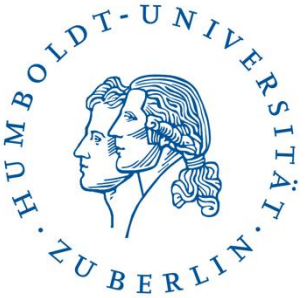
Übung Algorithmen und Datenstrukturen



Sommersemester 2016

Kim Völlinger

Humboldt-Universität zu Berlin

Gegeben sei eine Hashtabelle mit 11 Feldern für Einträge (Index 0 bis 10, siehe Konventionen) und eine Hashfunktion $h(x) = x \bmod 11$. Führen Sie für die folgenden Hashverfahren einen Schreibtischtest durch, indem Sie jeweils die Werte 19, 2, 14, 41, 33, 47 und 12 in dieser Reihenfolge in eine leere Tabelle einfügen und diese nach jeder Einfügeoperation ausgeben.

a) **Hashing mit direkter Listenverkettung**
Hierbei ist die Hashtabelle als Array von einfach verketteten Listen realisiert.

b) **Offenes Hashing mit linearem Sondieren**
Bei Kollisionen wird hier das einzufügende Element an der nächsten freien Stelle links vom berechneten Hashwert eingefügt. Das heißt, die Sondierungsreihenfolge (die Reihenfolge, in der die Plätze im Array durchgegangen werden, bis erstmals ein freier Platz angetroffen wird) ist gegeben durch $s(x, i) = (x - i) \bmod 11$ für $i = 0, \ldots, 10$.

c) **Doppeltes Hashing**
Das doppelte Hashing ist ein offenes Hashing, bei dem die Sondierungsreihenfolge von einer zweiten Hashfunktion $h'(x) = 1 + (x \bmod 7)$ abhängt. Die Position für das i-te Sondieren ist bestimmt durch die Funktion $s(x, i) = (h(x) - i \cdot h'(x)) \bmod 11$ für $i = 0, \ldots, 10$.

e) **Uniformes offenes Hashing**

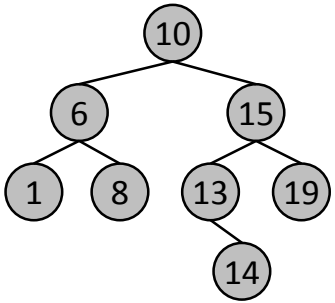
Hier erhält jeder Schlüssel mit gleicher Wahrscheinlichkeit eine der 11! Permutationen von $\{0, 1, \ldots, 10\}$ als Sondierungsreihenfolge zugeordnet.

Als „zufällige“ Permutationen nutzen Sie:

$$L(19) = 2, 10, 9, 3, 0, 8, 7, 6, 1, 4, 5 \qquad L(2) = 1, 5, 9, 3, 7, 10, 4, 8, 6, 2, 0$$
$$L(14) = 0, 8, 5, 2, 1, 10, 7, 9, 6, 3, 4 \qquad L(41) = 1, 4, 10, 5, 2, 0, 7, 3, 6, 9, 8$$
$$L(33) = 6, 8, 4, 0, 9, 1, 5, 2, 10, 3, 7 \qquad L(47) = 4, 2, 3, 5, 1, 8, 9, 0, 7, 6, 10$$
$$L(12) = 1, 3, 6, 0, 4, 5, 2, 10, 8, 7, 9$$

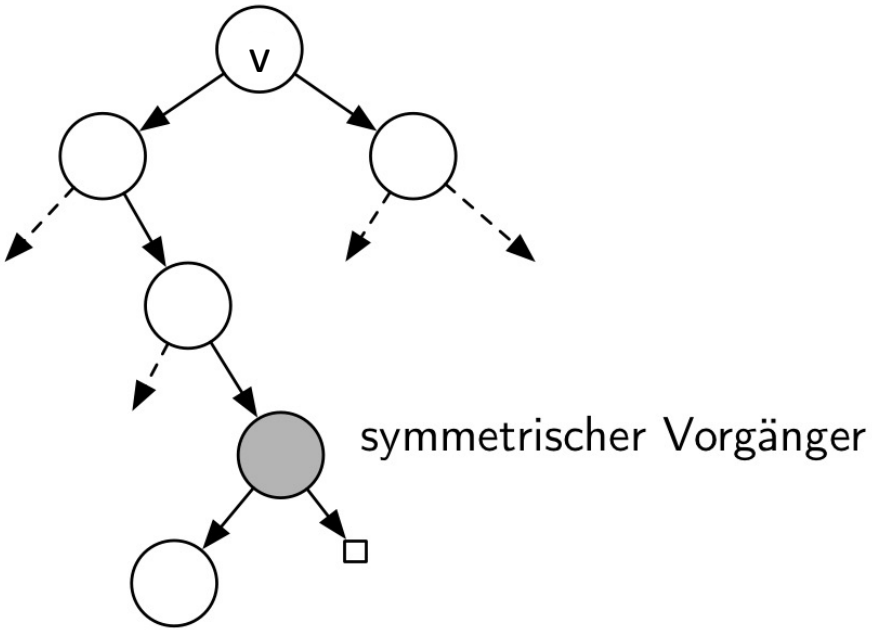
Binäre Suchbäume

- Knoten
  - beinhaltet „Schlüssel“ (label(v))
  - hat Verweis auf linkes (v.leftChild()) und rechtes Kind (v.rightChild())
- Sortierung/Sucheigenschaft
  - Schlüssel des linken Teilbaums eines Knotens sind kleiner als Schlüssel des Knotens selbst
  - Schlüssel des rechten Teilbaums eines Knotens sind größer als Schlüssel des Knotens selbst
  - betrachten nur binäre Suchbäume ohne Duplikate



Symmetrischer Vorgänger

Der symmetrische Vorgänger ist der Knoten mit dem größten Schlüsselwert kleiner als v. Er kann gefunden werden, ohne einen einzigen Vergleich der Schlüsselwerte durchzuführen.



Symmetrischer Vorgänger

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

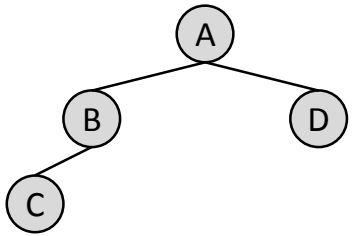
```
(1)  if (v.rightChild() != null) then
(2)    return treeMaxmimum(v.rightChild());
(3)  endif
(4)  return label(v)
```

Baumtraversierung

- Häufig müssen alle Knoten eines Baumes besucht werden, um bestimmte Operationen auf ihnen durchführen zu können.
- Traversierungsarten
  - Methoden unterscheiden sich in der Reihenfolge, in der Knoten besucht werden.

- Preorder: Wurzel, Linker TB, Rechter TB: **A, B, C, D**
- Inorder: Linker TB, Wurzel, Rechter TB: **C, B, A, D**
- Postorder: Linker TB, Rechter TB, Wurzel: **C, B, D, A**

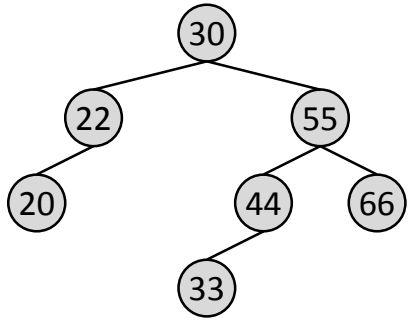
- Komplexität: $O(|V|)$



Baumtraversierung

Gegeben Sei der nachfolgende Baum.

Gib die Schlüssel in Inorder- und Preorder-Reihenfolge an.



Baumtraversierung

- Inorder-/Preorder-/Postorder-Traversierung können elegant rekursiv ausgedrückt werden.

Algorithmus $preorder(v)$

Input: Node v

```
(1)  if (v!= null) then
(2)    print label(v);
(3)    preorder(v.leftchild());
(4)    preorder(v.rightchild());
(5)  endif
```

Algorithmus $inorder(v)$

Input: Node v

```
(1)  if (v!= null) then
(2)    inorder(v.leftchild());
(3)    print label(v);
(4)    inorder(v.rightchild());
(5)  endif
```

Algorithmus $postorder(v)$

Input: Node v

```
(1)  if (v!= null) then
(2)    postorder(v.leftchild());
(3)    postorder(v.rightchild());
(4)    print label(v);
(5)  endif
```

Preorder-Traversierung: Iterativ & Rekursiv

- Zum Vergleich ist der iterative Algorithmus der Preorder-Traversierung deutlich schlechter lesbar.

Algorithmus $preorder(v)$

Input: Node v

```
(1)  if (v == null) then return endif;
(2)  Stack<Node> stack;
(3)  stack.push(v)
(4)  while (NOT stack.empty()) do
(5)    v = stack.pop();
(6)    print label(v);
(7)    if (v.rightChild() != null) then v = v.rightChild() endif
(8)    if (v.leftChild() != null) then v = v.leftChild() endif
(9)  end while
(10) return list;
```

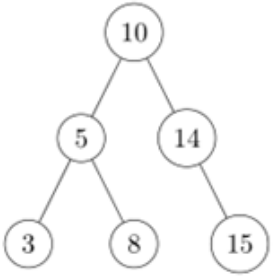
Algorithmus $preorder(v)$

Input: Node v

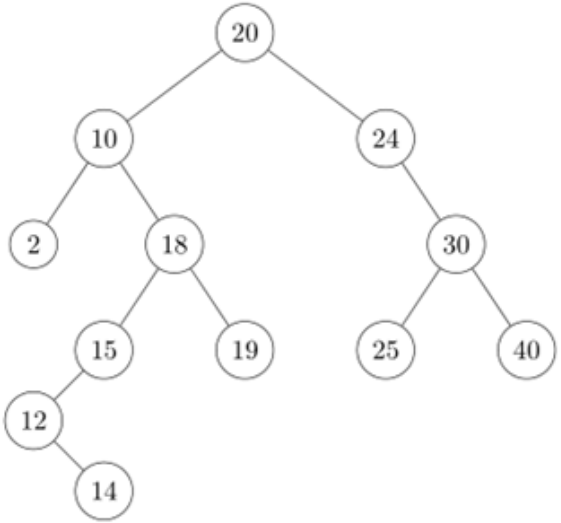
```
(1)  if (v!= null) then
(2)    print label(v);
(3)    preorder(v.leftchild());
(4)    preorder(v.rightchild());
(5)  endif
```

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.

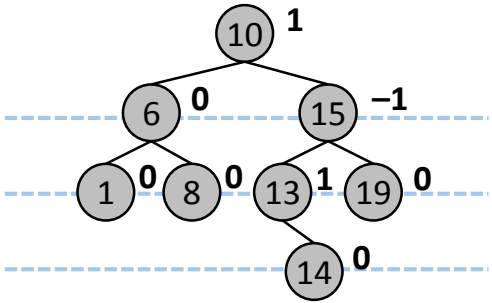


b) Führen Sie in dem folgenden Suchbaum die Operationen `delete(24)` und `delete(10)` in dieser Reihenfolge aus. Geben Sie nach jeder Operation den neuen Baum an.



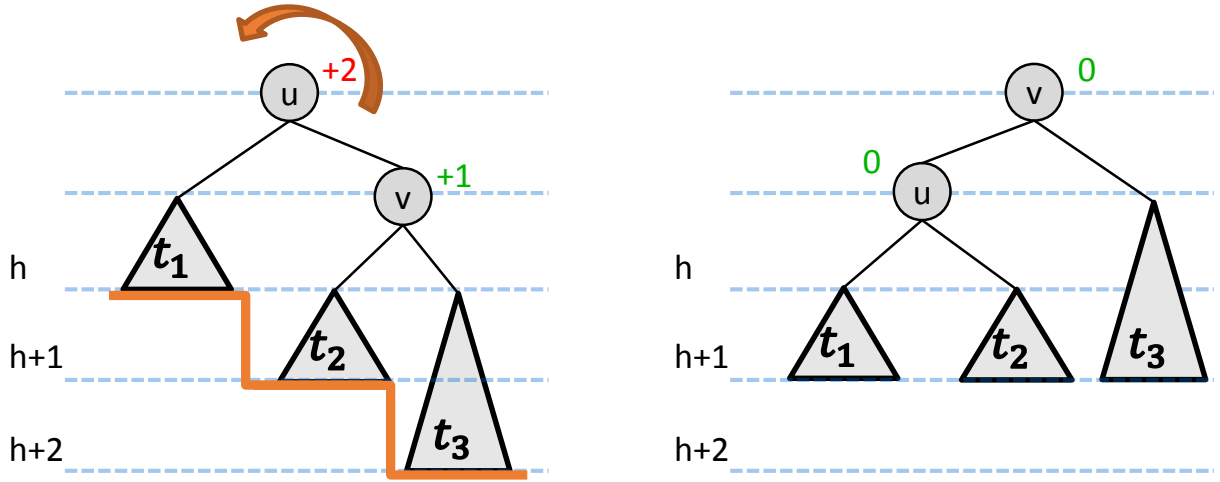
AVL-Bäume

- Problem: Komplexität von Such-/Einfüge-/Lösch-Operationen abhängig von der Höhe des binären Suchbaums - maximal $O(|V|)$.
- Lösung: Balancierte Suchbäume wie AVL-Baum garantieren logarithmische Höhe und damit Komplexität $O(\log|V|)$.
- AVL-Baum: In jedem Knoten unterscheidet sich die Höhe der beiden Teilbäume um höchstens Eins.
- Rebalancierung (Rotation) beim Einfügen und Löschen.
- 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*)| ≤ 1



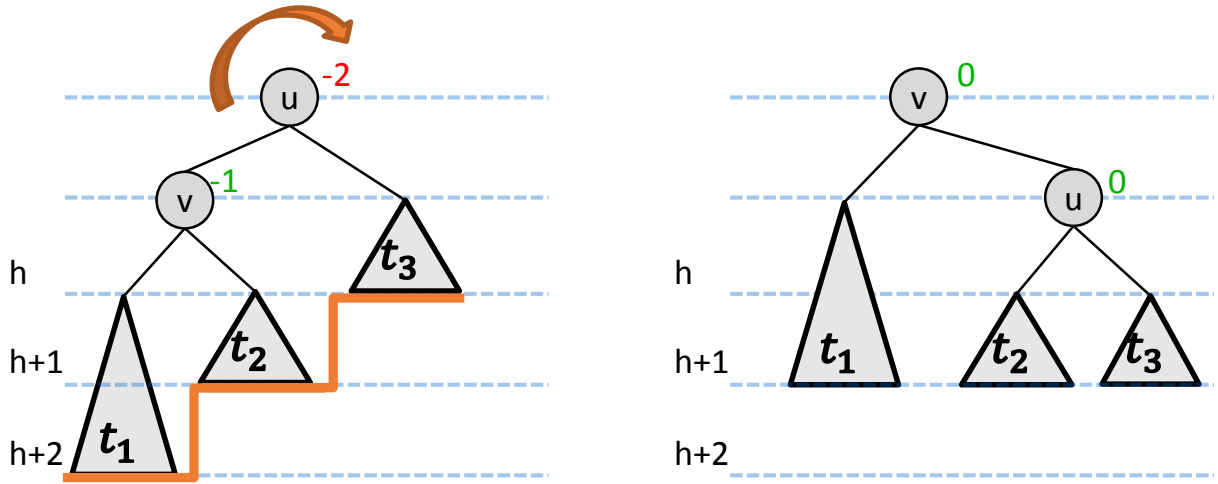
Rebalancierung von AVL-Bäumen

- 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*)



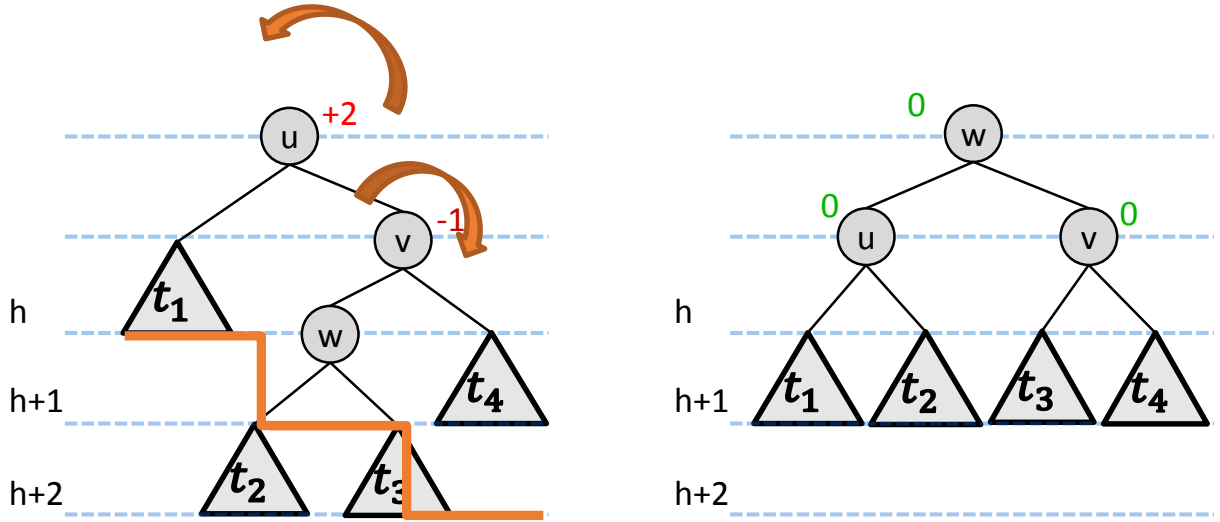
Rebalancierung von AVL-Bäumen

- 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*)



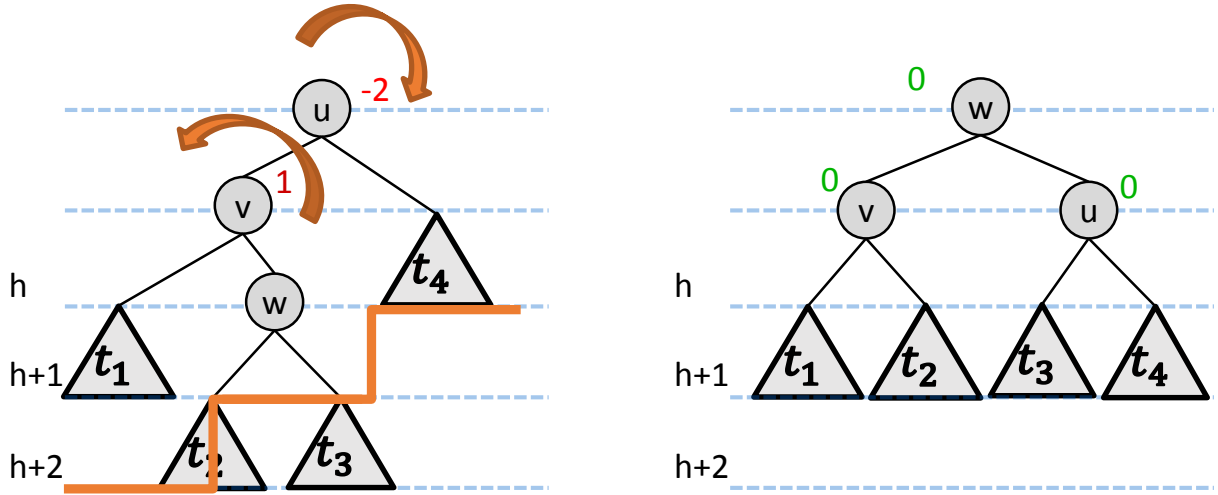
Rebalancierung von AVL-Bäumen

- 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

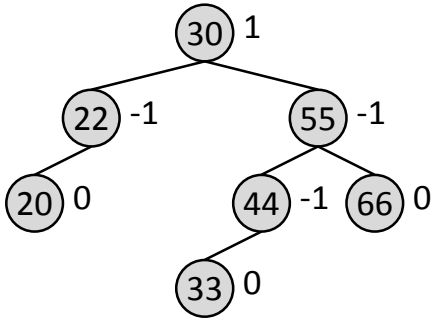
- Sei *u* Knoten, *v* Kind von *u* im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. bal(*u*) = 2, bal(*v*) = 1: Einfachrotation Links(*u*)
  2. bal(*u*) = -2, bal(*v*) = -1: Einfachrotation Rechts(*u*)
  3. bal(*u*) = 2, bal(*v*) = -1: Doppelrotation Rechts(*v*) + Links(*u*)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

- Komplexität: Rotationen sind lokale Operationen, die nur Umsetzen einiger Zeiger erfordern, und in Zeit $\mathcal{O}(1)$ erfolgen.

- Aufgabe: Sei T ein leerer AVL-Baum. Fügen Sie nacheinander die Elemente
  30, 20, 22, 55, 66, 44, 33
  ein

Aufgabe zu AVL-Bäumen

Sei *T* folgender AVL-Baum:



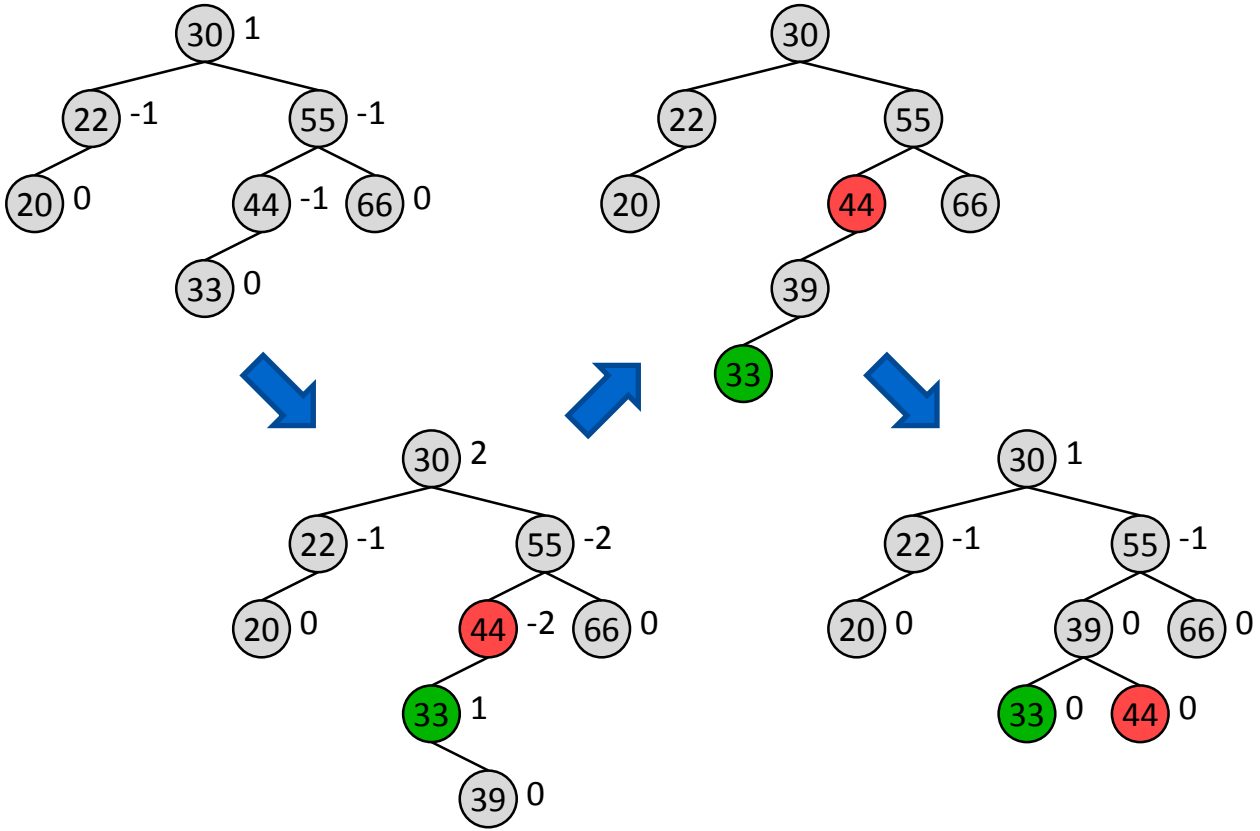
Fügen Sie nacheinander die Schlüssel

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*)