Algorithmen und Datenstrukturen

Datentyp Menge (*Forts.*):

Suchbäume ♦ Balancieren

Verwalten von Mengen durch Suchbäume

Suchbäume

- **Idee:** Verwalten der Elemente einer Menge in einem Baum, in den die Elemente in gewisser Weise geordnet eingefügt werden (**Suchbaum**)

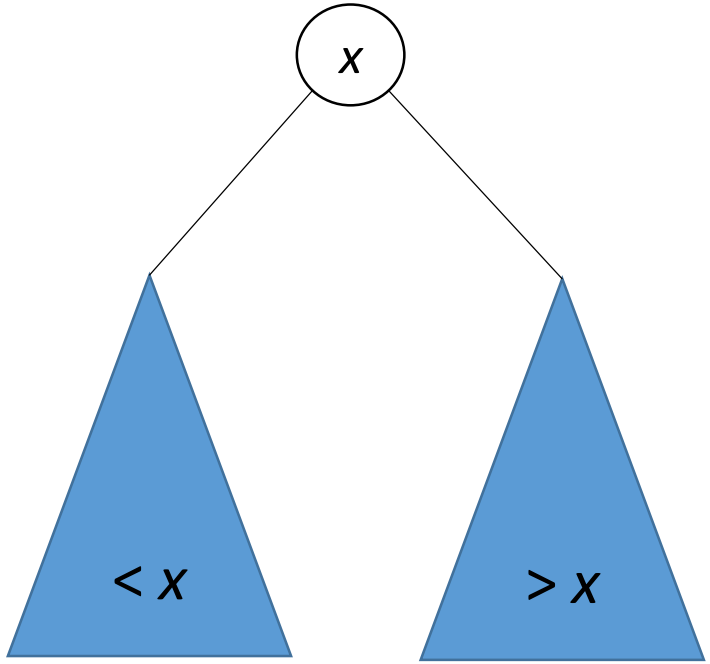
- **Voraussetzung:** Lineare Ordnung auf dem Grundbereich definiert (→ *Schlüssel verwenden)*

- Implementierung einer geeigneten Operation zum Einfügen eines Elements in einen geordneten Baum

Binärer Suchbaum

- Für jeden Knoten gilt:

1. Alle Werte in seinem linken Teilbaum sind kleiner als der Wert des Knotens.
2. Alle Werte in seinem rechten Teilbaum sind größer als der Wert des Knotens.



Funktion Suchen

Name: **contains(x,t)**

Eingabe: binärer Suchbaum *t* mit Wurzel *r*,
Wert *x* aus dem Grundbereich

Ausgabe: True, falls *x* Wert eines Knotens von *t* ist, False sonst

```
def contains(x,t):
    if isEmpty(t): return False
    if value(content(t)) == x: return True
    if value(content(t)) > x:
            return contains(x,left(content(t)))
    else: return contains(x,right(content(t)))
```

***O*(depth(*t*))**

Funktion Einfügen

Name: **add(*x*,*t*)**

Eingabe: binärer Suchbaum *t* mit Wurzel *r*,
Wert *x* aus dem Grundbereich

Ausgabe: True, falls *x* dem Baum *t* hinzugefügt wurde,
False sonst

Nebeneffekt: *t* ist binärer Suchbaum mit einem neuen Blatt, dessen Wert *x* ist, falls *x* noch nicht enthalten war

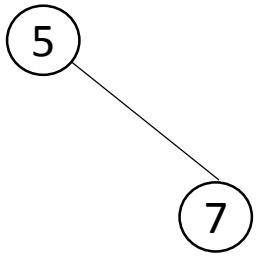
Methode: Suchen; falls nicht gefunden, am zuletzt aufgefundenen Knoten auf der korrekten Seite anfügen

***O*(depth(*t*))**

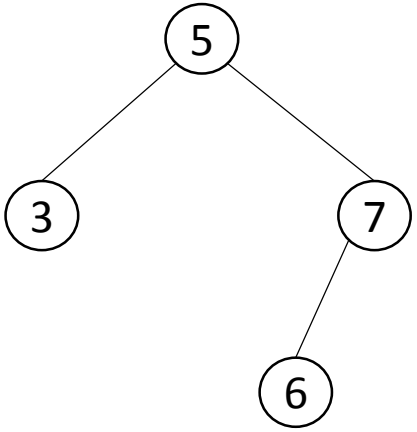
Funktion Einfügen – Beispiel

`add(5,null)`

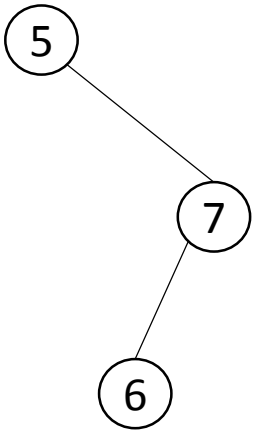
`add(7,t)`



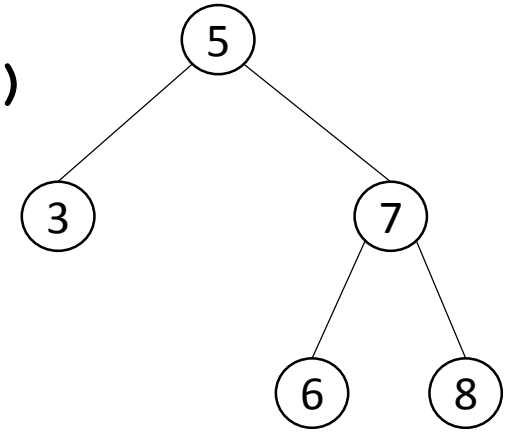
`add(3,t)`



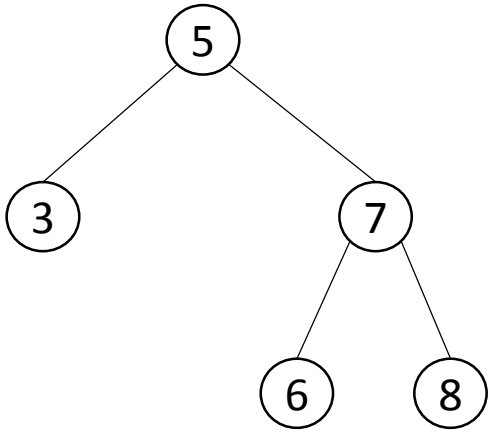
`add(6,t)`



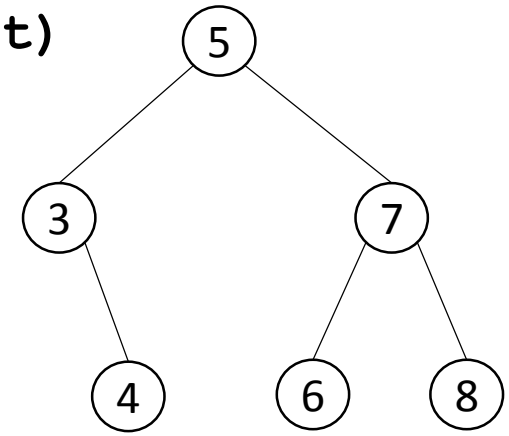
`add(8,t)`



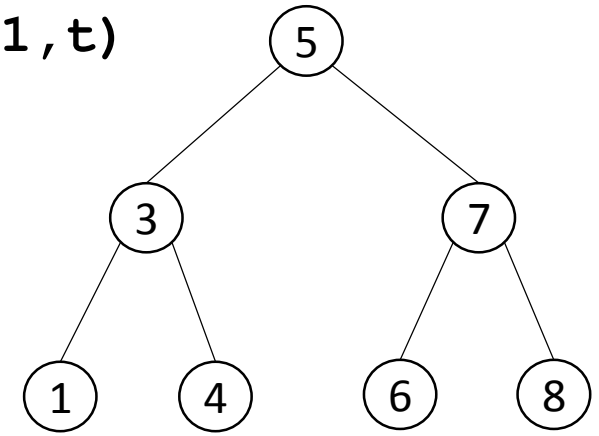
Funktion Einfügen – Beispiel (*Forts.*)



`add(4,t)`



`add(1,t)`



Funktion Entfernen

Name: **remove(*x*,*t*)**

Eingabe: binärer Suchbaum *t* mit Wurzel *r*,
Wert *x* aus dem Grundbereich

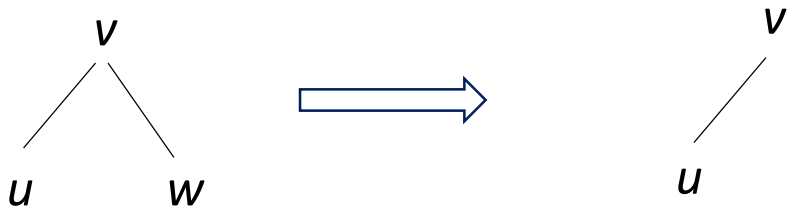
Ausgabe: True, falls *x* aus dem Baum *t* entfernt wurde,
False sonst

Nebeneffekt: *t* ist binärer Suchbaum, aus dem der Knoten mit Wert *x* entfernt ist, falls dieser existierte

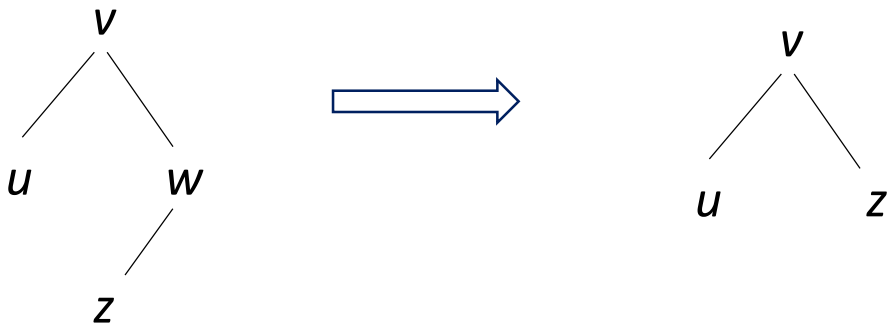
Methode: Suchen, falls gefunden Knoten entfernen
nach folgender Fallunterscheidung:

Funktion Entfernen (falls enthalten)

- 1. Fall: *x* ist Wert eines Blattes *w*
  → Streichen dieses Blattes *w*

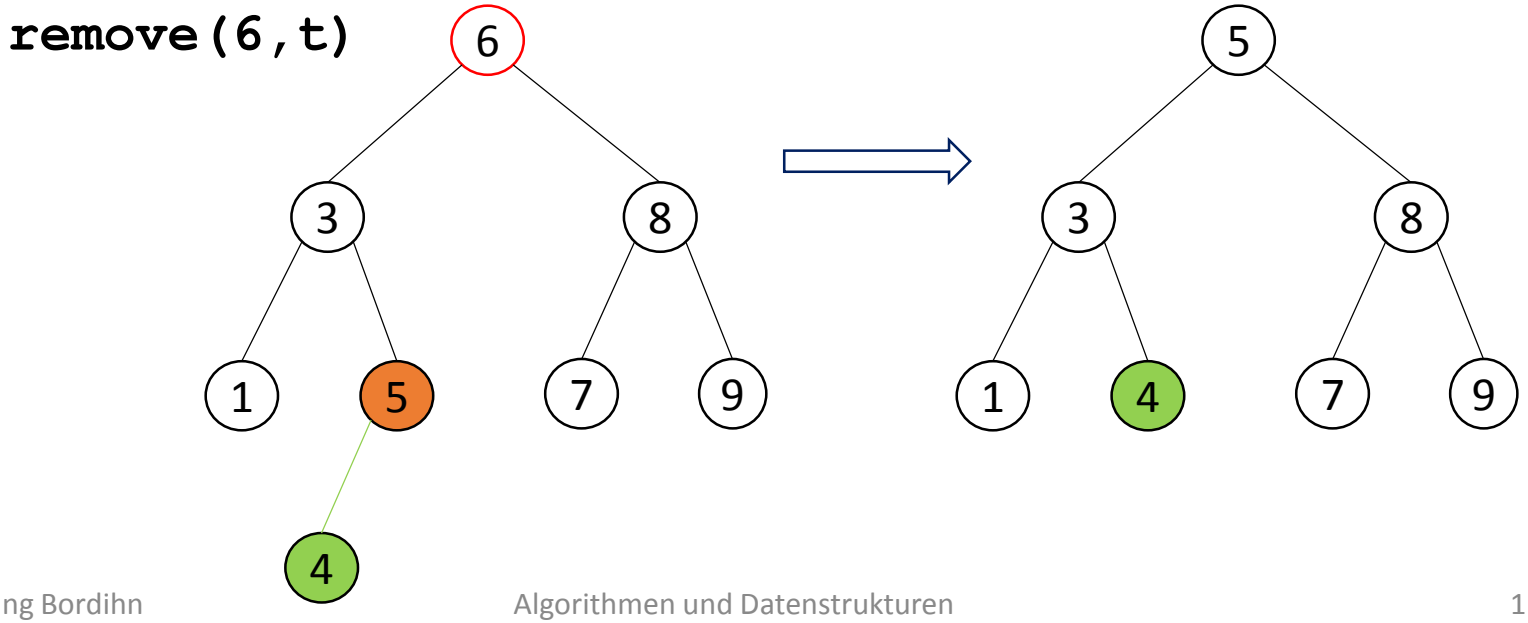


- 2. Fall: Knoten *w* mit Wert *x* hat genau ein Kind *z*
  → Ersetzen von *w* mit *z*



Funktion Entfernen (falls enthalten)

- 3. Fall: Knoten *w* mit Wert *x* hat zwei Kinder
  → Tauschen des Wertes in *w* mit dem Wert im am weitesten rechts stehenden Knoten *z* des linken Teilbaums von *w* und Entfernen von *z*



Funktion Entfernen

- Voraussetzung: Vater ist bekannt
- *entweder* Kopieren der Werte zwischen Knoten *oder* Umsetzen von Zeigern
- **Laufzeit:**
  - Alle Operationen auf einem Weg von der Wurzel zu einem Blatt:
    Suchen des zu löschenden Knotens, seines Vaters und des Knotens am weitesten rechts;
    Austausch von Knoten auf diesem Weg
  - somit $\boldsymbol{O(\text{depth}(t))}$

Balancierte binäre Suchbäume

Entarten von Suchbäumen

- Basisoperationen **`contains`**, **`add`**, **`remove`** mit Laufzeit $O(\text{depth}(t))$

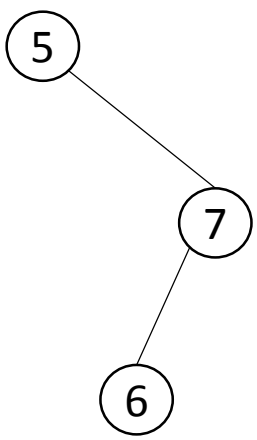
➢ Laufzeit $O(\log n)$ bei Suchbaum mit n Knoten bei einem **balancierten Suchbaum**

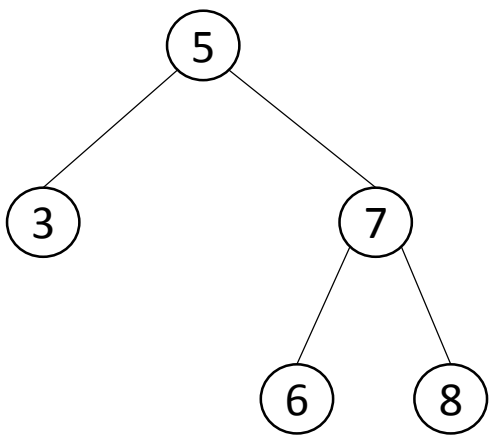
- Tiefe des Baums hängt aber von der Reihenfolge ab, in der Elemente eingefügt/entfernt werden

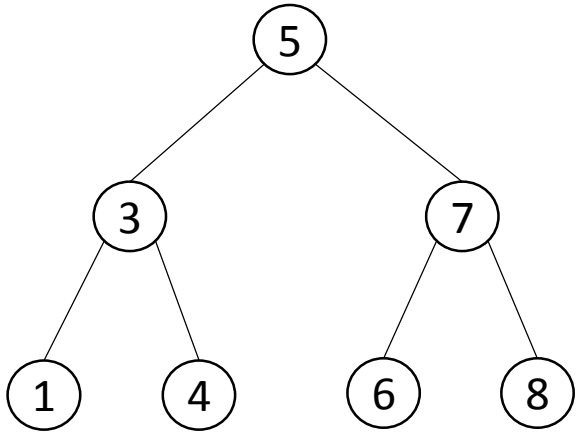
➢ Bäume können zu linearer Liste entarten

Entarten von Suchbäumen – Beispiel

1. günstige Reihenfolge des Einfügens: 5, 7, 6, 3, 8, 4, 1

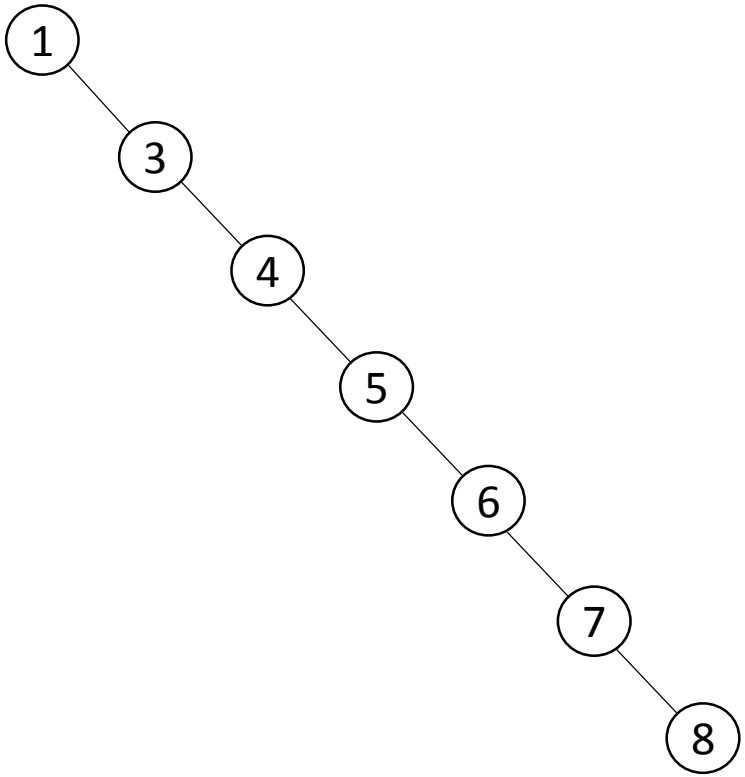






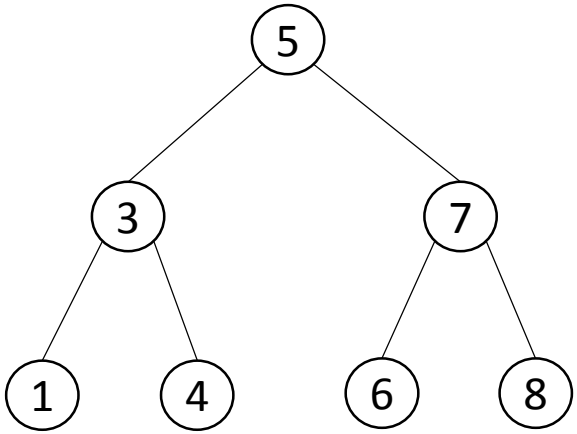
Entarten von Suchbäumen – Beispiel

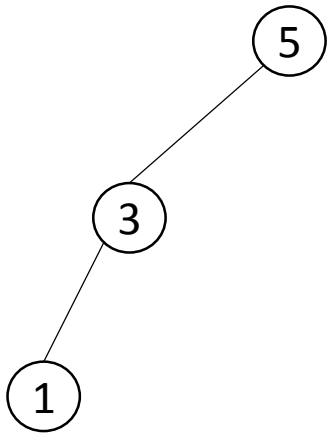
2. ungünstige Reihenfolge des Einfügens: 1, 3, 4, 5, 6, 7, 8



`contains`,
`add` und **`remove`**
mit ***O*(*n*)** Laufzeit
im schlechtesten Fall

Entarten von Suchbäumen durch Entfernen von Knoten – Beispiel






```
remove(6,t);
remove(8,t);
remove(7,t);
remove(4,t)
```

Balancieren von Suchbäumen

- Ist ein Suchbaum „zu stark entartet“
  - z.B. wenn es einen Knoten *v* gibt, so dass
    $|\text{depth}(\text{left}(v)) - \text{depth}(\text{right}(v))| > D$ für ein gewisses *D*,

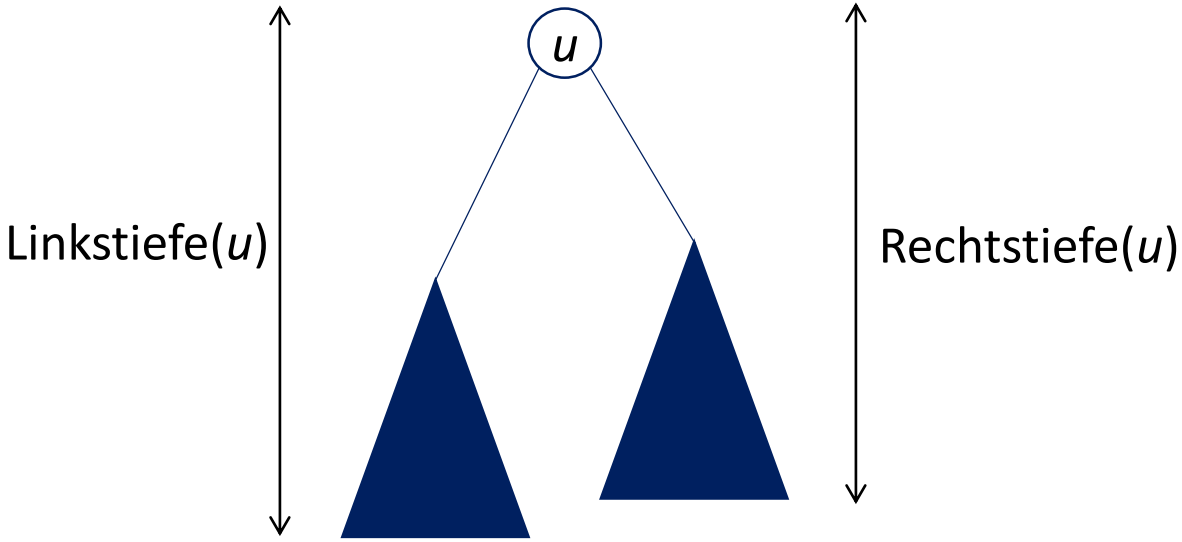
  dann wird der Suchbaum ausgeglichen
- *Naiver Algorithmus:*
  - Bestimmung des Medians der gespeicherten Werte
  - neuer Suchbaum mit Median an der Wurzel
  - ➢ lineare Zeit zum Balancieren ☹

AVL-Bäume

- binäre Suchbäume mit Operation „Rotieren“ zum effizienten Balancieren
- von Adelson-Velskii und Landis (1962)
- Aufruf der Funktion zum Balancieren nach jedem Einfügen oder Löschen, wenn die Eigenschaft *balanciert* dadurch zerstört wurde
- Jeder Knoten *v* bekommt zusätzlich einen `int`-Wert, der den Unterschied der Tiefen seiner Teilbäume misst

  (**Balancefaktor** von *v*)

Balancefaktor eines Knotens

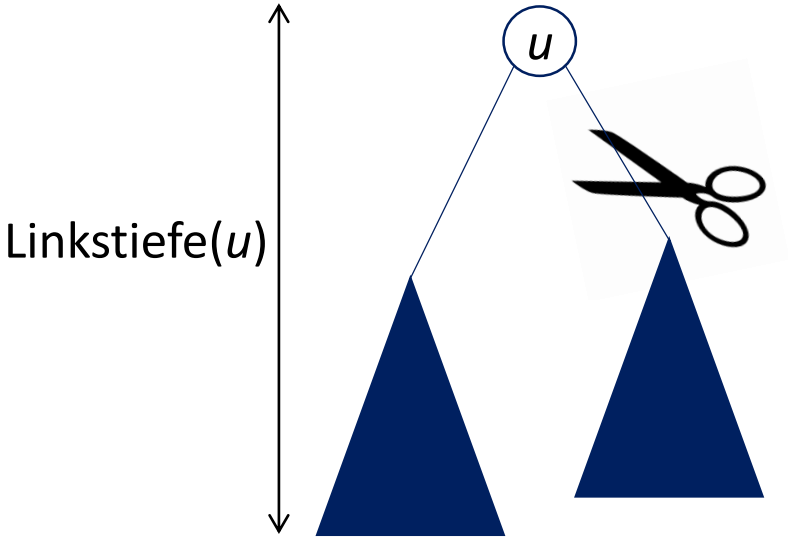


Balancefaktor von u:

b_u = Linkstiefe(u) – Rechtstiefe(u) = $b_{\text{address}(u)}$

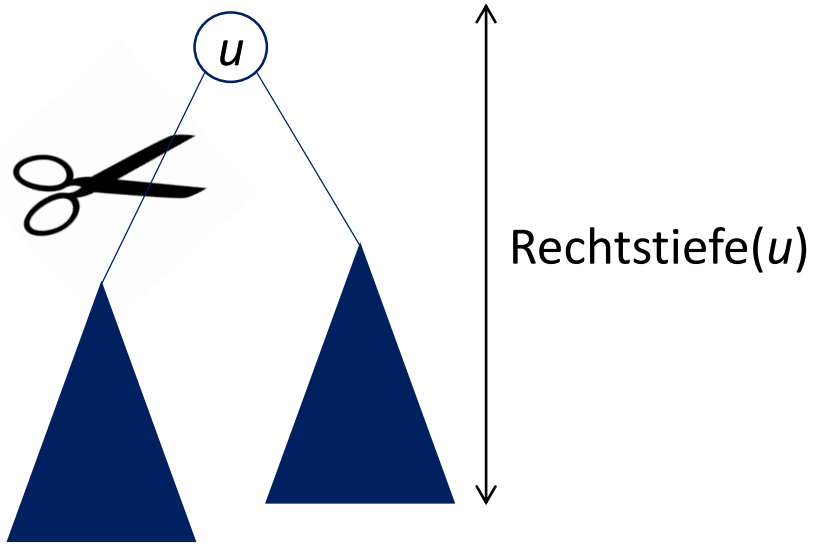
Linkstiefe eines Knotens

- Sei *t* ein gewurzelter Baum und *u* ein Knoten von *t*.
- **Linkstiefe(*u*)** ist die Tiefe des Teilbaums von *t* mit Wurzel *u* ohne right(*u*)



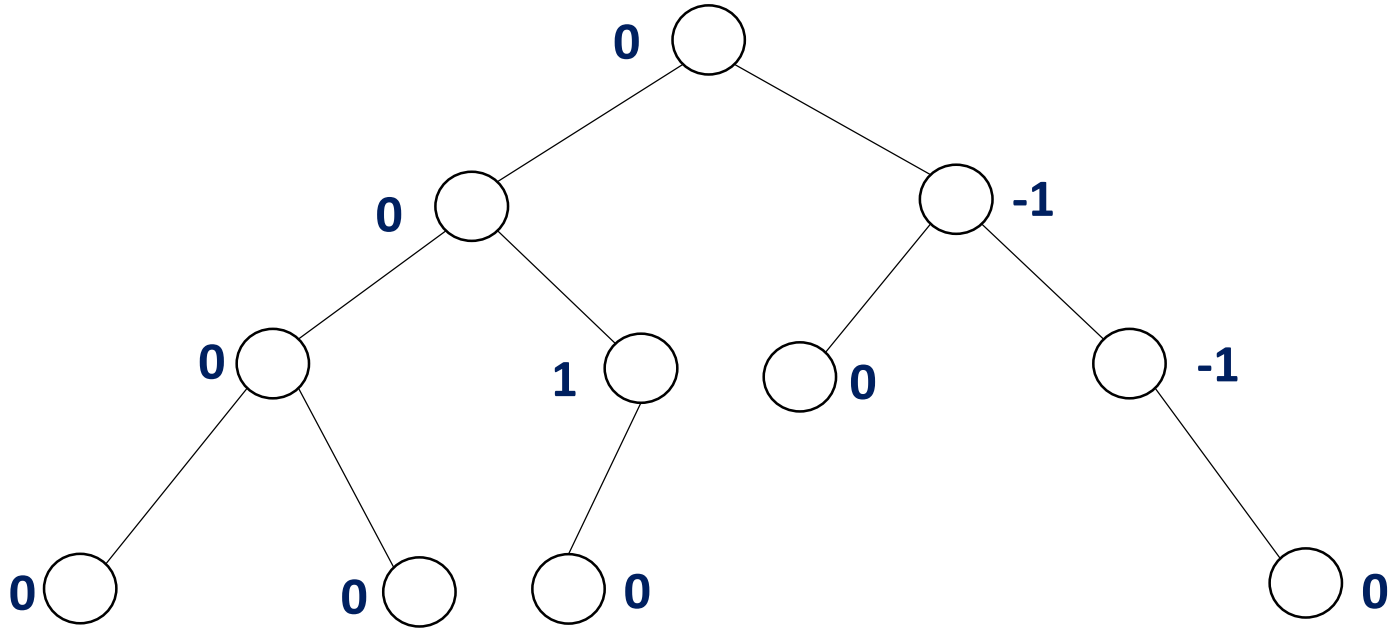
Rechtstiefe eines Knotens

- Sei *t* ein gewurzelter Baum und *u* ein Knoten von *t*.
- **Rechtstiefe(*u*)** ist die Tiefe des Teilbaums von *t* mit Wurzel *u* ohne left(*u*)



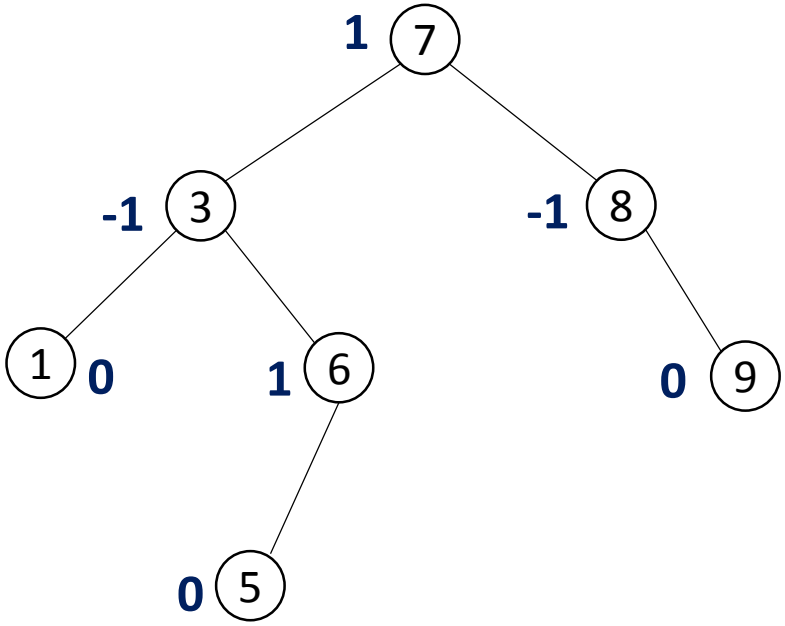
Balancefaktor – Beispiel

- balancierter Binärbaum mit Balancefaktoren:

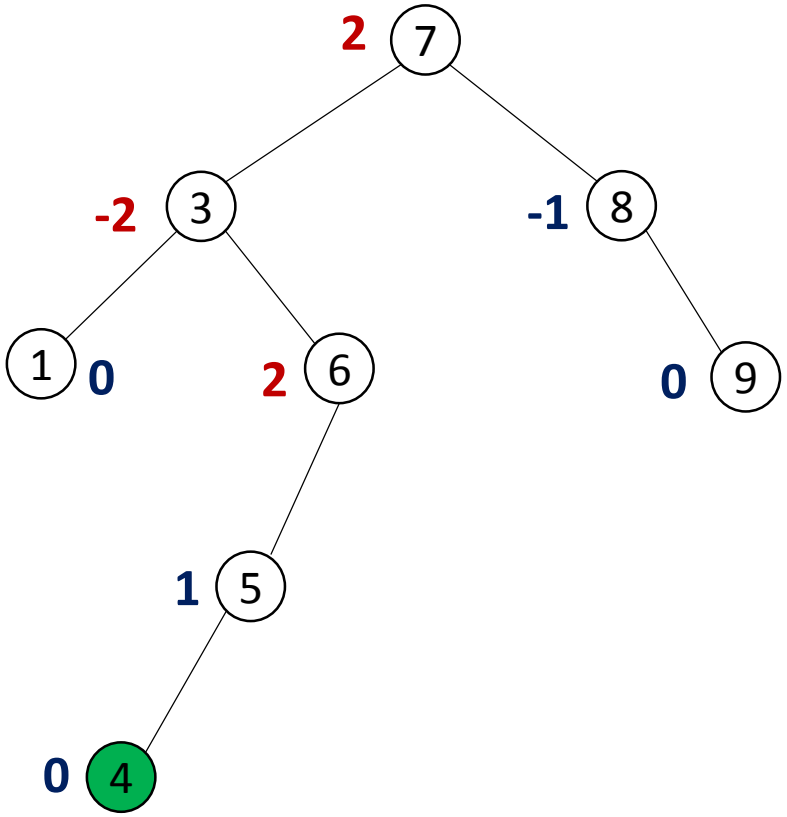


➢ Bei balancierten Bäumen für alle v: $b_v \in \{-1, 0, 1\}$

Einfügen in AVL-Bäume – Beispiel



4 einfügen!



Balancefaktoren nach Einfügen aktualisieren

Eingabe: Blatt *n* eines AVL-Baums *t*
Ausgabe: *t* mit neu berechneten Balancefaktoren von *n* an aufwärts bis zur Wurzel

update_factors_add(*n*):

$b_n \longleftarrow 0$

$v \longleftarrow$ Vater von *n*

solange $v \neq$ null # n ist nicht die Wurzel

führe aus

falls (left(*v*) = address(*n*))

$b_v \longleftarrow b_v + 1$

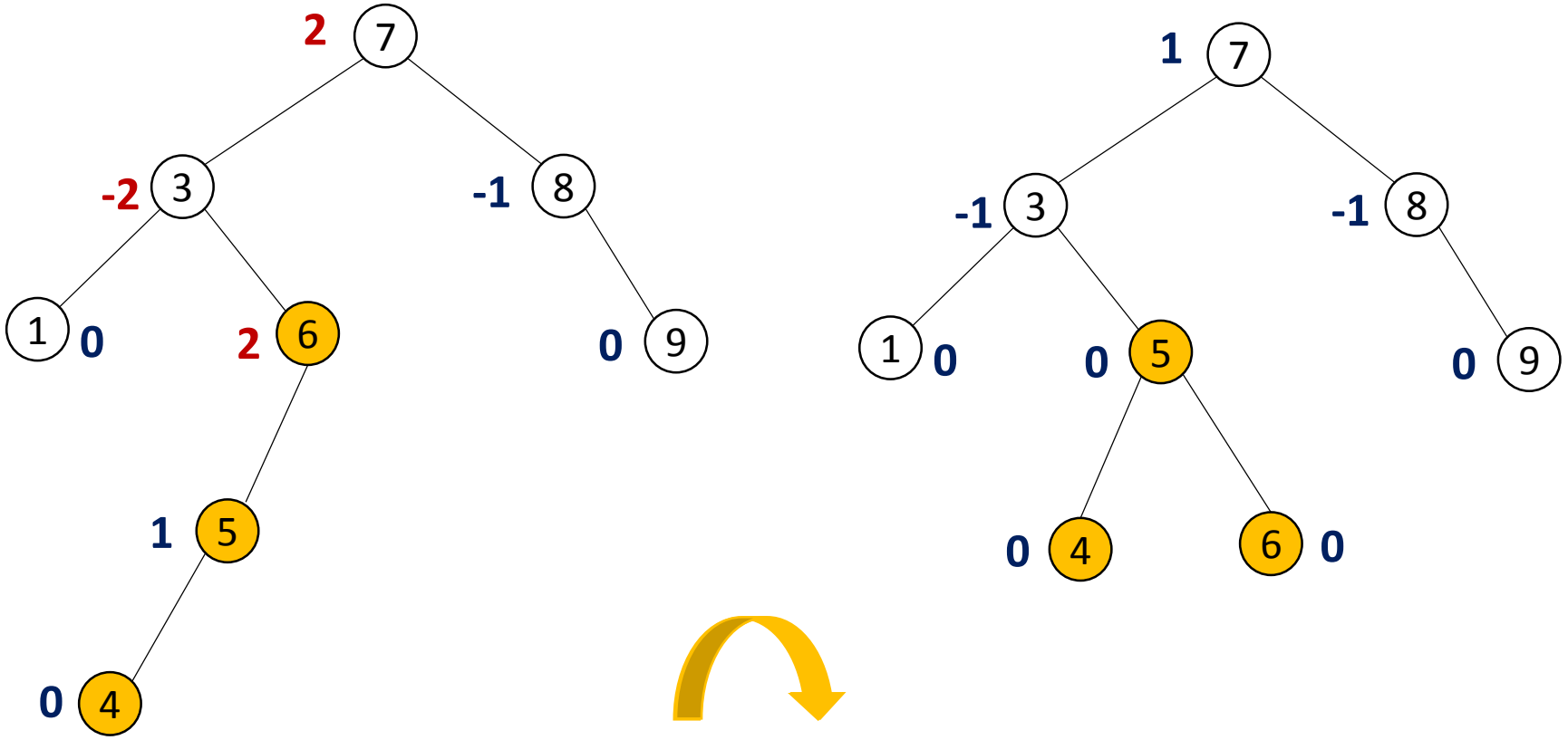
sonst

$b_v \longleftarrow b_v - 1$

$n \longleftarrow v$

$v \longleftarrow$ Vater von *n*

Rechtsrotation – Beispiel



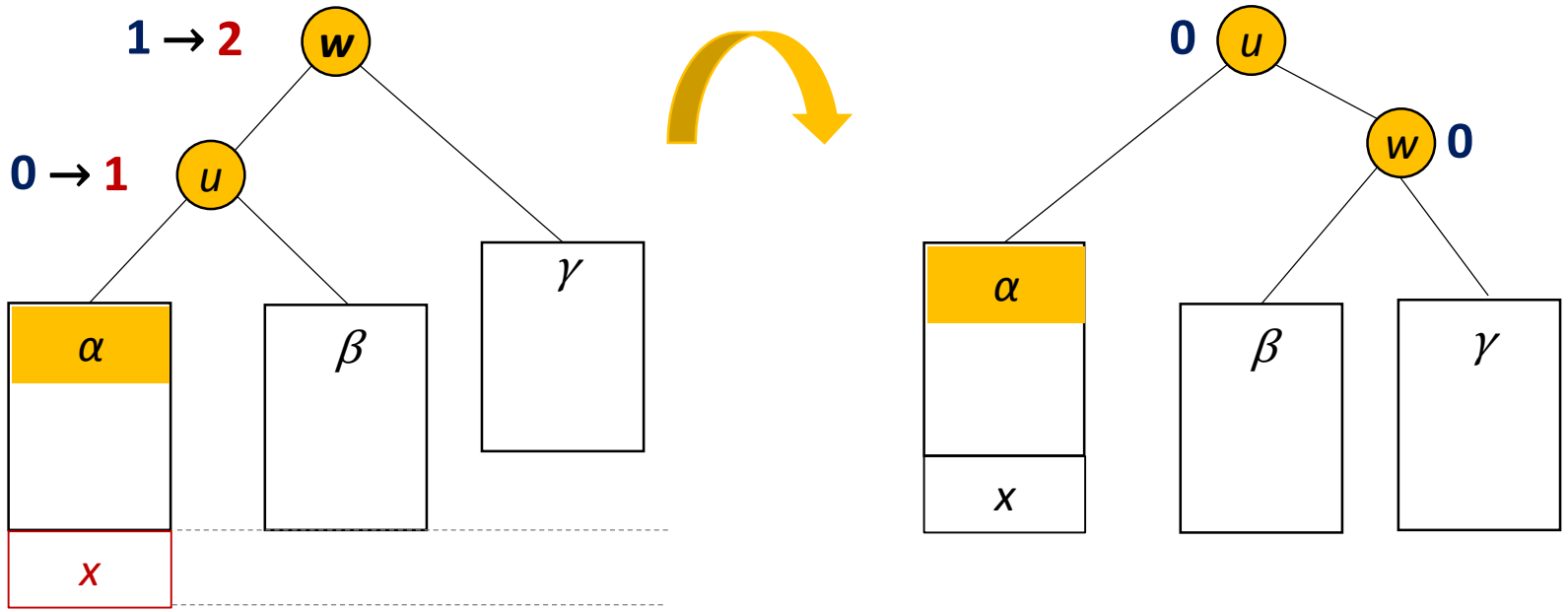
Einfügen in einen AVL-Baum (1)

- neues Element als Blatt einfügen
- Balancefaktoren können sich nur entlang des Suchpfades verändert haben
- Ist der Balancefaktor nach dem Einfügen überall aus {-1, 0, 1}, so ist wieder ein AVL-Baum entstanden.
- Sonst: suche Knoten ***w*** mit maximaler Tiefe, dessen Balancefaktor 2 oder -2 ist

  Nun gibt es vier Fälle ...

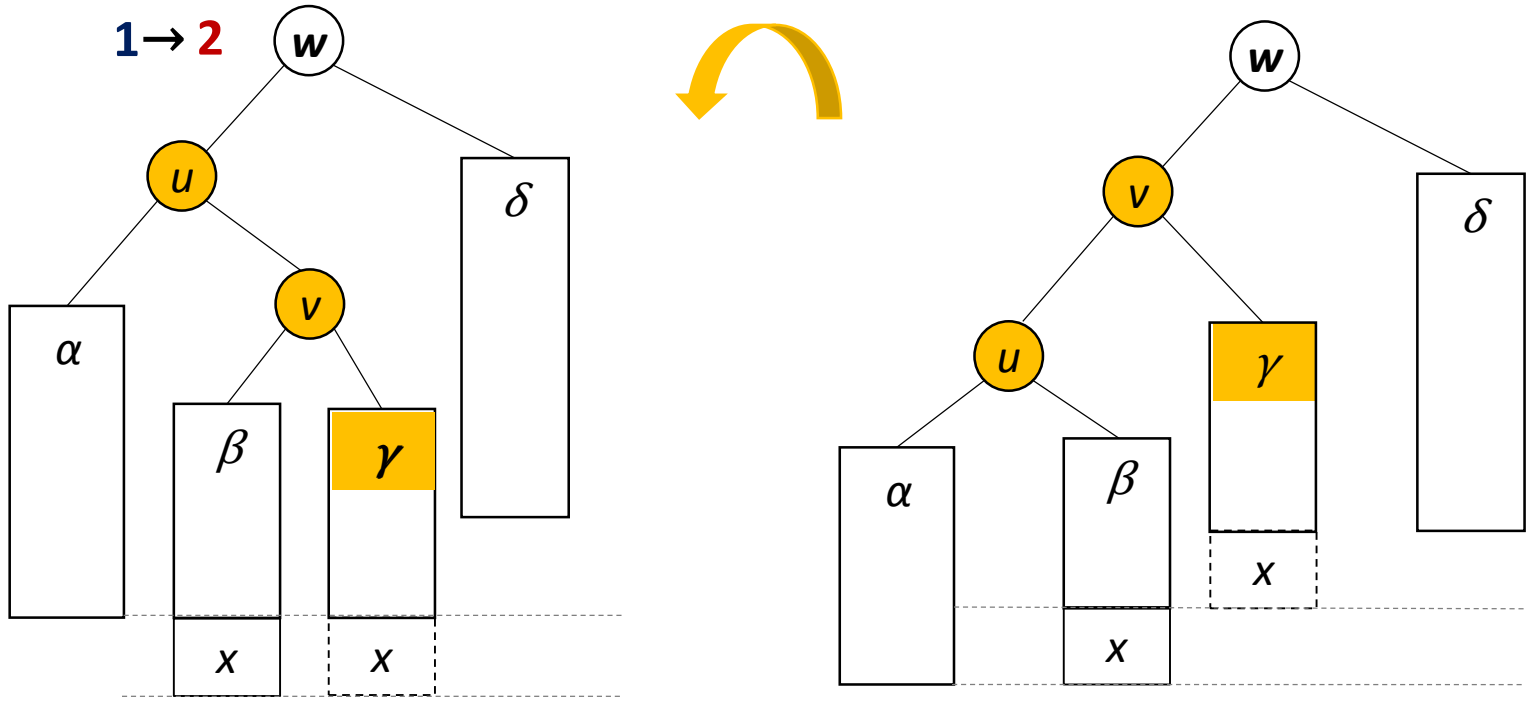
Einfügen in einen AVL-Baum (2)

1. „Überhang“ am linken Teilbaum links von ***w*** (Suchpfad von ***w*** beginnt mit *links-links*)
   → Rechtsrotation



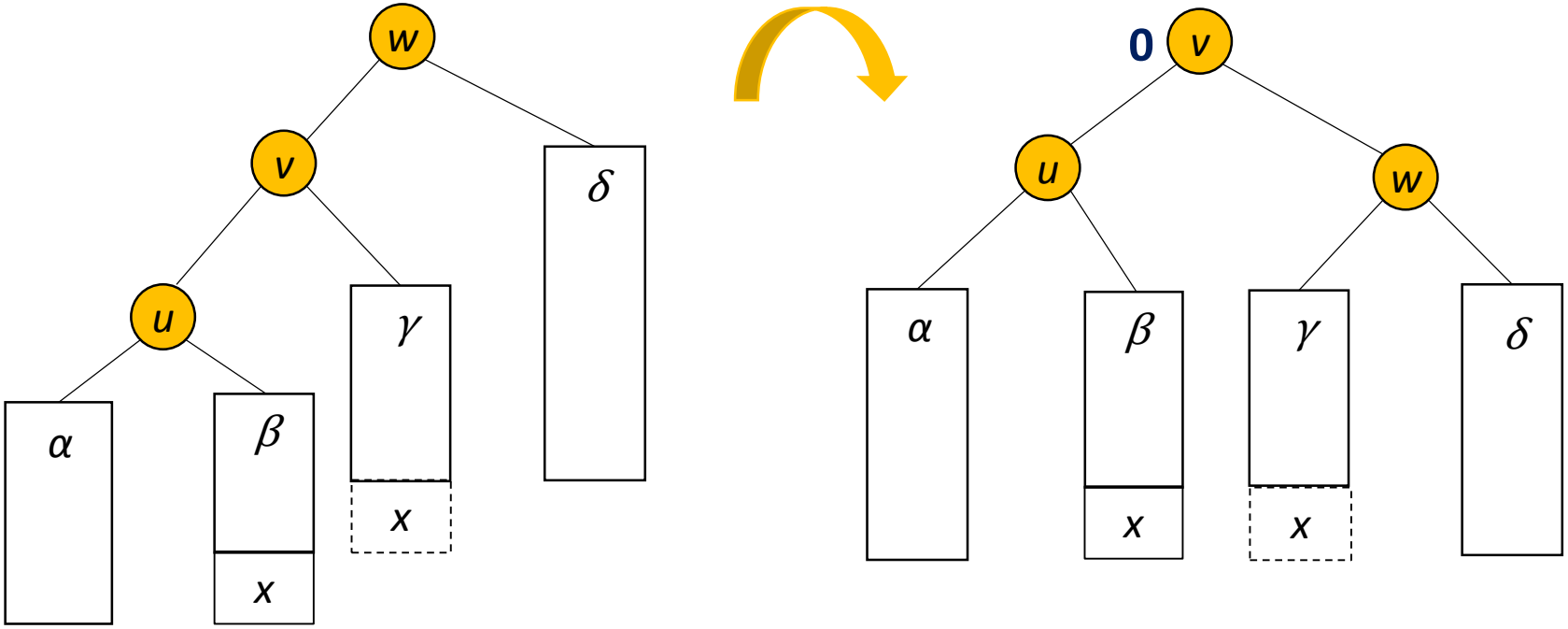
Einfügen in einen AVL-Baum (3.1)

2. „Überhang“ am rechten Teilbaum links von ***w***
   (Suchpfad von ***w*** beginnt mit *links-rechts*)
   → **Doppelrotation (*zuerst Linksrotation ...*)**



Einfügen in einen AVL-Baum (3.2)

2. „Überhang“ am rechten Teilbaum links von ***w***
   (Suchpfad von ***w*** beginnt mit *links-rechts*)
   → **Doppelrotation (... *dann Rechtsrotation*)**



Einfügen in einen AVL-Baum (4)

symmetrische Fälle:

3. „Überhang“ am linken Teilbaum rechts von ***w***
   (Suchpfad von ***w*** beginnt mit *rechts-links*)
   → **Doppelrotation**
   (*erst Rechts- dann Linksrotation*)

4. „Überhang“ am rechten Teilbaum rechts von ***w***
   (Suchpfad von ***w*** beginnt mit *rechts-rechts*)
   → **Linksrotation**

Pseudocode: Einfache Rotation

R_rotate(w):

$u \longleftarrow$ content(left(w))

$\beta \longleftarrow$ content(right(u))

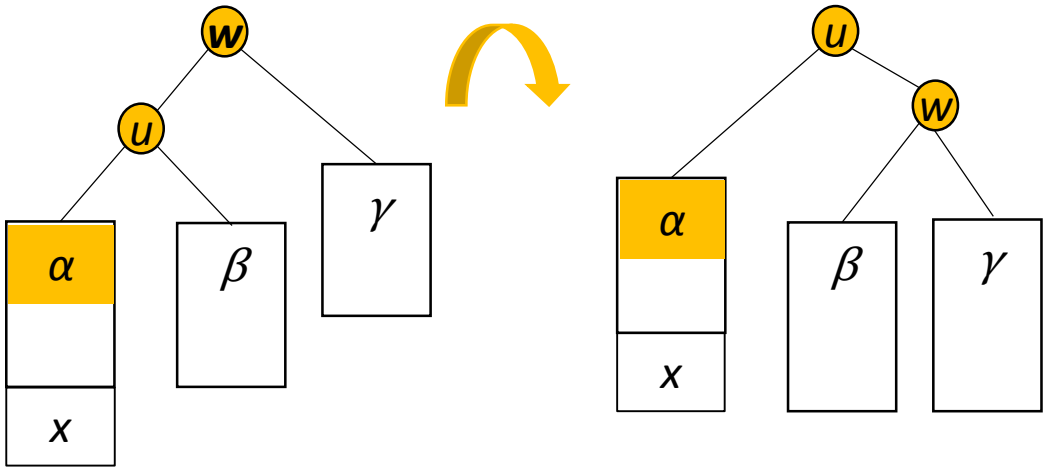
setLeft(w, address(β))

setRight(u, address(w))

setParent(u, getParent(w))

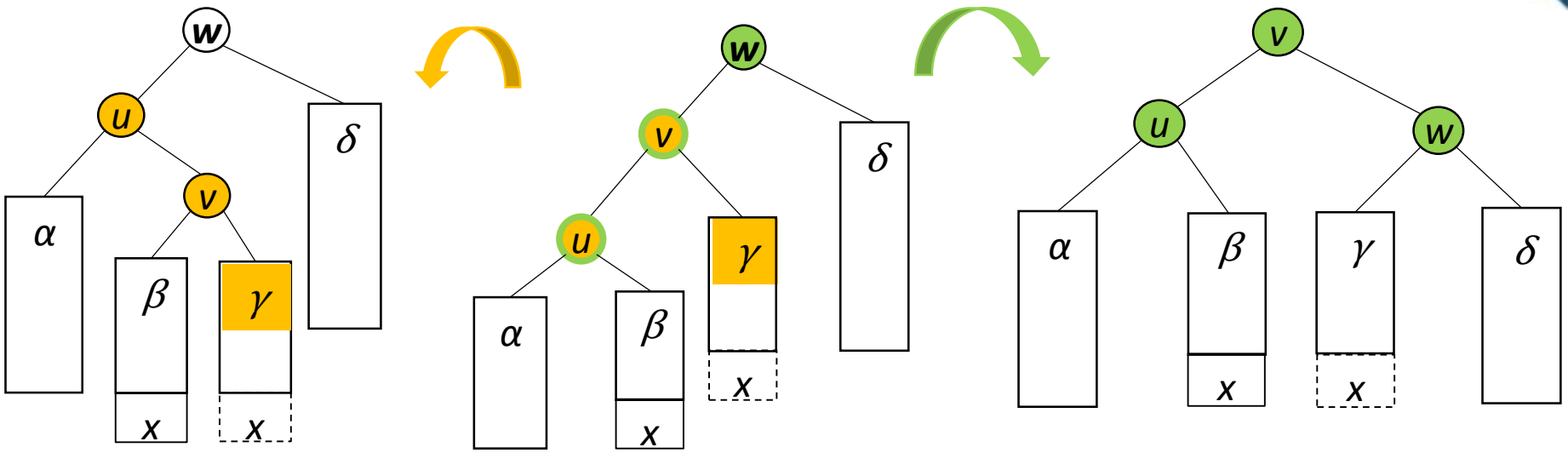
setParent(w, address(u))

setParent(β, address(w))



Analog L_rotate(w)

Doppelte Rotation



LR_rotate(*w*):

u ⟵ content(left(*w*))

L_rotate(*u*)

R_rotate(*w*)

(Alternativ kann die Konstruktion des Ergebnisbaums auch direkt implementiert werden.)

Analog RL_rotate(*w*)

Pseudocode Einfügen (1)

Eingabe: AVL-Baum *t* mit Wurzel *r*,
Element *x* aus Grundbereich

Ausgabe: True, falls *x* dem Baum hinzugefügt wurde, False sonst

Nebeneffekt: *t* ist binärer AVL-Baum mit einem neuen Blatt *n*,
dessen Wert *x* ist, falls *x* noch nicht enthalten war

res ⟵ add(*x*,*t*) # *Suchbaum mit neuem Blatt n* (mit Wert *x*)

falls *res* = False **gib** *res* **aus**, **STOP**

update_factors_add(*n*)

w ⟵ Vater von *n*

solange $-1 \leq b_w \leq 1$ UND $w \neq r$ # tiefsten *Knoten w mit* $|b_w| = 2$ *suchen*

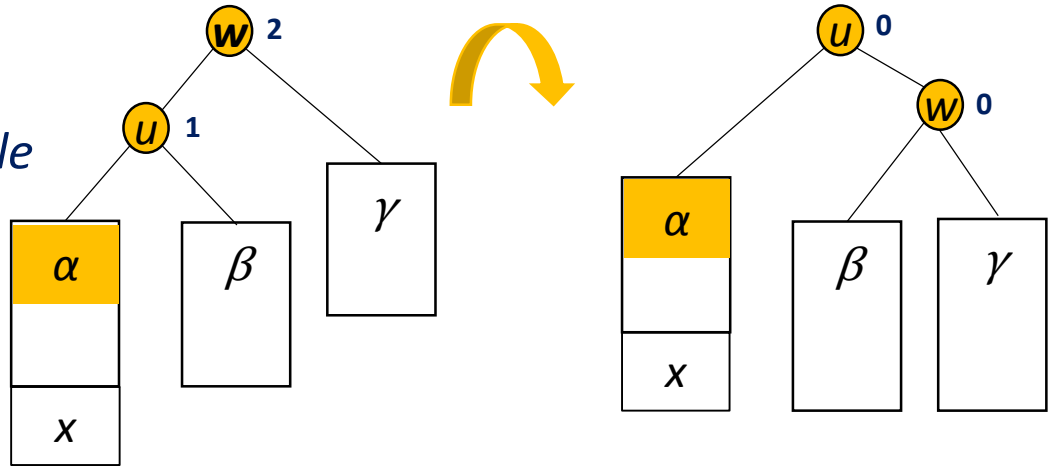
w ⟵ Vater von *w*

falls $-1 \leq b_w \leq 1$ **gib** *res* **aus** # *sonst: siehe nächste Folie*

Pseudocode Einfügen (2)

sonst
  falls $b_w = 2$ UND $b_{\text{left(content}(w))} = 1$ # *w* ist jetzt Knoten mit $b_w = 2$ oder $b_w = -2$
    $u \longleftarrow$ content(left(*w*)) # *Fall, der die Rechtsrotation erfordert*
    R_rotate(*w*)
    $b_w \longleftarrow 0$
    $b_u \longleftarrow 0$
    Balancefaktoren entlang des Suchpfads neu berechnen
    gib *res* **aus**
    STOP

 # Analog die drei anderen Fälle



Löschen aus AVL-Bäumen

- `remove` aufrufen
- Jeden Knoten entlang des Suchpfads, beginnend am Vater des gelöschten Knotens bis zur Wurzel, auf Ausgeglichenheit testen und ggf. durch Rotieren ausgleichen (rebalancieren).
- *Annahme*: Wir sind dabei an einem Knoten angelangt, dessen linker Teilbaum „verkürzt“ wurde. (Betrachtungen für rechten Teilbaum symmetrisch)

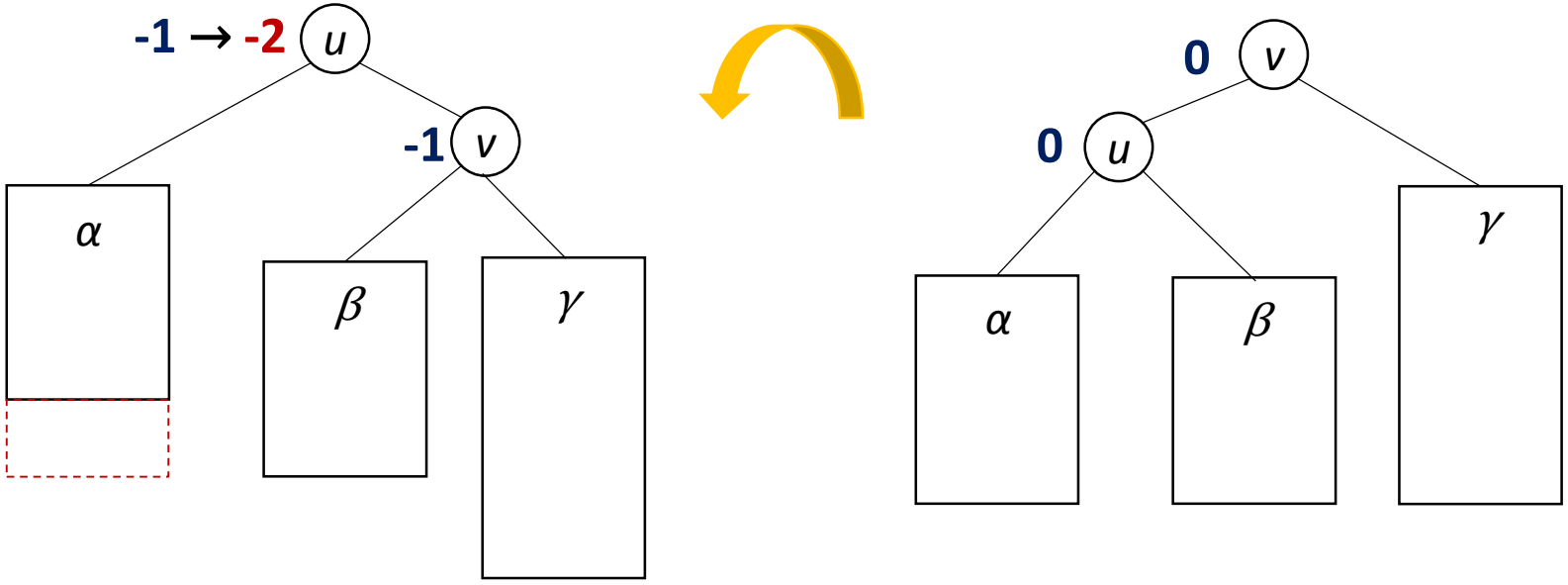
Fallunterscheidung Löschen

- Sei *u* der zu gerade behandelte Knoten, dessen linker Teilbaum „verkürzt“ wurde. *Es gibt drei Fälle:*
  1. Vorher war $b_u = 1$. Dann setze b_u auf 0.
  2. Vorher war $b_u = 0$. Dann setze b_u auf -1.
  3. Vorher war $b_u = -1$. Dann unterscheide drei Unterfälle.

     Sei *v* rechtes Kind von *u*:
     1. $b_v = -1$
     2. $b_v = 0$
     3. $b_v = 1$

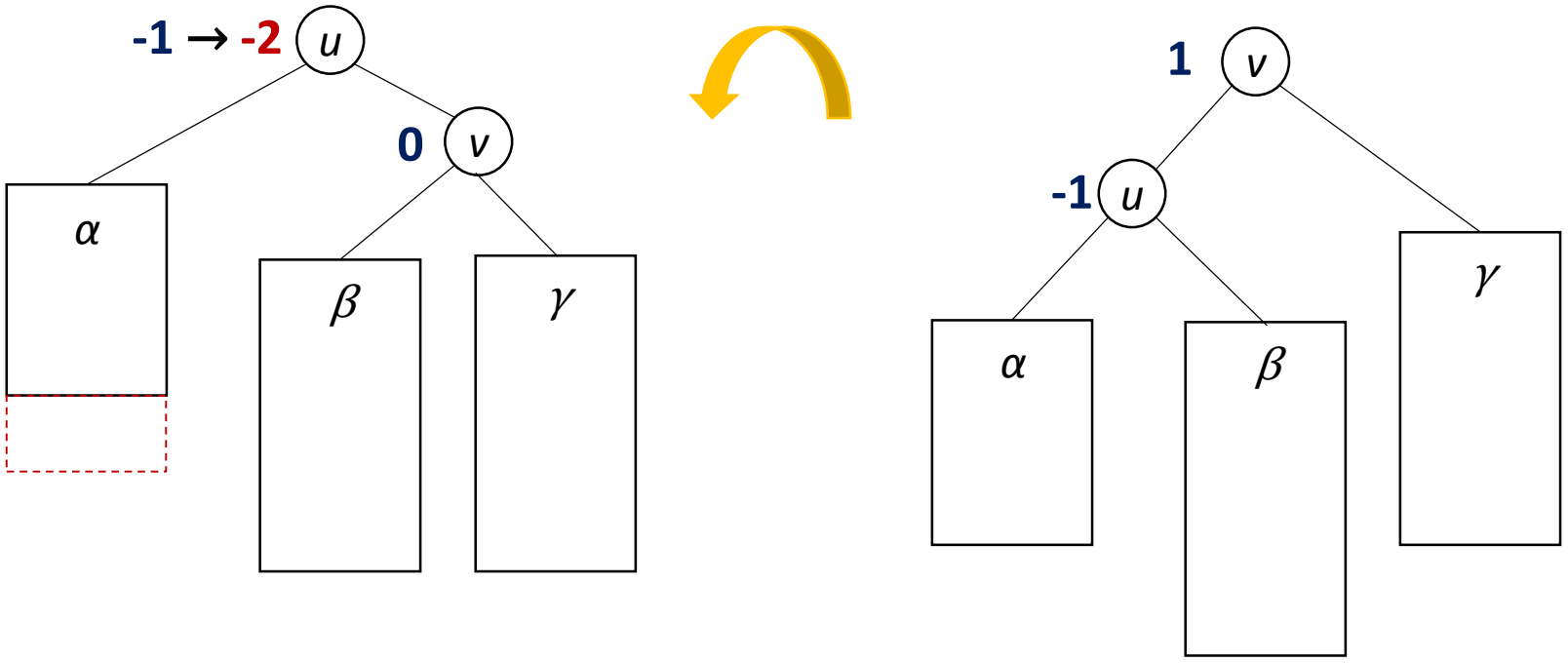
Löschen – Fall 3.1

- L-Rotation von *u*



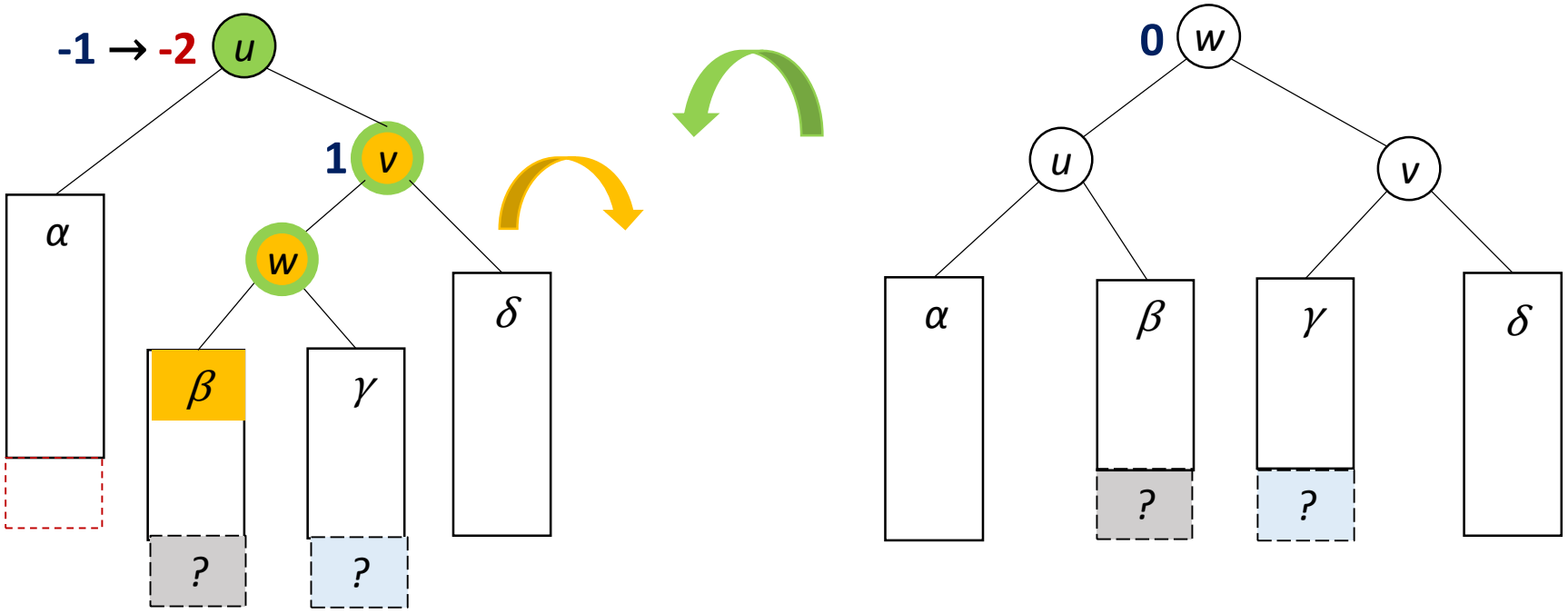
Löschen – Fall 3.2

- L-Rotation von *u*



Löschen – Fall 3.3

- RL-Rotation



Laufzeitanalyse

- Für einen AVL-Baum ***t*** mit ***n*** Elementen (*worst case*):
- **`add`** bzw. **`remove`**: ***O*(depth(*t*)) = *O*(log *n*)**
- Aufsuchen der Stellen, an denen rotiert werden muss, nur entlang des Suchpfads (zurück zur Wurzel): ***O*(depth(*t*)) = *O*(log *n*)**
- Rotationsoperationen: ***O*(1)**
- Aktualisierung der Balancefaktoren nur entlang des Suchpfads: ***O*(log *n*)** (nach dem Einfügen bzw. Löschen und nach jeder Rotation)

➢ ***O*(log *n*)** für Suchen, Einfügen und Löschen in AVL-Bäumen