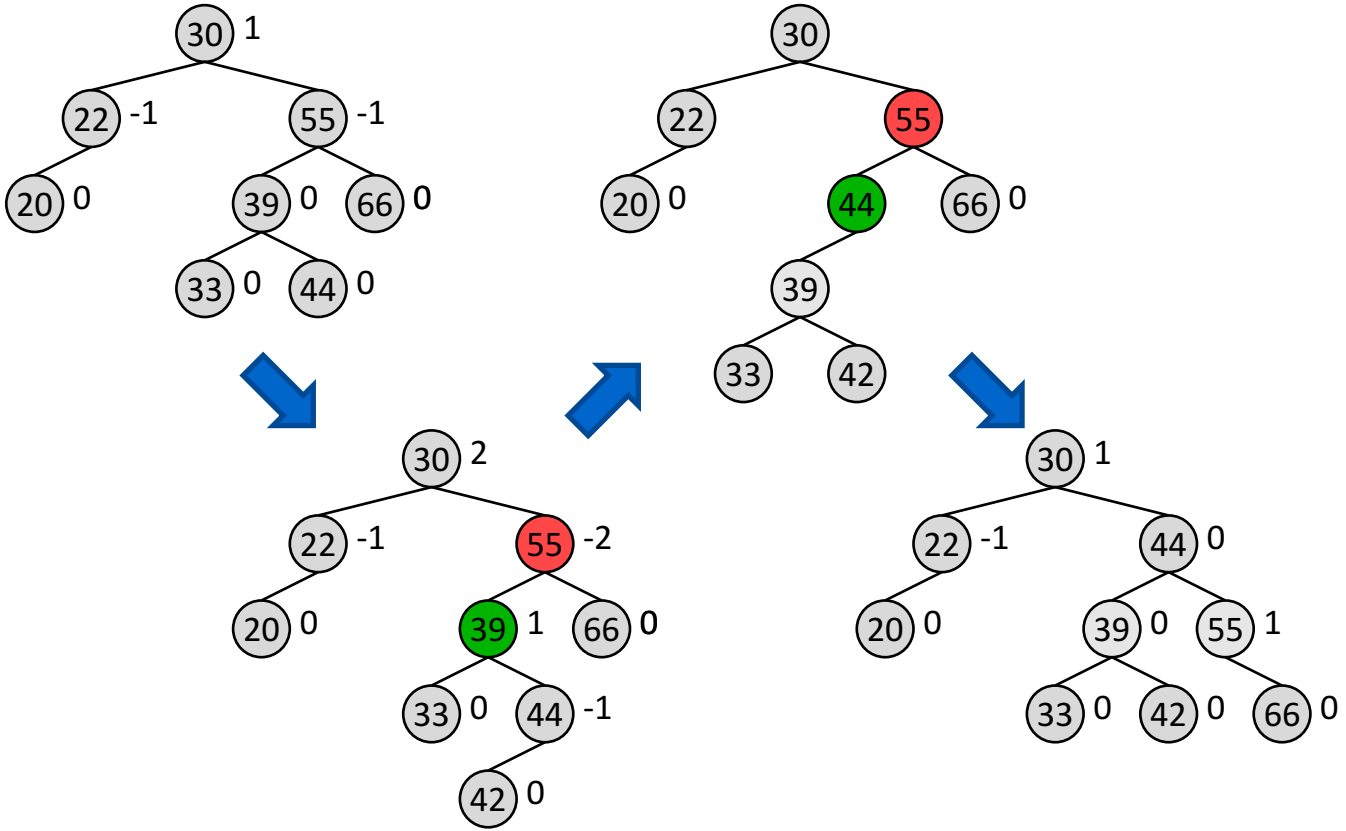
Einfügen von 39

bal(*44*) = -2, bal(*33*) = 1: Doppelrotation Links(*33*) + Rechts(*44*)



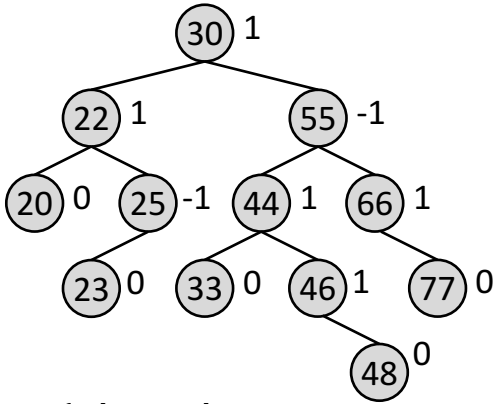
Einfügen von 42

bal(*55*) = -2, bal(*39*) = 1: Doppelrotation Links(*39*) + Rechts(*55*)



Aufgabe: Löschen in AVL-Bäumen

Sei *T* folgender AVL-Baum:



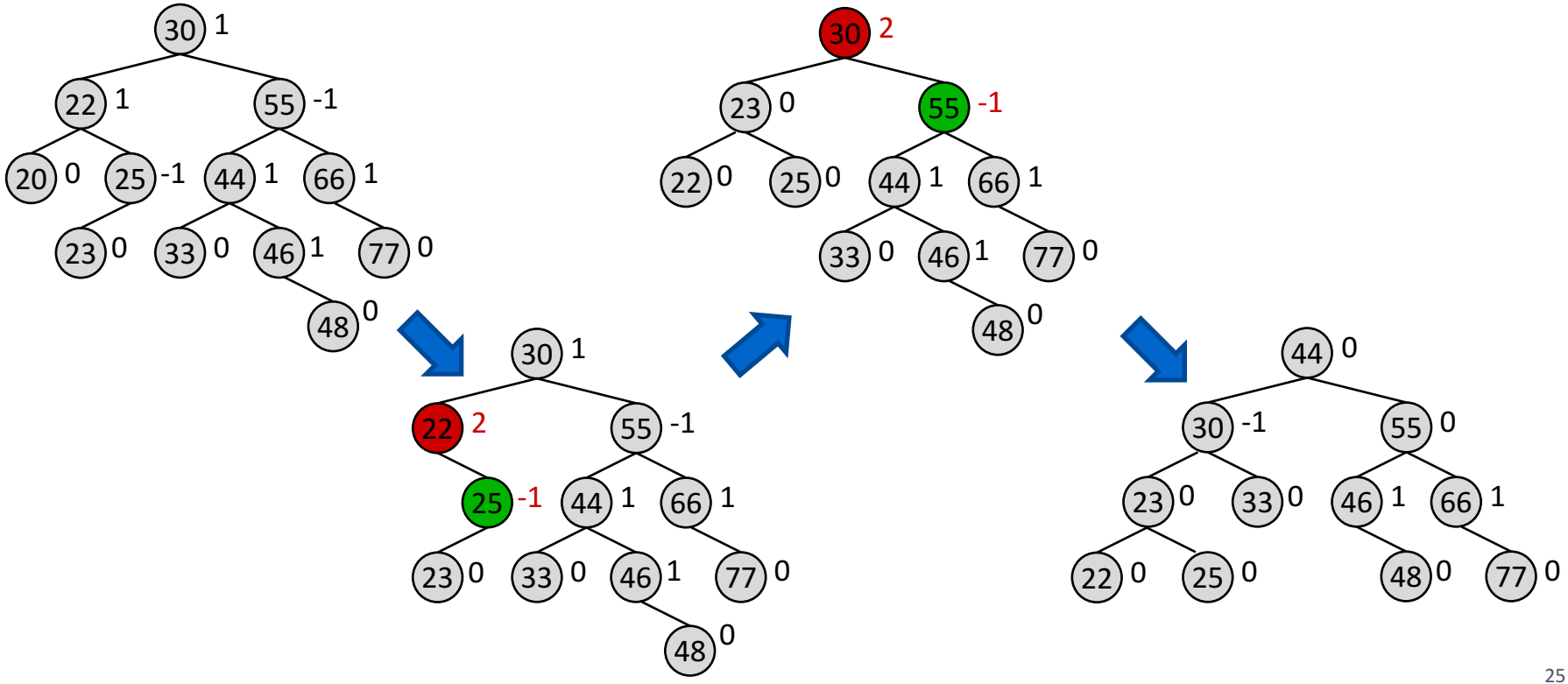
Löschen Sie folgenden Knoten

20

aus *T* und zeichnen Sie den entstehenden AVL-Baum.

Löschen in AVL-Bäumen

1. bal(*22*) = 2, bal(*25*) = -1: Doppelrotation Rechts(*25*) + Links(*22*)
2. bal(*30*) = 2, bal(*55*) = -1: Doppelrotation Rechts(*55*) + Links(*30*)



Löschen In AVL-Bäumen

- Verwendet die gleichen 4 Rotationsoperationen:
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. $bal(u) = 2, \quad bal(v) \in \{\mathbf{0}, \mathbf{1}\}$: Einfachrotation Links($u$)
  2. $bal(u) = -2, \quad bal(v) \in \{-\mathbf{1}, \mathbf{0}\}$: Einfachrotation Rechts(u)
  3. $bal(u) = 2, \quad bal(v) = -1$: Doppelrotation Rechts($v$) + Links($u$)
  4. $bal(u) = -2, bal(v) = 1$: Doppelrotation Links(v) + Rechts(u)

- Aber:
  - Es gibt Fälle, wo jeder Knoten entlang des Suchpfades rotiert werden muss.
  - Löschen hat Komplexität $\mathcal{O}(\log n)$

Wichtige Datenstrukturen

Datentyp	Wichtige Operationen	Besonderheit	Java-Klassen (Auswahl)
Array	A[i] |A|	**Indexbasierter Zugriff** in $O(1)$, Feste Länge.	[]
Liste	add, contains, delete, length	**Dynamische Datenstruktur**, Hohe Kosten für indexbasierten Zugriff / Suche	LinkedList, ArrayList
Stack	push, pop, top, isEmpty	Zugriff auf zuletzt eingefügtes Element in O(1), **keine Längenoperation**	Stack
Queue	enqueue, dequeue, head, isEmpty	Zugriff auf zuerst eingefügtes Element in O(1), **keine Längenoperation**	LinkedList
Heap	add, getMin, deleteMin, merge, create, size	Zugriff auf **Min / Max** in O(1), Entfernen von Min / Max in O(log n)	PriorityQueue
Hash-Tabelle	put, get, contains, size	**Konstante Laufzeit** nur im Average Case	HashMap, HashSet
Balancierter Suchbaum (AVL)	search, insert, delete, label, leftChild, rightChild	Höhenbalanciert O(log n)	TreeMap, TreeSet