

Kap. 4.2 Binäre Suchbäume ff Kap. 4.3: AVL-Bäume



Professor Dr. Petra Mutzel

Lehrstuhl für Algorithm Engineering, LS11

Fakultät für Informatik, TU Dortmund

12./13. VO DAP2 SS 2009 28.5./2.6.2009

Motivation

„Warum soll ich heute hier bleiben?“

Balancierte Bäume brauchen Sie immer wieder!

„Was gibt es heute Besonderes?“

Schöne Animationen: Rotationen

Überblick

- Kurz-Wiederholung binäre Suchbäume + Pseudocode

- Einführung von AVL-Bäumen

- Implementierung der Operationen

- Schönes Java-Applet

Implementierung von $SEARCH(r,s)$ in binären Suchbäumen

Wiederholung

1. Vergleiche s mit dem Schlüssel $r.key$ an der Wurzel r (des Teilbaums)
2. Falls gefunden: STOP!
3. Sonst: Falls $s < r.key$: suche im linken Teilbaum
4. Sonst: suche im rechten Teilbaum
5. Gehe zu 1.
6. Ausgabe: nicht gefunden!

Pseudocode von SEARCH

Eingabe: Baum mit Wurzel p ; Schlüssel s

Ausgabe: Knoten mit Schlüssel s oder nil , falls s nicht da

Function SEARCH(p,s):TreeNode

(1) **while** $p \neq nil \wedge p.key \neq s$ **do** {

(2) **if** $s < p.key$ **then**

(3) $p := p.left$

(4) **else** $p := p.right$

(5) }

(6) **return** p

Implementierung von INSERT(r, q) in binären Suchbäumen

- Einfügen eines Knotens q mit Schlüssel s und Wert v in den Baum mit Wurzel r , falls noch nicht vorhanden; sonst Wert überschreiben mit v

1. Suche nach s
2. **Falls** die Suche erfolgreich endet, dann gilt $p.key == s$; **Dann:** $p.info := v$
3. **Sonst:** endet mit der Position eines leeren Unterbaums: Einfügen von q an diese Position

Pseudocode von INSERT

Eingabe: Baum mit Wurzel $root$; Schlüssel s und Wert v

Prozedur INSERT($root, s, v$):TreeNode

(1) **var** TreeNode r, p

(2) $r := nil$; $p := root$

(3) **while** $p \neq nil$ **do** {

(4) $r := p$ // r ist der zuletzt besucht Knoten

(5) **if** $s < p.key$ **then**

(6) $p := p.left$

(7) **else if** $s > p.key$ **then** $p := p.right$

(8) **else** { // $s == p.key$

(9) $p.info := v$

(10) **return**

(11) } } //end while

Pseudocode von INSERT ff

```
// Suche endet an leerem Unterbaum
```

```
// r ist letzter nicht-leerer Knoten
```

```
(12) q := new TreeNode
```

```
(13) q.parent := r
```

```
(14) q.key := s
```

```
(15) q.info := v
```

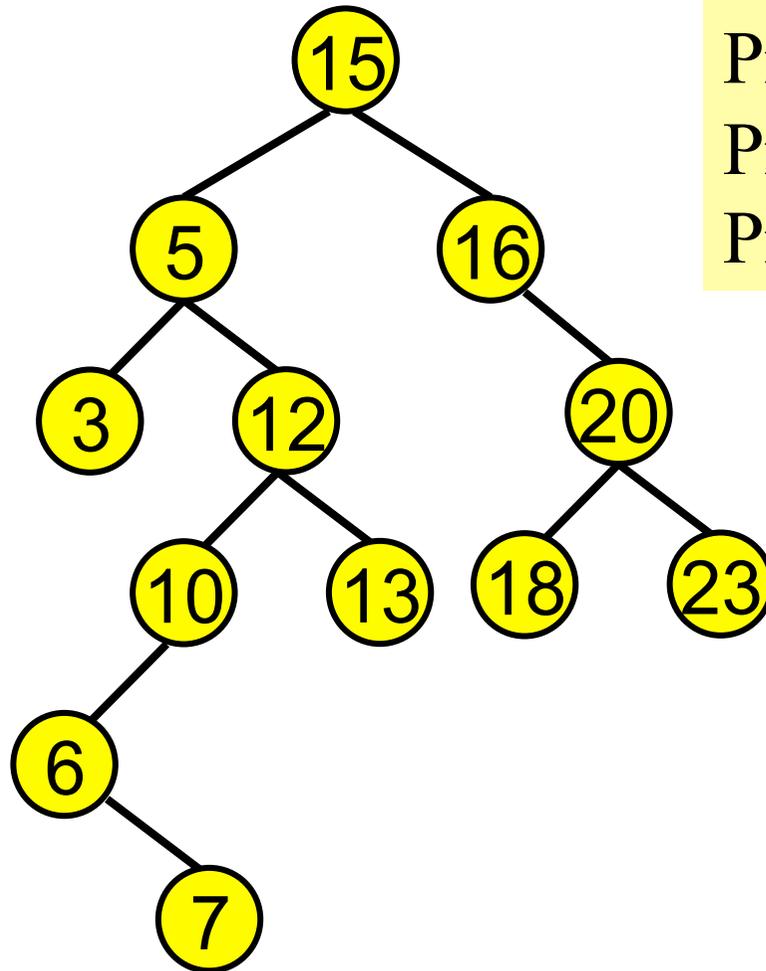
```
(16)if r == nil then
```

```
(17)   root := q // neuen Knoten in leeren Baum einfügen
```

```
(18) else if q.key < r.key then r.left := q
```

```
(19)   else                               r.right := q
```

Predecessor-Suche



Predecessor(15) = 13

Predecessor(13) = 12

Predecessor(6) = 5

Implementierung von `PREDECESSOR(r,p)`

1. Falls `p` linkes Kind hat: Return `Maximum(p.left)`
2. Sonst: Falls `p` rechtes Kind ist: Return(`p.parent`)
3. Sonst: wandere solange nach oben bis der aktuelle Knoten zum ersten Mal **rechtes Kind ist**; dann: Return(`p.parent`)
4. oder die Wurzel erreicht ist; dann: existiert kein Vorgänger (größter Knoten mit $key \leq p.key$).

Pseudocode von PREDECESSOR

Eingabe: Knoten $p \neq \text{nil}$

Ausgabe: Vorgänger von Knoten p in Inorder-Traversierung

Function PREDECESSOR(p):TreeNode

(1) **var** TreeNode q // q ist parent von p

(2) **if** $p.\text{left} \neq \text{nil}$ **then**

(3) **return** MAXIMUM($p.\text{left}$)

(4) **else** {

(5) $q := p.\text{parent}$

(6) **while** $q \neq \text{nil}$ and $p == q.\text{left}$ **do** {

(7) $p := q$

(8) $q := q.\text{parent}$

(9) }

(10) **return** q }

Implementierung von **MAXIMUM**

1. Wir durchlaufen von Wurzel aus rekursiv den rechten Unterbaum, bis wir auf ein leeres Kind treffen.
2. Der letzte durchlaufene Knoten enthält dann den größten Schlüssel.

Pseudocode:

Eingabe: nichtleerer Baum mit Wurzel $p \neq \text{nil}$

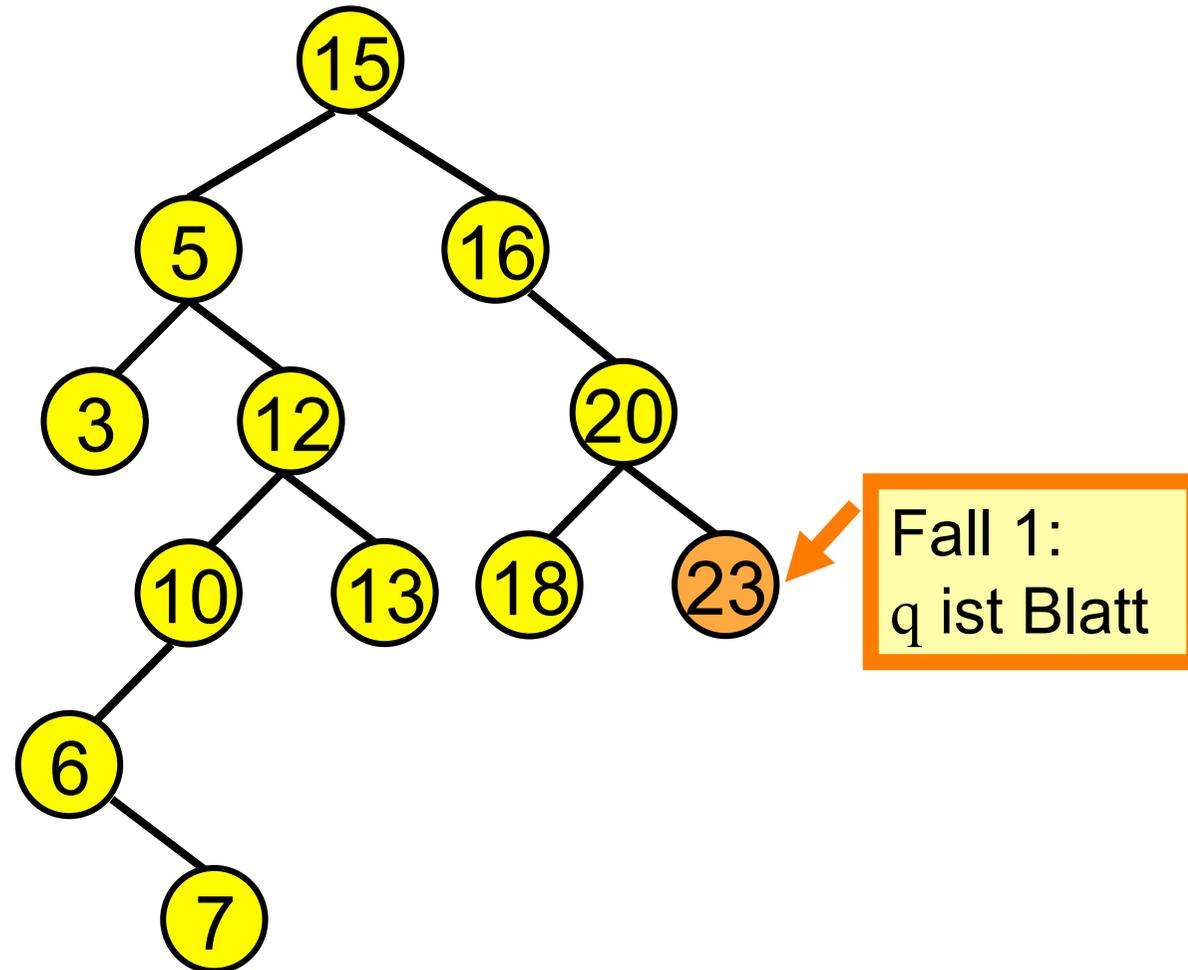
Ausgabe: Knoten im Baum mit kleinstem Schlüssel

Function **MINIMUM**(p):TreeNode

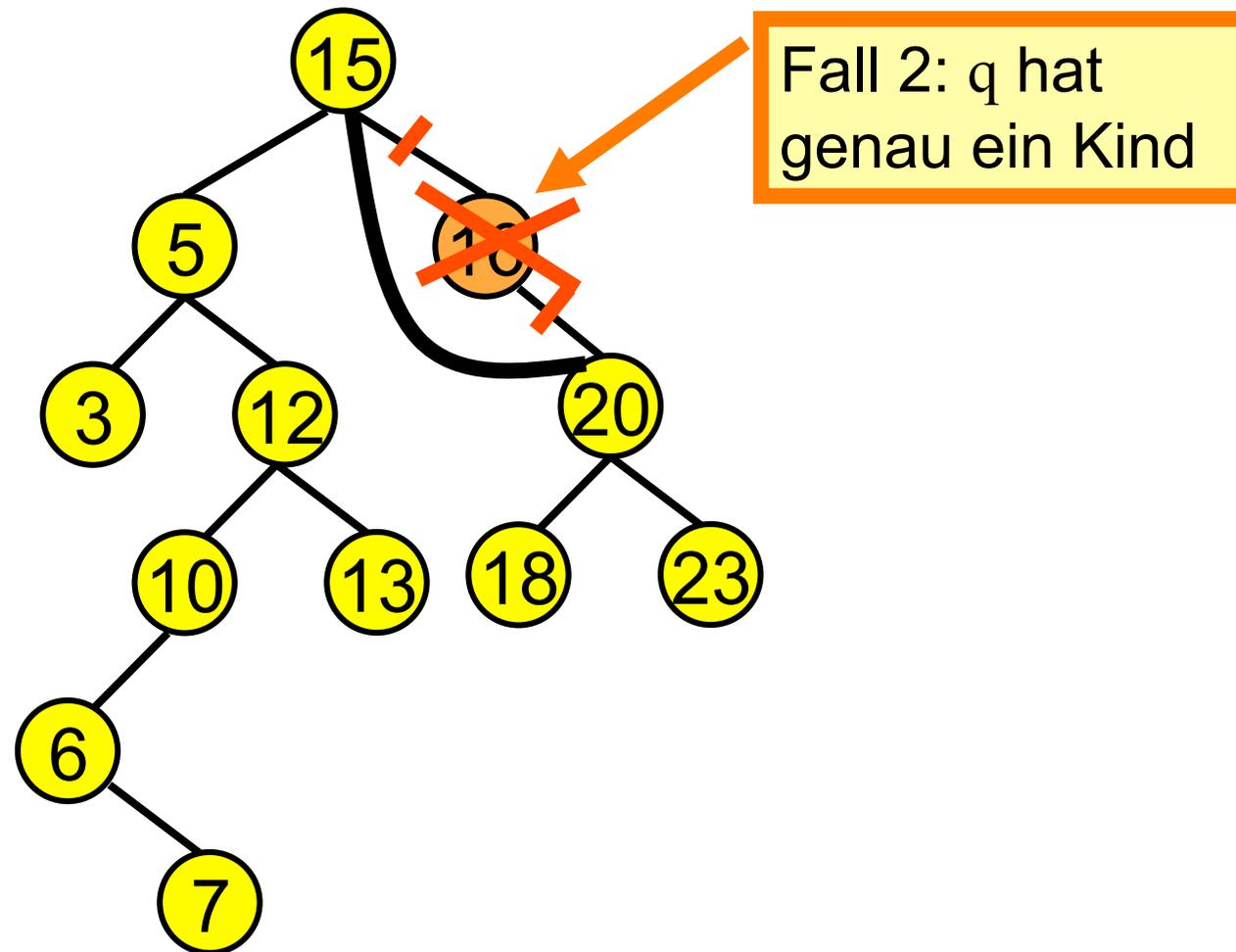
(1) **while** p.right \neq nil **do** p:=p.right

(2) **return** p

Beispiel für DELETE(r,q)

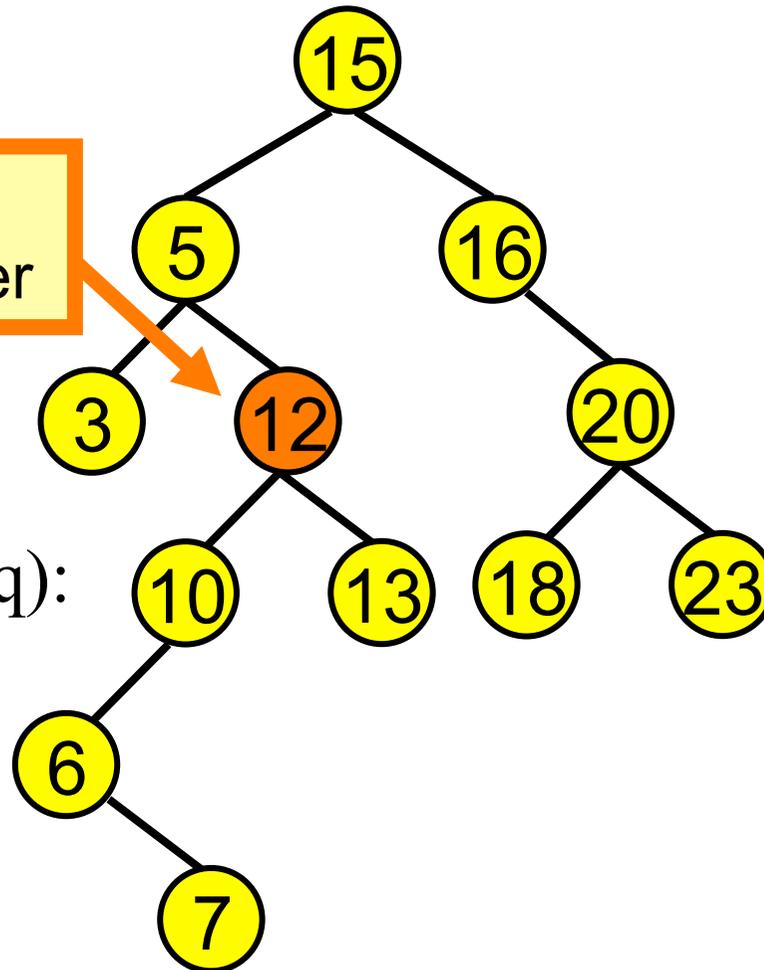


Beispiel für DELETE(r,q)



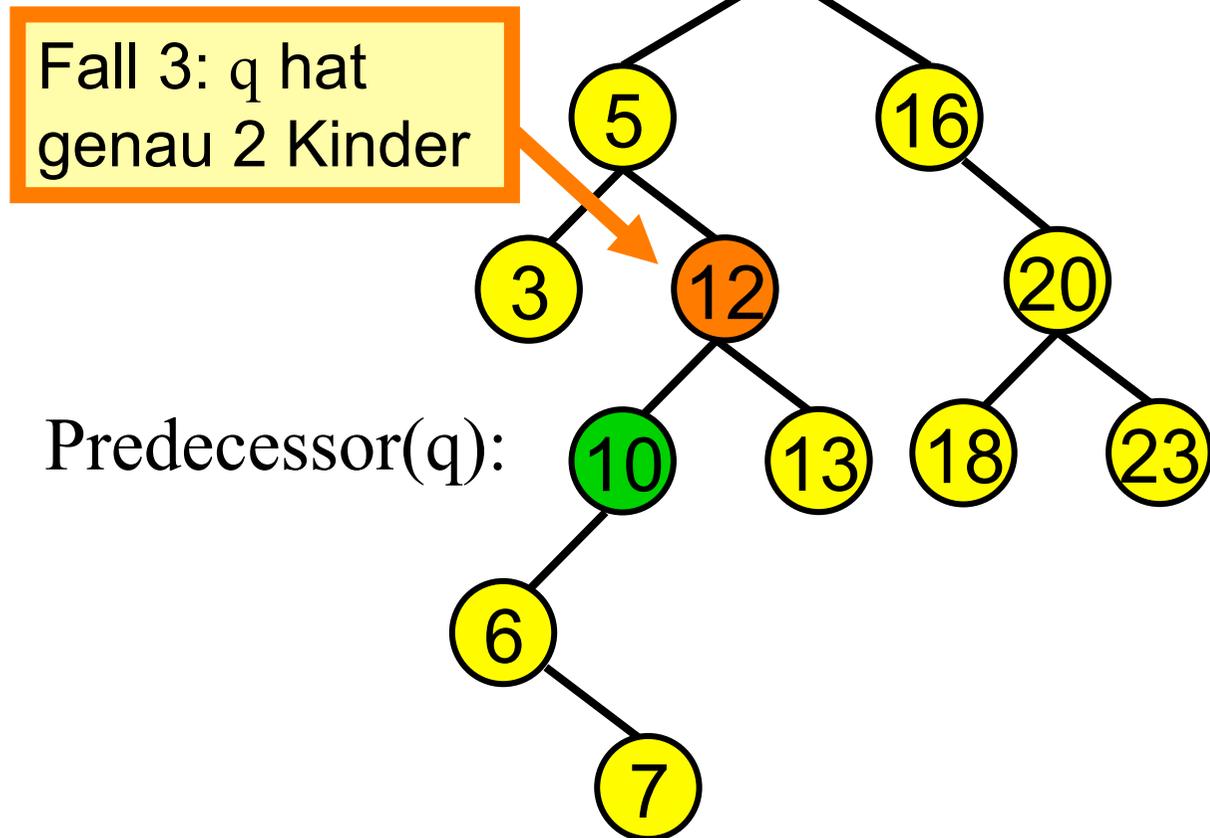
Beispiel für DELETE(r,q)

Fall 3: q hat genau 2 Kinder

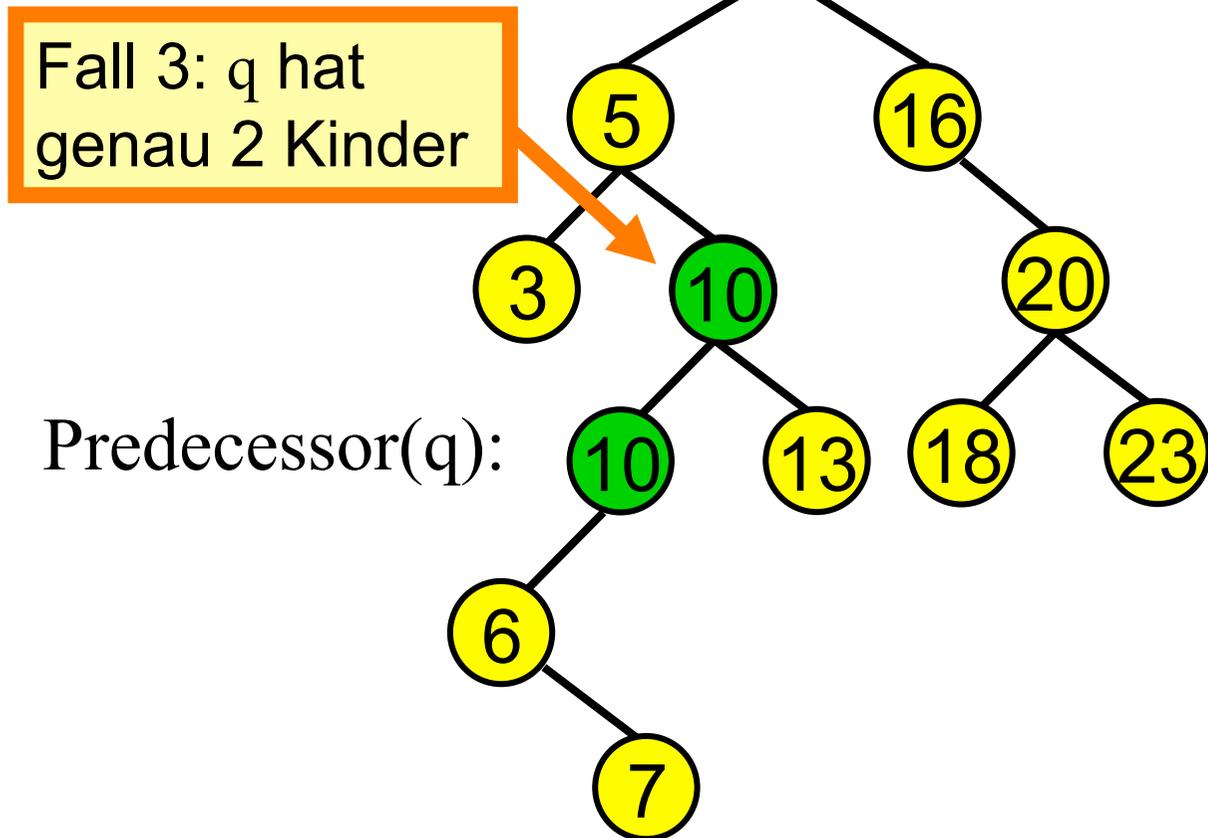


Predecessor(q):

Beispiel für DELETE(r,q)

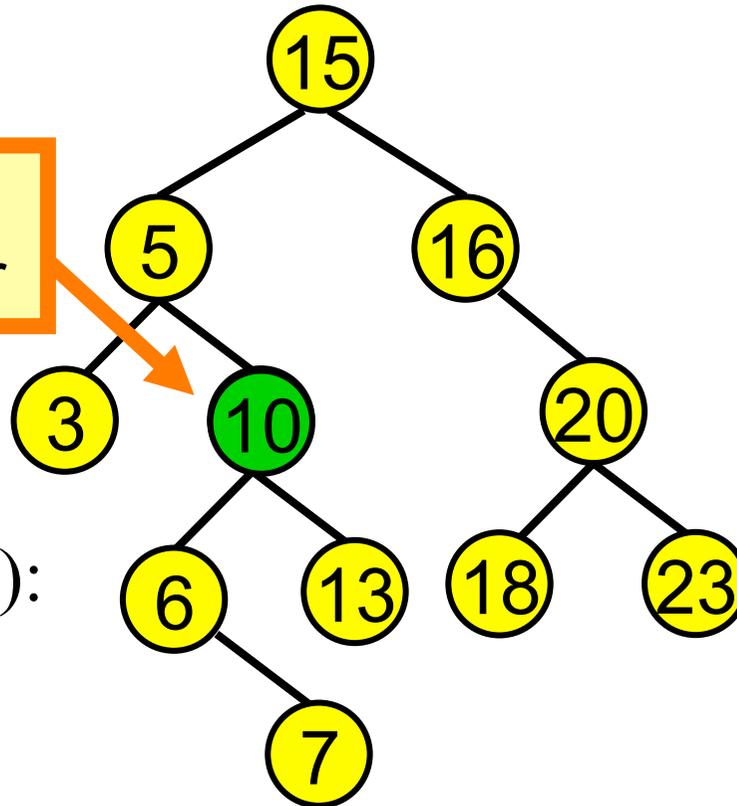


Beispiel für DELETE(r,q)



Beispiel für DELETE(r,q)

Fall 3: q hat genau 2 Kinder



Predecessor(q):

Implementierung von DELETE(r, q) in binären Suchbäumen

- Entfernt Knoten q im Baum mit Wurzel r

1. Suche nach $q.key \rightarrow$ Suche endet an Knoten v
2. Falls $v \neq \text{nil}$, dann
3. Falls v **keine Kinder** besitzt, dann: streiche v
4. Falls v **genau ein Kind** hat: ändere 2 Zeiger („herausschneiden“)
5. Falls v **zwei Kinder** hat, dann hat $y := \text{Predecessor}(v)$ kein rechtes Kind (warum??): Herausschneiden bzw. Entfernen von y und Ersetzen von q durch y .

Pseudocode von DELETE

Eingabe: Baum mit Wurzel `root`; Schlüssel `s` (existiert!)

Entfernt Knoten mit Schlüssel `s` aus Baum

Prozedur DELETE(`root,s`)

```
(1) var TreeNode r,p,q           // r wird herausgeschnitten
(2)  q := SEARCH(root,s)
(3) if q.left == nil or q.right==nil then
(4)   r := q
(5) else {
(6)   r:=PREDECESSOR(q)           // Vorgänger existiert
(7)   q.key := r.key; q.info := r.info //Umhängen der Daten von r nach q
(8) }
```

Pseudocode von DELETE ff

```
// jetzt löschen wir Knoten r mit maximal einem Kind
```

```
// lasse p auf Kind von r zeigen (falls Kind ex.)
```

```
(9) if r.left  $\neq$  nil then p := r.left else p:=r.right
```

```
(10) if p  $\neq$  nil then p.parent := r.parent //neuer Elter von p wird Elter von r
```

```
(11) if r.parent == nil then root := p // neuer Elter von p wird Elter von r
```

```
(12) else if r == r.parent.left then
```

```
(13)         r.parent.left := p           // p wird linker Nachfolger
```

```
(14)         else r.parent.right := p // p wird rechter Nachfolger
```

Analyse der Operationen

1. Alle Operationen benötigen eine Laufzeit von $O(h(T))$ für binäre Suchbäume, wobei $h(T)$ die Höhe des gegebenen Suchbaumes T ist.

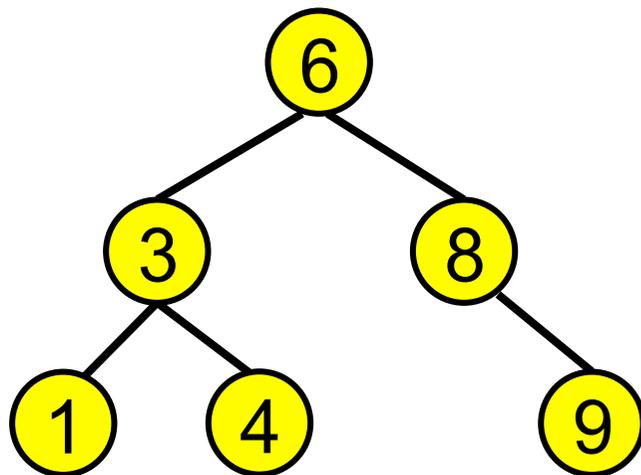
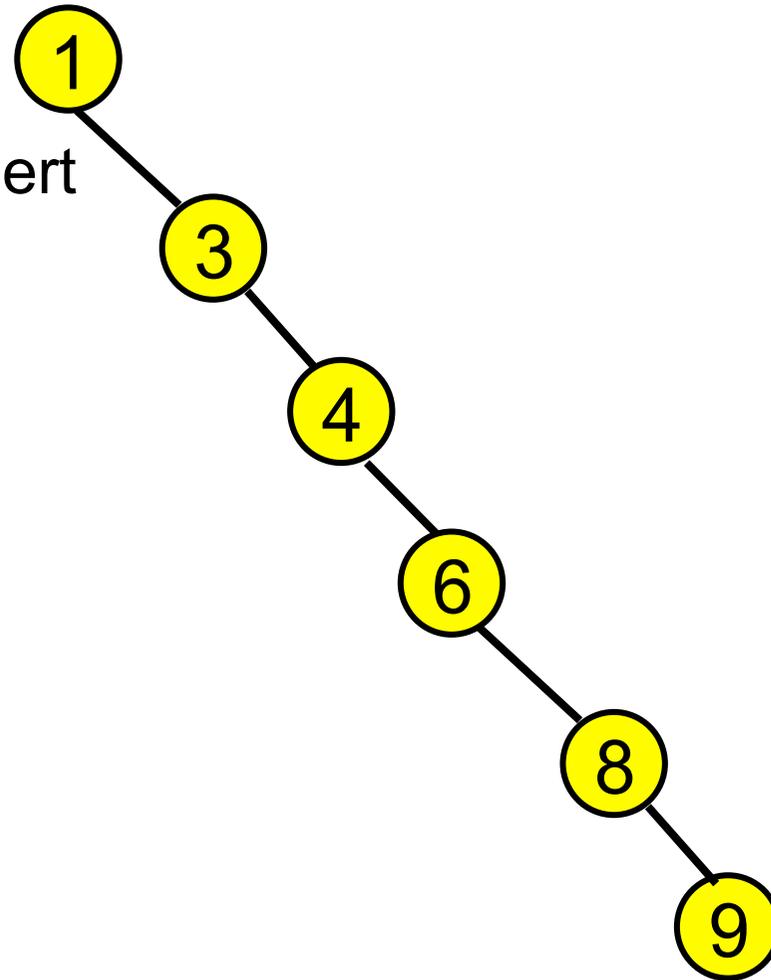
Frage: Wie hoch kann $h(T)$ für einen binären Suchbaum mit n Knoten sein?

Abhängigkeit der Höhe von der Einfügereihenfolge

Einfügereihenfolge: 1,3,4,6,8,9

Baum ist zu linearer Liste degeneriert
Problem: Suchzeit ist linear,
da $h(T)=n-1$

Einfügereihenfolge: 6,8,3,4,1,9



Kap. 4.3: AVL-Bäume

Balancierte Bäume

- AVL-Bäume sind sogenannte balancierte Bäume.
- **Balancierte Bäume** versuchen sich regelmäßig wieder „auszubalancieren“, um zu garantieren, dass die Höhe logarithmisch bleibt.
- Hierfür gibt es verschiedene Möglichkeiten:
 - **höhenbalancierte Bäume**
 - **gewichtsbalancierte Bäume**
 - **(a,b)-Bäume**
- Um die bisherigen binären Suchbäume von den balancierten Bäumen abzugrenzen, nennt man erstere auch **natürliche binäre Suchbäume**.

Balancierte Bäume

- **Höhenbalancierte Bäume:** Die Höhen der Unterbäume eines Knotens unterscheiden sich um höchstens eine Konstante **AVL-Bäume**
- **Gewichtsbalancierte Bäume:** Die Anzahl der Knoten in den Unterbäumen jedes Knotens unterscheidet sich höchstens um eine Konstante.
- **(a,b)-Bäume ($2 \leq a \leq b$):** Jeder innere Knoten (außer der Wurzel) hat zwischen a und b Kinder und alle Blätter haben den gleichen Abstand zur Wurzel.

B-Bäume: s. Kap. 4.4

Definitionen für AVL-Bäume

- **Höhe eines Knotens:** Länge eines längsten Pfades von v zu einem Nachkommen von v
- **Höhe eines Baumes:** Höhe seiner Wurzel
- **Höhe eines leeren Baumes:** -1
- **Balance eines Knotens:** $\text{bal}(v) = h_2 - h_1$, wobei h_1 und h_2 die Höhe des linken bzw. rechten Unterbaumes von v bezeichnen
- **v heißt balanciert:** wenn $\text{bal}(v) \in \{-1, 0, +1\}$, sonst heißt v unbalanciert

Bezeichnungen

$$h(3)=1$$

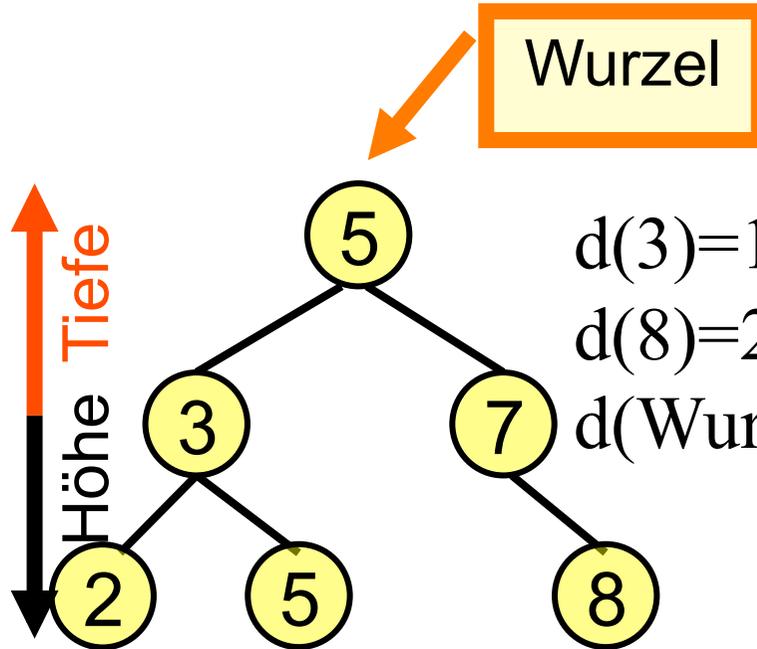
$$h(8)=0$$

$$h(\text{Wurzel})=2$$

$$\text{bal}(3)=0$$

$$\text{bal}(7)=1$$

$$\text{bal}(\text{Wurzel})=0$$



$$d(3)=1$$

$$d(8)=2$$

$$d(\text{Wurzel})=0$$

$$h(T_3)=1$$

$$h(T_8)=0$$

$$h(T_{\text{Wurzel}})=2$$

Beispiele für balanciert üben!

Höhe $h(v)$: Länge eines längsten Pfades von v zu einem Nachkommen von v

Höhe $h(T_r)$ eines (Teil-)baumes T_r mit Wurzel r :

$\max \{ d(v) : v \text{ ist Knoten in } T_r \} = \text{Höhe seiner Wurzel}$

AVL-Bäume

- **Definition:** Ein AVL-Baum ist ein binärer Suchbaum, bei dem alle Knoten balanciert sind.

- AVL-Bäume wurden 1962 eingeführt von G.M. Adelson-Velskii und Y.M. Landis
- AVL-Bäume sind höhenbalanciert.
- Man kann zeigen:

- **Theorem:** Ein AVL-Baum mit n Kindern hat Höhe $O(\log n)$.

Implementierungen der Operationen

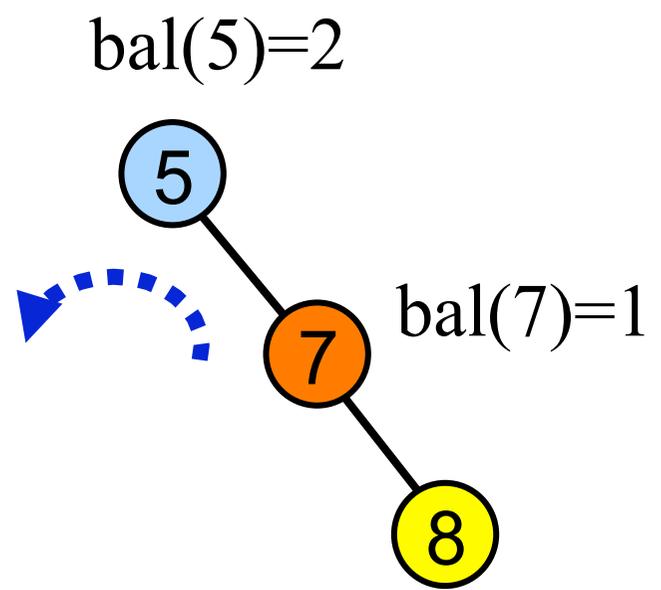
- **Search**
- **Minimum**
- **Maximum**
- **Successor**
- **Predecessor**
- genau wie bei den natürlichen binären Suchbäumen.

Implementierung der Operationen Insert und Delete in AVL-Bäumen

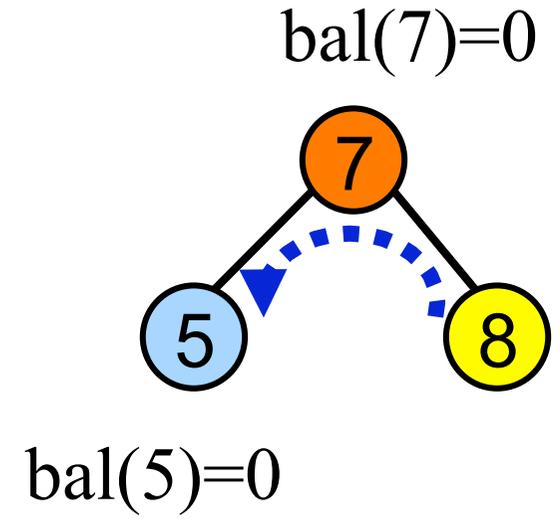
- **Idee:**
- zunächst wie bei den natürlichen binären Suchbäumen.
- Falls Baum nicht mehr balanciert ist, dann wissen wir, dass ein Knoten u auf dem Suchpfad existiert mit $\text{bal}(u) \in \{-2, +2\}$
- Wir rebalancieren den Baum nun an dieser Stelle, so dass er danach wieder balanciert ist.

Beispiel 1:

Rotationen (1)



Rotation
nach
links
an 5



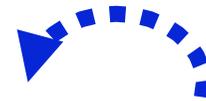
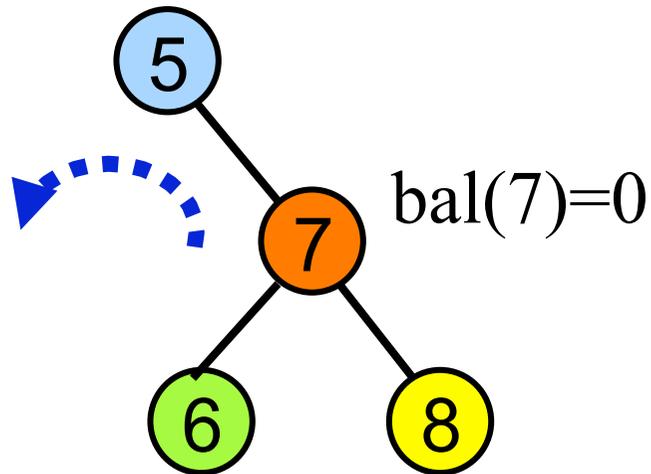
5 ist unbalanciert

5, 7, 8 sind balanciert

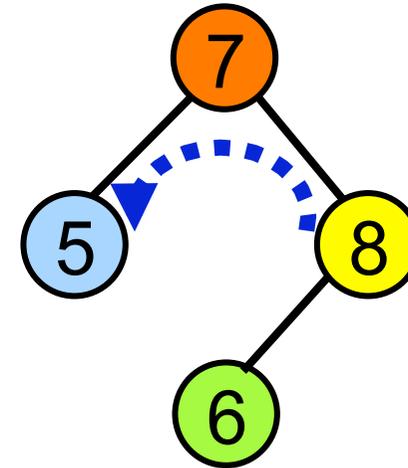
Beispiel 2:

Rotationen (2)

$bal(5)=2$



Rotation
nach
links
an 5

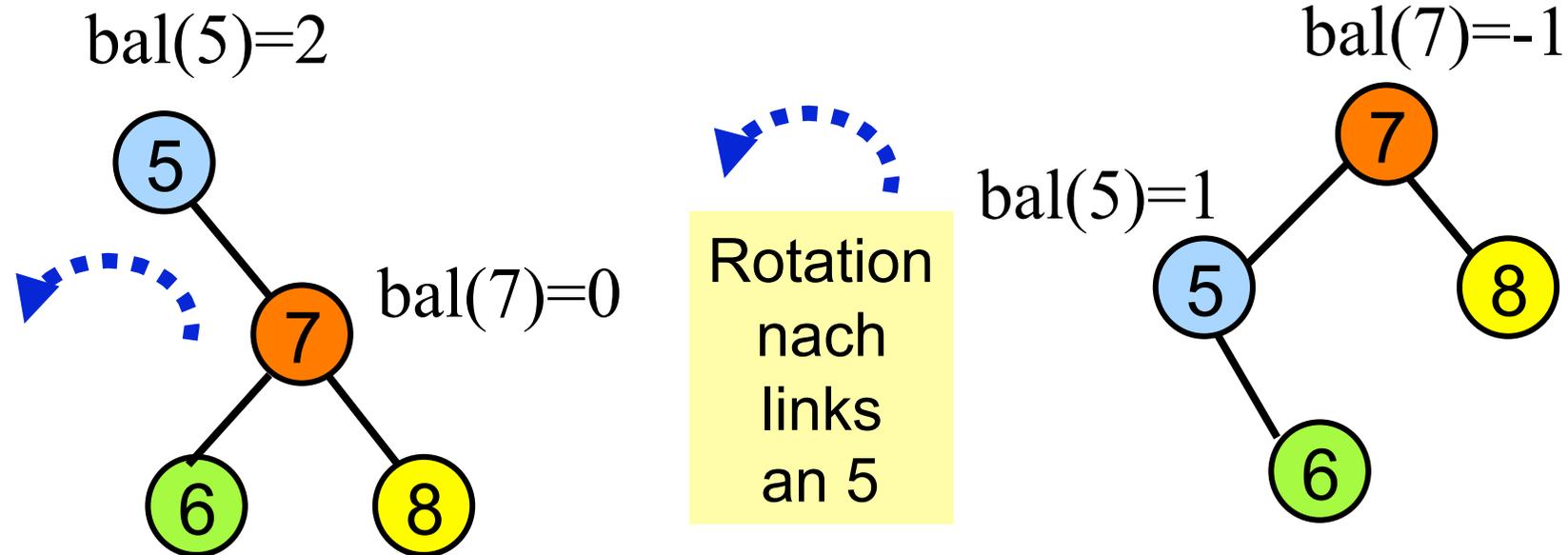


5 ist unbalanciert

falsch: die 6 muss mitwandern

5, 7, 8 sind balanciert

Rotationen (3)



so korrekt! Warum eigentlich?

5 ist unbalanciert

5, 7, 8 sind balanciert

Die Suchbaumeigenschaft bleibt bei einer Rotation erhalten.

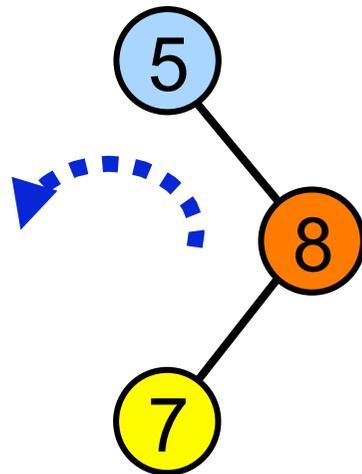
Nach einer Rotation entsteht wieder ein binärer Suchbaum

Achtung: dies ist nicht so wenn gleiche Schlüssel enthalten sind

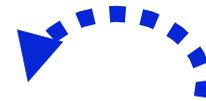
Beispiel 3:

Rotationen (4)

$$\text{bal}(5)=2$$

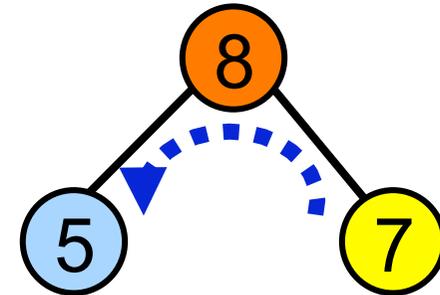


$$\text{bal}(8)=-1$$



Rotation
nach
links
an 5

$$\text{bal}(8)=0$$

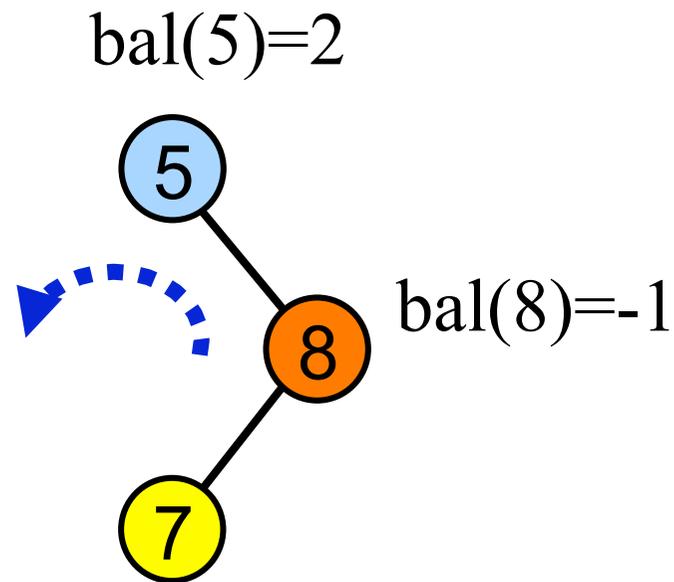


$$\text{bal}(5)=0$$

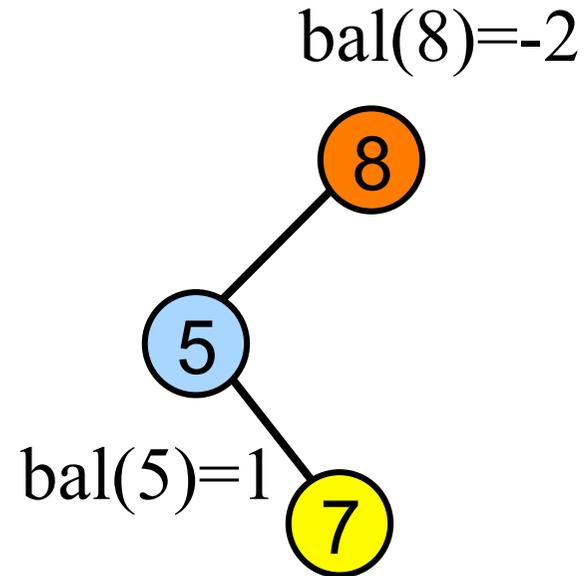
5 ist unbalanciert

falsch: Das war keine
Rotation!!!

Rotationen (5)



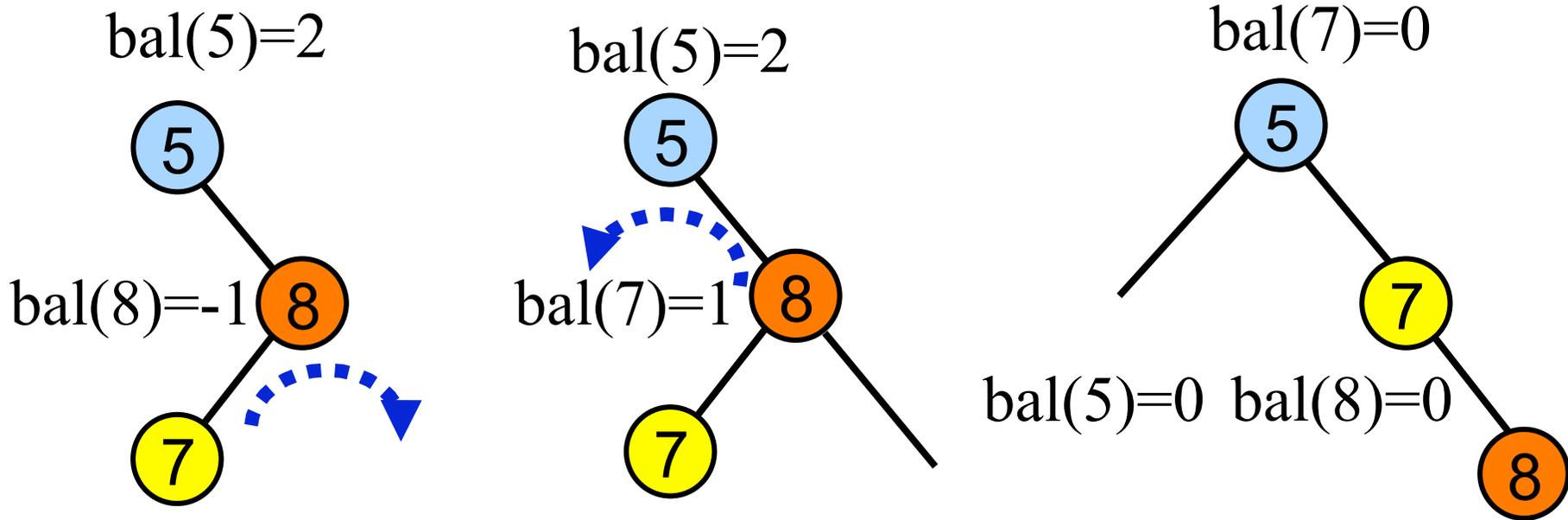
Rotation
nach
links
an 5



5 ist unbalanciert

Diese Rotation hat
nichts genützt!

Rotationen (6)

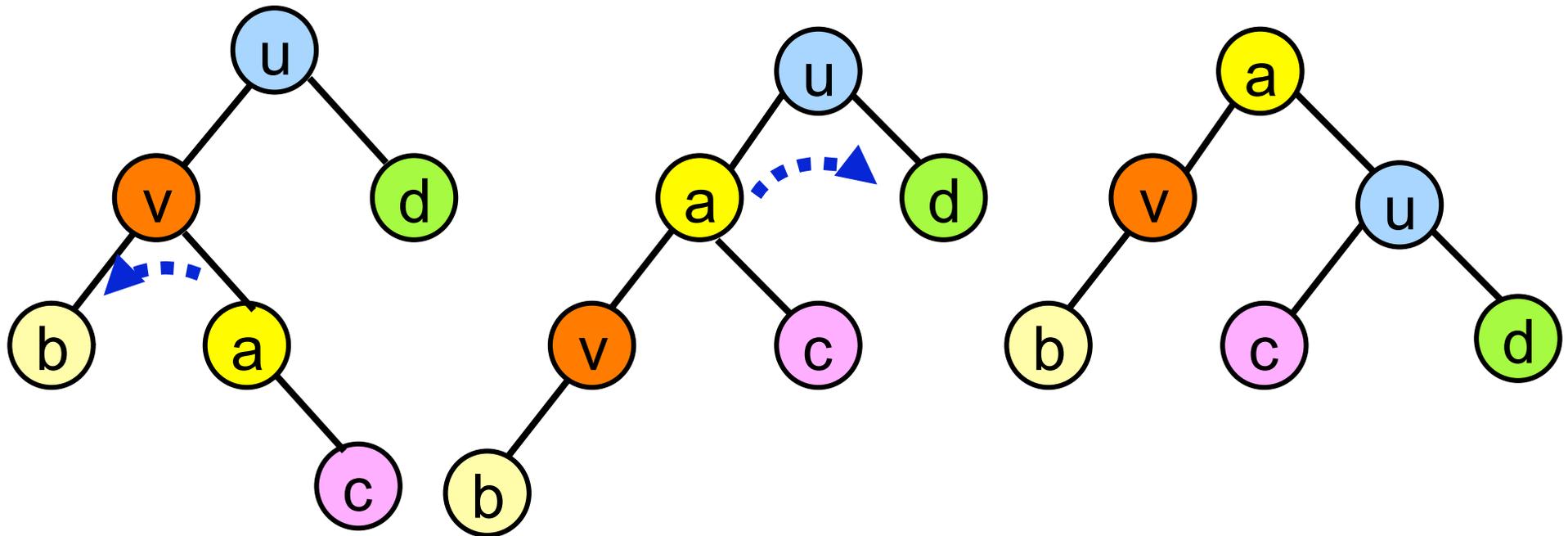


Rotation nach
rechts an 8

Rotation nach
links an 5

Doppelrotation Rechts-Links notwendig!

Doppelrotationen



Definition

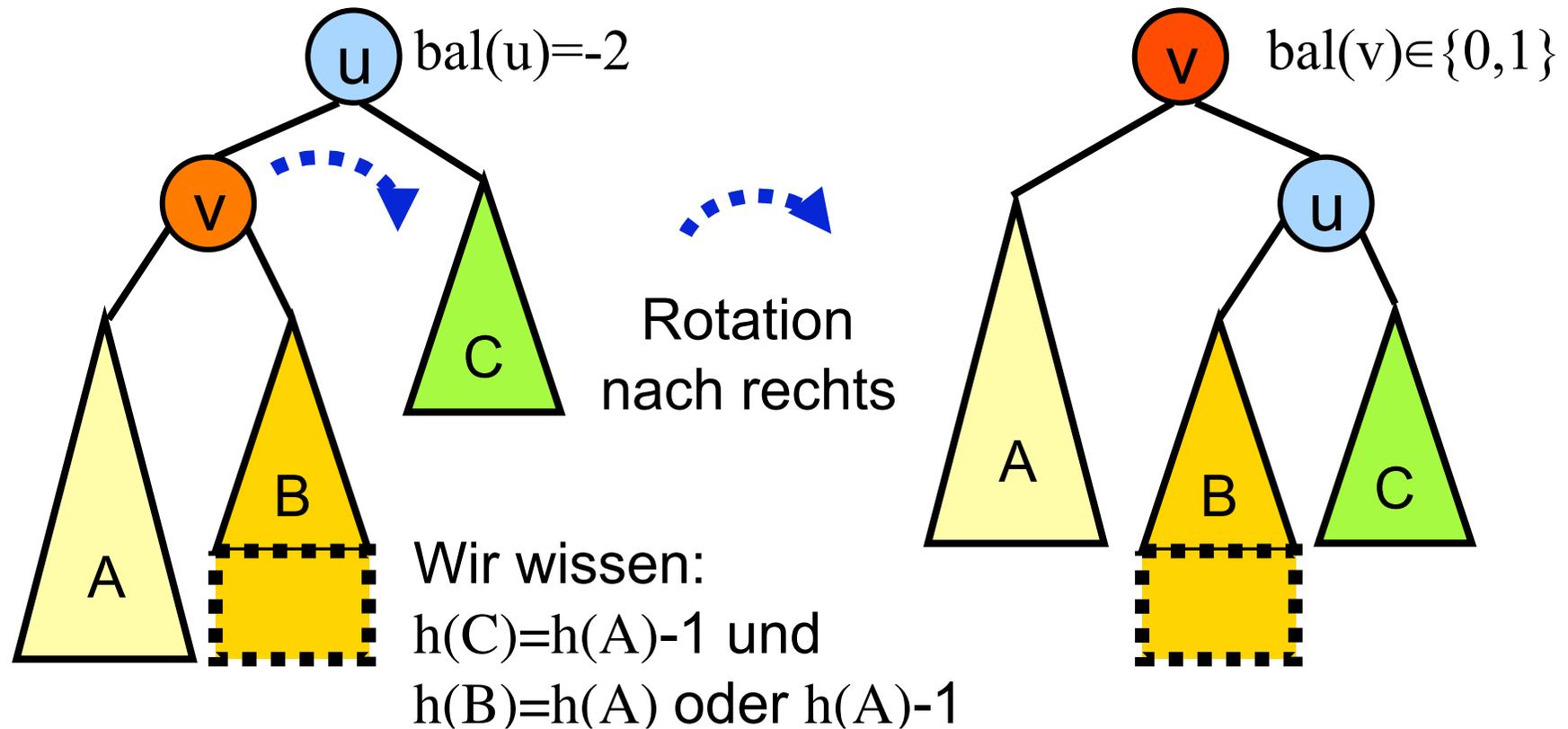
- **AVL-Ersetzung:** Operation (z.B. Insert, Delete), die einen Unterbaum T eines Knotens z durch einen modifizierten AVL-Baum ersetzt, dessen Höhe um höchstens 1 von der Höhe von T abweicht.

Rebalancierung

- Wir betrachten einen AVL-Baum unmittelbar nach einer AVL-Ersetzung (Einfügung bzw. Entfernen eines Knotens).
- Sei u ein unbalancierter Knoten **maximaler Tiefe**, d.h. $\text{bal}(u) \in \{-2, +2\}$.
- Sei T der maximale Unterbaum mit Wurzel u in T .
- Wir unterscheiden vier verschiedene Situationen.

1. Fall: $\text{bal}(u) = -2$

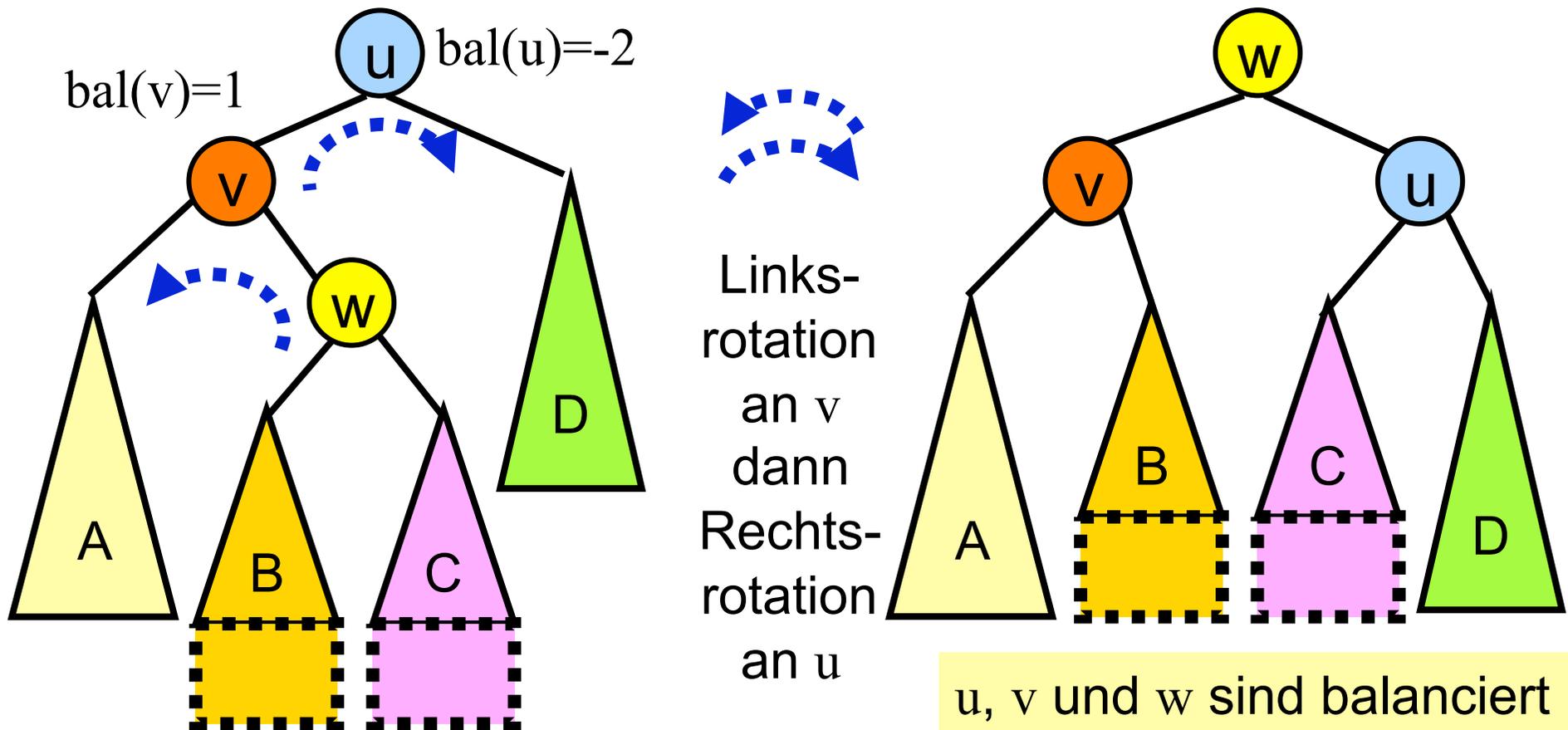
- Sei v das linke Kind von u (existiert!)
- **Fall 1.1:** $\text{bal}(v) \in \{-1, 0\}$: Rotation nach rechts an u



- Suchbaumeigenschaft bleibt erhalten; u und v sind balanciert
- Für die Knoten unterhalb hat sich nichts geändert.

1. Fall: $\text{bal}(u) = -2$ ff.

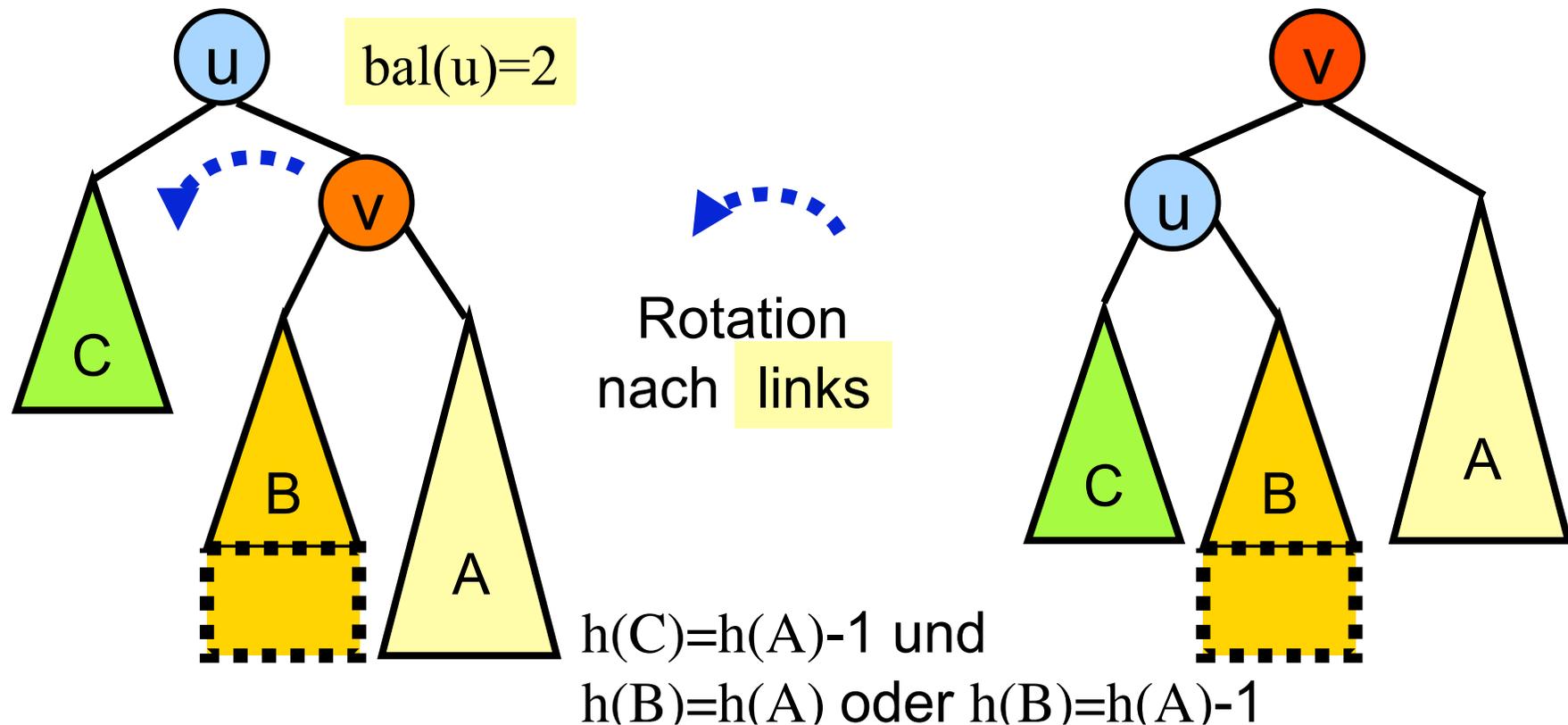
- Sei v das linke Kind von u (existiert!)
- **Fall 1.2: $\text{bal}(v) = +1$:** Links-Rechtsrotation an u



$h(D) = h(A)$ und ($h(B) = h(A)$ oder $h(C) = h(A)$)

2. Fall: $\text{bal}(u)=+2$

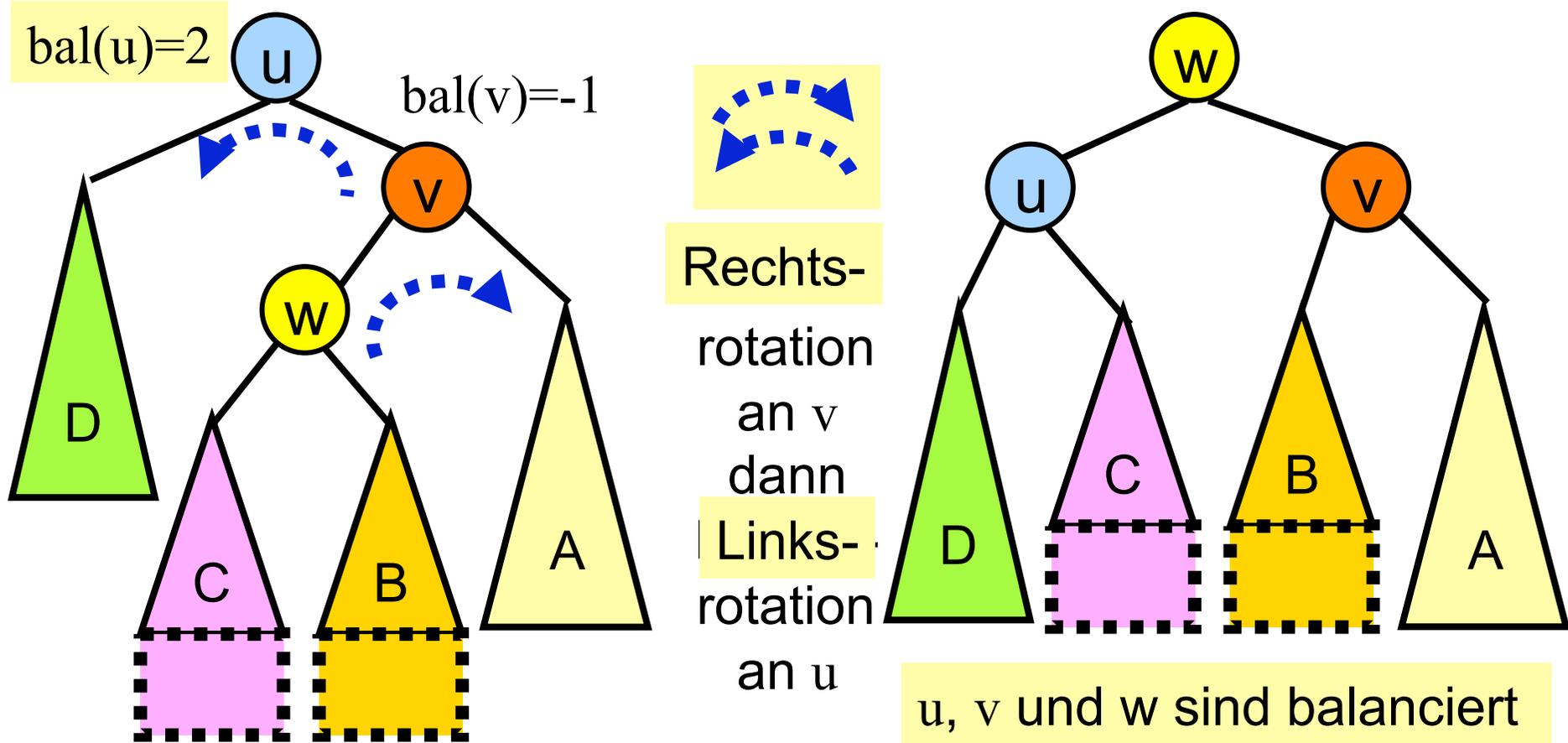
- Sei v das rechte Kind von u (existiert!)
- **Fall 2.1:** $\text{bal}(v) \in \{0, 1\}$: Rotation nach links an u



- Inorder-Reihenfolge bleibt erhalten und u und v sind balanciert
- Für die Knoten unterhalb hat sich nichts geändert.

2. Fall: $\text{bal}(u) = +2$ ff.

- Sei v das rechte Kind von u (existiert!)
- **Fall 2.2:** $\text{bal}(v) = -1$ Rechts-Links rotation an u



$h(D)=h(A)$ und ($h(B)=h(A)$ oder $h(C)=h(A)$)

Rebalancierung ff.

- Seien T der maximale Unterbaum mit Wurzel u in T vor der Rebalancierung, T' der gleiche Teilbaum nach einer einfachen Rotation und T'' nach einer Doppelrotation.

- Für die einfache Rotation gilt: $h(T') - h(T) \in \{-1, 0\}$
- Im Falle der Doppelrotation gilt: $h(T'') - h(T) = -1$

- Alle Transformationen (Einfach- und Doppelrotationen) kann man also als eine AVL-Ersetzung auffassen.

Rebalancierung ff.

- Nach der AVL-Ersetzung gilt:
- Die Unterbäume mit Wurzel u und v sind danach balanciert. Für die Unterbäume unterhalb hat sich die Balancierung nicht geändert.

- Bestimme nun den nächsten unbalancierten Knoten maximaler Tiefe. Diese Tiefe ist nun kleiner als vorher.
- Wiederhole die Rebalancierung (AVL-Ersetzung) solange, bis alle Knoten balanciert sind.

- Das Verfahren konvergiert nach $O(h(T))$ Iterationen zu einem gültigen AVL-Baum.

Rebalancierung ff.

- Die Insert-Operation unterscheidet sich nur insofern von der Delete-Operation, dass hier ein einziger Rebalancierungsschritt genügt.
- Bei der Delete-Operation hingegen kann es sein, dass mehrere Rebalancierungsschritte notwendig sind.

Implementierungen:

vereinfacht: ohne parent

RotateRight(u)

Function $h(u):int$

If $u=nil$ **then return -1 else return** $u.height$

Function RotateRight(u):TreeNode

(1) $v:=u.left$ //Bestimme v

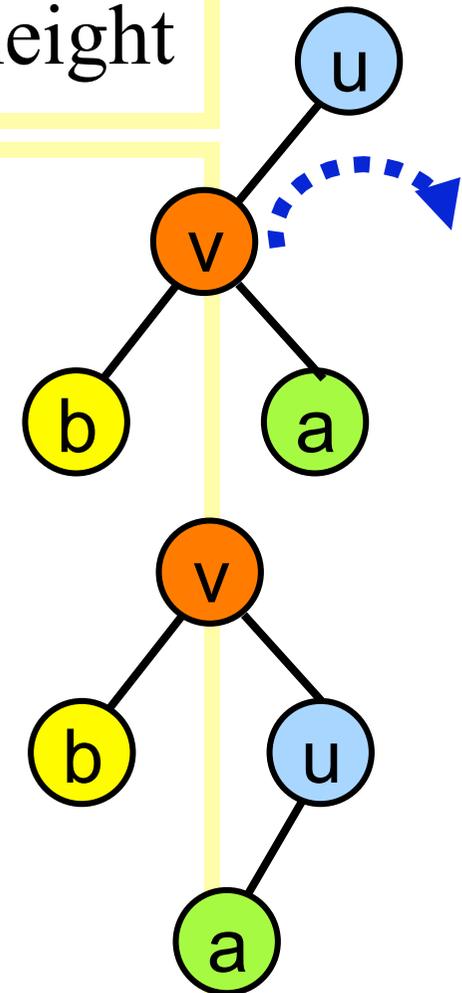
(2) $u.left:=v.right$

(3) $v.right:=u$

(4) $u.height:=\max(h(u.left),h(u.right))+1$

(5) $v.height:=\max(h(v.left),h(u))+1$

(6) return v //neue Wurzel



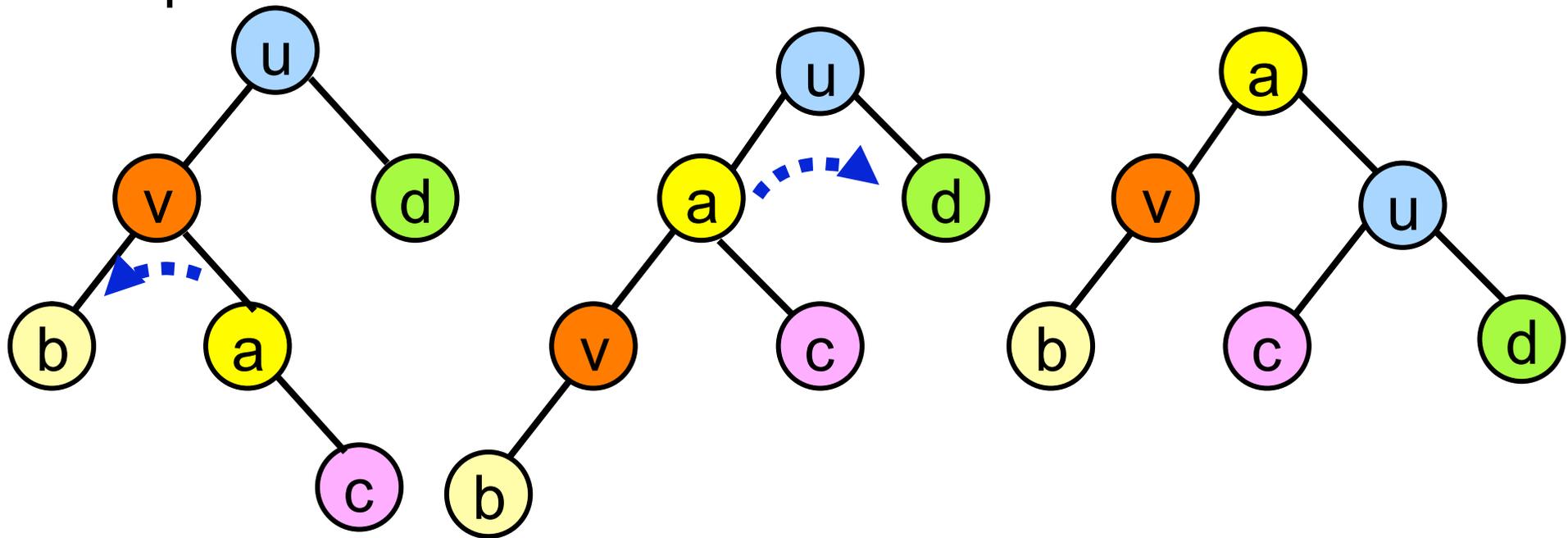
RotateLeftRight(u)

Function RotateLeftRight(u):TreeNode

(1) u.left:=RotateLeft(u.left)

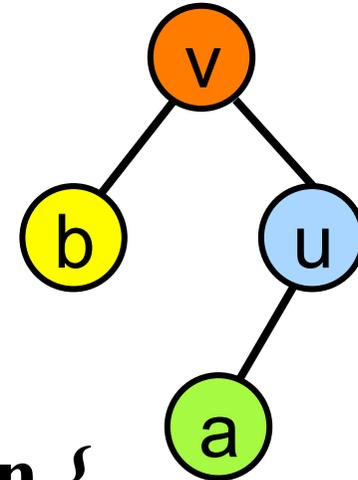
(2) return RotateRight(u)

Beispiel:



INSERTAVL(p,q)

- (1) **If** $p == \text{nil}$ **then** INSERT(p,q)
- (2) **else**
- (3) **If** $q.\text{key} < p.\text{key}$ **then** {
- (4) INSERTAVL(p.left,q)
- (5) **if** $h(p.\text{right}) - h(p.\text{left}) == -2$ **then** {
- (6) **if** $h(p.\text{left}.\text{left}) > h(p.\text{left}.\text{right})$ **then**
- (7) $p = \text{RotateRight}(p)$
- (8) **else** $p = \text{RotateLeftRight}(p)$
- (9) } }



INSERTAVL(p,q) ff

(10) Else {

(11) If $q.key > p.key$ then

(12) INSERTAVL(p.right,q)

(13) if $h(p.right) - h(p.left) == 2$ then

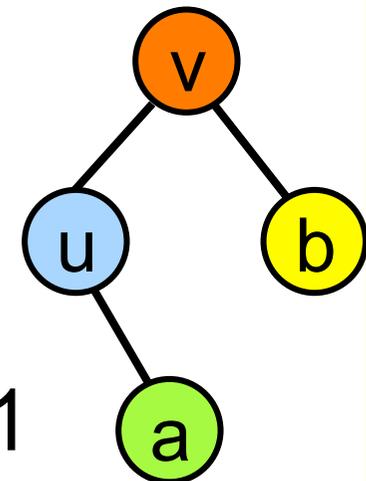
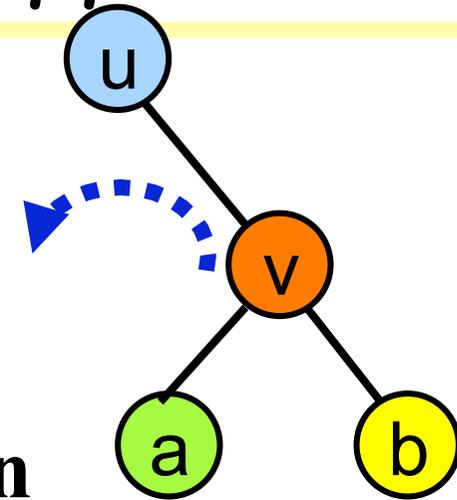
(14) if $h(p.right.right) > h(p.right.left)$ then

(15) p=RotateLeft(p)

(16) else p=RotateRightLeft(p)

(17) } }

(18) p.height:=max(h(p.left),h(p.right))+1



Analyse

- Rotationen und Doppelrotationen können in konstanter Zeit ausgeführt werden.
- Eine Rebalancierung wird höchstens einmal an jedem Knoten auf dem Pfad vom eingefügten oder gelöschten Knoten zur Wurzel durchgeführt.

- Insert und Delete-Operationen sind in einem AVL-Baum in Zeit $O(\log n)$ möglich.

- AVL-Bäume unterstützen die Operationen Suchen, Insert, Delete, Minimum, Maximum, Successor, Predecessor, in Zeit $O(\log n)$.

Diskussion

- Wenn oft gesucht und selten ungeordnet wird, sind AVL-Bäume günstiger als natürliche binäre Suchbäume.
- Falls jedoch fast jeder Zugriff ein Einfügen oder Entfernen ist (hinreichend zufällig), dann sind natürliche binäre Suchbäume vorzuziehen.

Java-Applets

- Es gibt zu AVL-Bäumen sehr schöne Java-Applets, die die Doppelrotationen sehr schön darstellen, z.B. die Studiumsseiten von Math^e(Prism)^a:
- <http://www.matheprisma.uni-wuppertal.de/Module/BinSuch/index.html>

ENDE

Zum Ausschneiden und Üben

1

2

3

4

5

6

7

8

9

10

11

12

