Übung Algorithmen und Datenstrukturen

Sommersemester 2016

Patrick Schäfer, Humboldt-Universität zu Berlin

Agenda

- Vorstellung des 6. Übungsblatts.
  - Hashing
  - Binäre Suchbäume
  - AVL-Bäume

Aufgabe: Hashing mit offener Adressierung

Sie sollen nun verschiedene Hashing Verfahren im Schreibtischtest anwenden. Es geht dabei nur darum, die Hashtabelle über eine Reihe von Einfügungen zu beobachten. Suchen oder Löschen wird nicht behandelt.
Verwenden Sie eine Hashtabelle mit 10 Plätzen (Index 0 bis 9) und fügen Sie in dieser Reihenfolge die folgenden Werte ein: 56, 34, 6, 13, 94, 27, 47. Geben Sie nach jedem Einfügeschritt die Hashtabelle an.

a) **Uniformes offenes Hashing.** Hier erhält jeder Schlüssel mit gleicher Wahrscheinlichkeit eine der 10! Permutationen von $\{0, 1, \ldots, 9\}$ als Sondierungsreihenfolge zugeordnet.
Als „zufällige“ Permutationen nutzen Sie:

$$h(56) = 2,9,3,0,8,7,6,1,4,5 \qquad h(34) = 1,5,9,3,7,4,8,6,2,0$$
$$h(6) = 0,8,5,2,1,7,9,6,3,4 \qquad h(13) = 1,4,5,2,0,7,3,6,9,8$$
$$h(94) = 6,8,4,0,9,1,5,2,3,7 \qquad h(27) = 4,2,3,5,1,8,9,0,7,6$$
$$h(47) = 1,3,6,0,4,5,2,8,7,9$$

b) **Offenes Hashing mit linearem Sondieren.** Hashfunktion $h(k) = k \mod 10$. 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 $s(k, j) = (h(k) - j) \mod 10$).

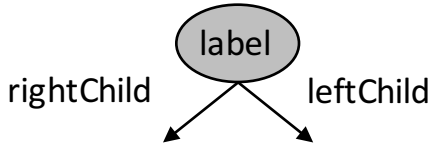
c) **Double Hashing.** Das doppelte Hashing ist ein offenes Hashing, bei dem die Sondierungsreihenfolge von einer zweiten Hashfunktion abhängt. Hashfunktion $h(k) = k \mod 10$, zweite Hashfunktion $h'(k) = 1 + (k \mod 8)$. Sondieren ist bestimmt durch die Funktion $s(k, j) = (h(k) - j \cdot h'(k)) \mod 10$.

Anmerkung:
(-1) mod 10
= (10-1) mod 10 = 9

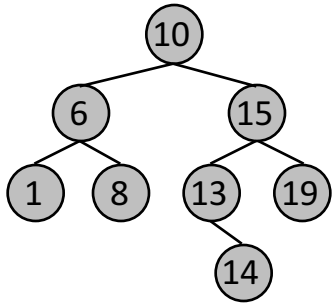
Binäre Suchbäume

- Knoten
  - beinhaltet „Schlüssel“ (label)
  - hat Verweis auf linkes (leftChild()) und rechtes Kind (rightChild())

```
class Tree {
  String label;
  Tree leftChild;
  Tree rightChild;
}
```

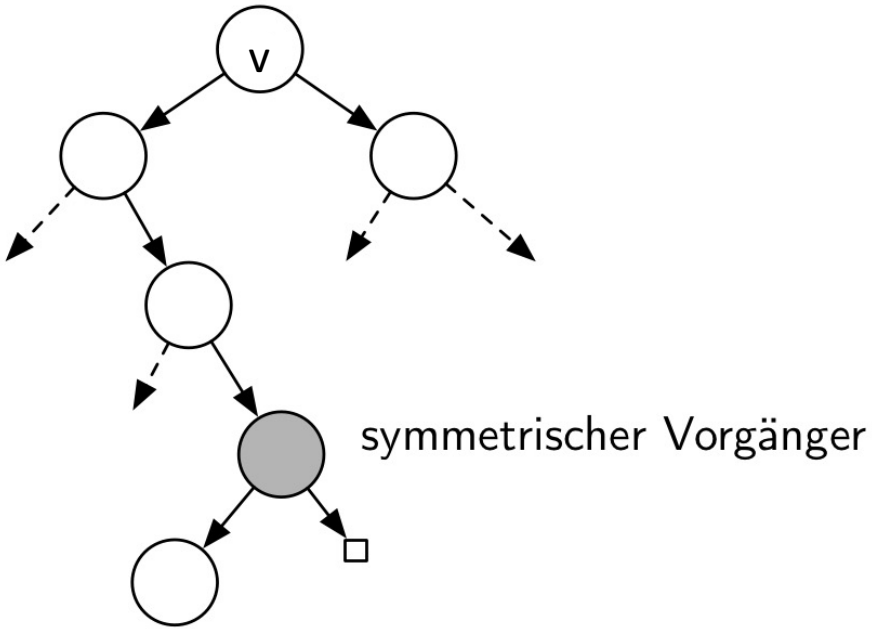


- Sortierung/Sucheigenschaft
  - Schlüssel des linken Teilbaums eines Knotens sind kleiner als Schlüssel des Knotens selbst
  - Schlüssel des rechten Teilbaums eines Knotens sind größer als Schlüssel des Knotens selbst
  - Gleiche Schlüssel entweder links oder rechts



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: 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

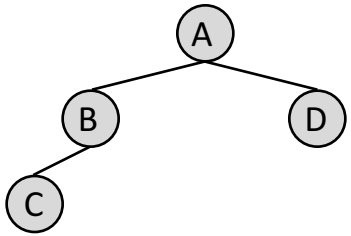
```
(1)  if (v.rightChild() != null) then
(2)    return treeMaxmimum(v.rightChild());
(3)  endif
(4)  return label(v)
```

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.

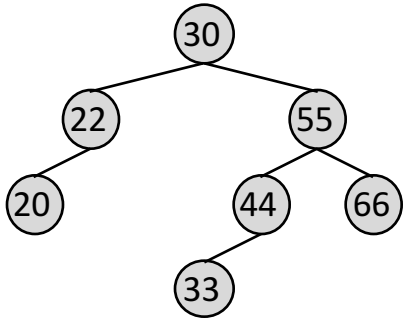
- Preorder: Wurzel, Linker TB, Rechter TB: **A, B, C, D**
- Inorder: Linker TB, Wurzel, Rechter TB: **C, B, A, D**
- Postorder: Linker TB, Rechter TB, Wurzel: **C, B, D, A**

- Komplexität: $O(|V|)$



Baumtraversierung

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



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

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 stack.push(v.rightChild()) endif
(8)     if (v.leftChild() != null) then stack.push(v.leftChild()) endif
(9)   end while
(10)  return list;
```

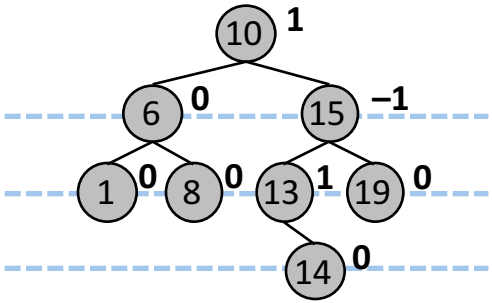
Algorithmus $preorder(v)$

Input: Node v

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

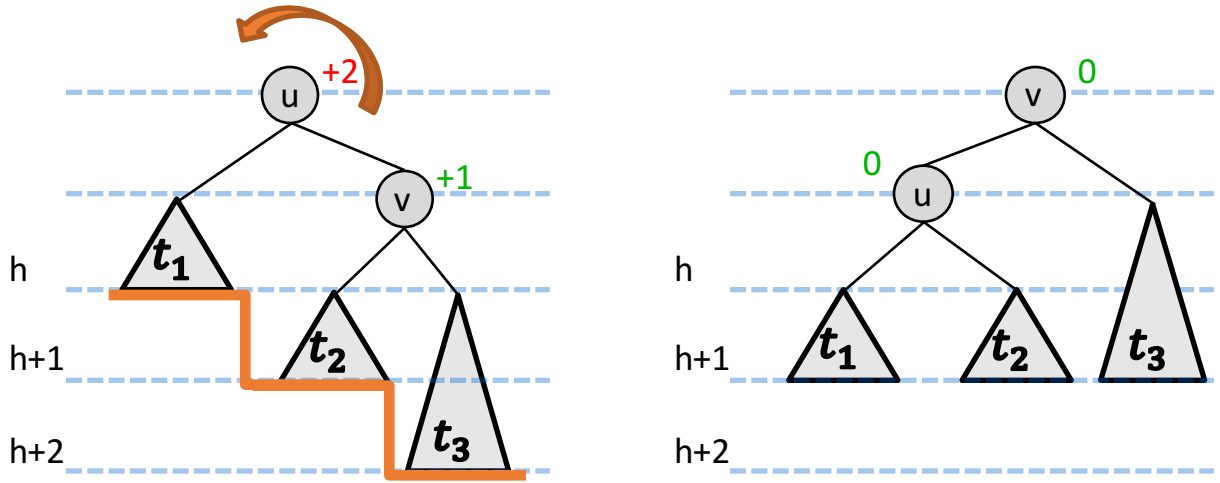
AVL-Bäume

- 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: $|\text{bal}(u)| \leq 1$



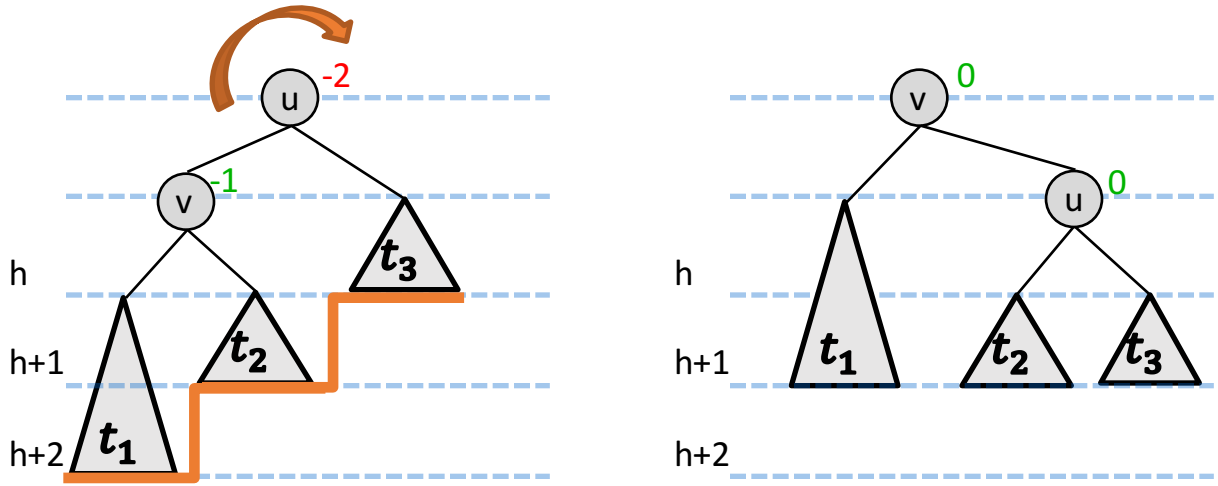
Rebalancierung von AVL-Bäumen

- 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*)



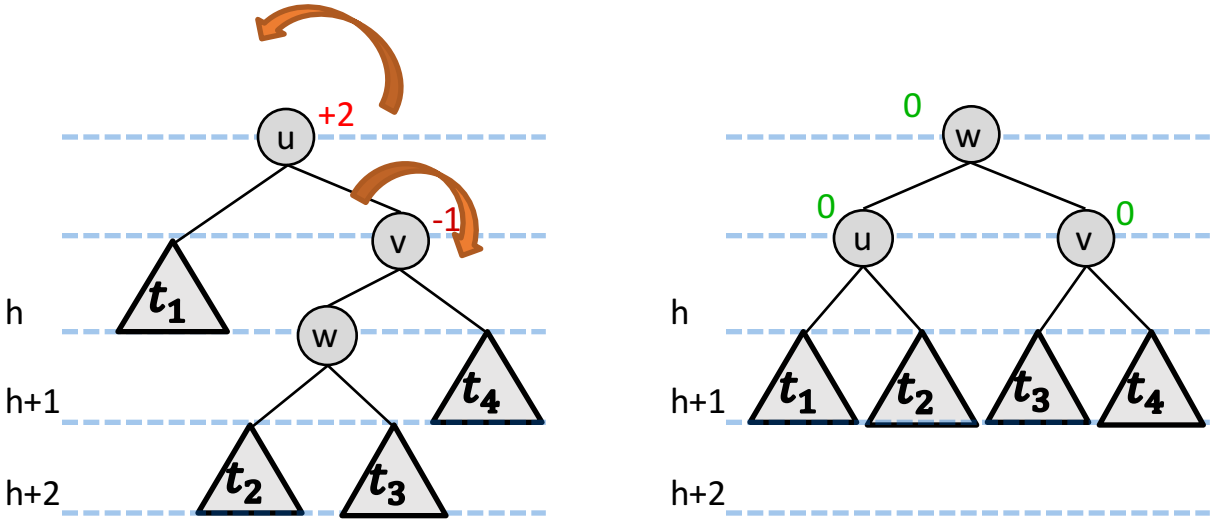
Rebalancierung von AVL-Bäumen

- 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*)



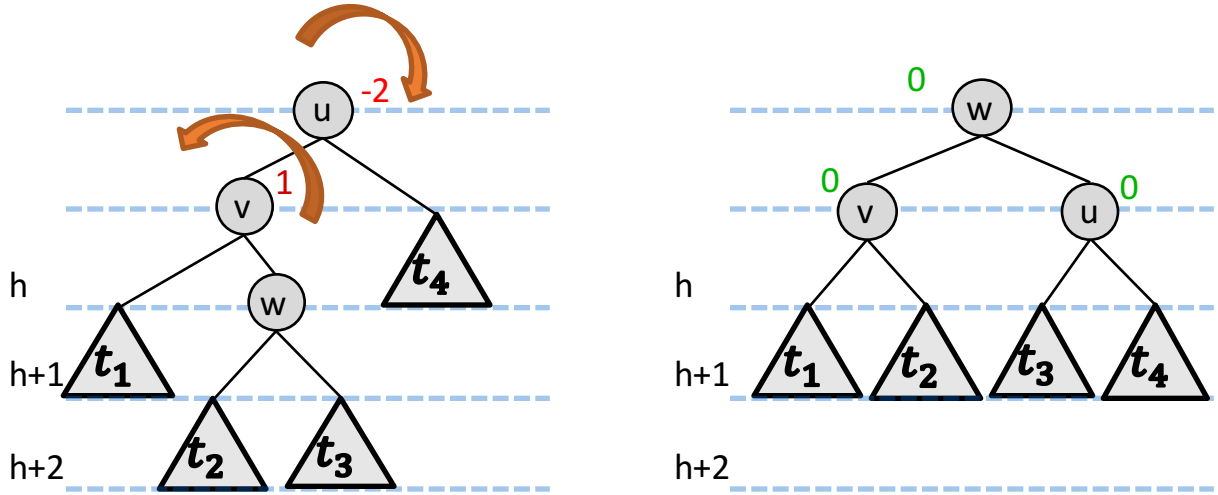
Rebalancierung von AVL-Bäumen

- 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

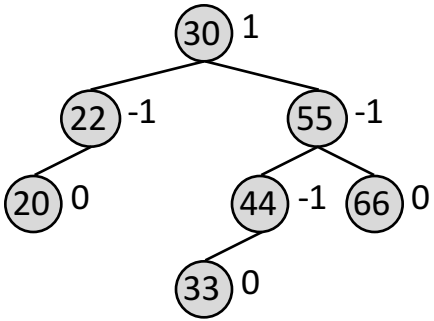
- Sei u Knoten, v Kind von u im Teilbaum mit größerer Höhe
- 4 Rotationsoperationen auf AVL-Bäumen:
  1. $\text{bal}(u) = 2$, $\text{bal}(v) = 1$: Einfachrotation $\text{Links}(u)$
  2. $\text{bal}(u) = -2$, $\text{bal}(v) = -1$: Einfachrotation $\text{Rechts}(u)$
  3. $\text{bal}(u) = 2$, $\text{bal}(v)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

- 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:



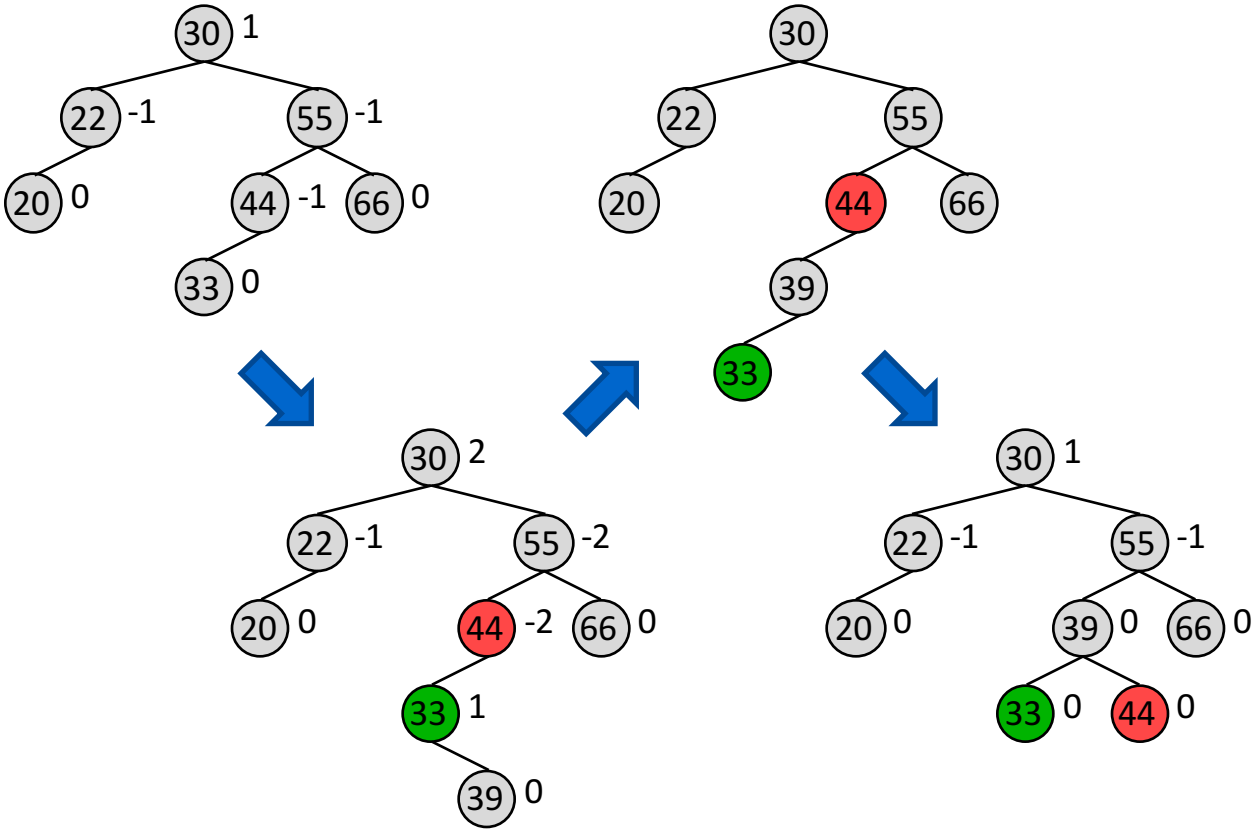
Fügen Sie nacheinander die Schlüssel

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*)