

SS 2021

# Algorithmen und Datenstrukturen

## 4. Kapitel

### Suchbäume

Martin Dietzfelbinger

Mai 2021

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp **Wörterbuch****.

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp **Wörterbuch****.

Zu diesem Datentyp gehören:

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp **Wörterbuch****.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

- *empty()*: leeres Wörterbuch erzeugen

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

- *empty()*: leeres Wörterbuch erzeugen
- *lookup( $f, x$ )*: falls  $x \in S$ :  $f(x)$  ausgeben (sonst: „undefined“)

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

- $empty()$ : leeres Wörterbuch erzeugen
- $lookup(f, x)$ : falls  $x \in S$ :  $f(x)$  ausgeben (sonst: „undefined“)
- $insert(f, x, r)$ :  $f(x) := r$  setzen (für  $x$  neu in  $S$ , oder  $f(x)$  auf  $r$  aktualisieren)

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

- $empty()$ : leeres Wörterbuch erzeugen
- $lookup(f, x)$ : falls  $x \in S$ :  $f(x)$  ausgeben (sonst: „undefined“)
- $insert(f, x, r)$ :  $f(x) := r$  setzen (für  $x$  neu in  $S$ , oder  $f(x)$  auf  $r$  aktualisieren)
- $delete(f, x)$ :  $x$  aus  $S = \text{Def}(f)$  löschen, falls  $x \in S$

---

## 4.1 Binäre Suchbäume

Binäre Suchbäume **implementieren den Datentyp Wörterbuch**.

Zu diesem Datentyp gehören:

Schlüsselmenge  $U$  und Wertemenge  $R$

Wörterbuch:  $f: S \rightarrow R$ , wobei  $S \subseteq U$  endlich.

- $empty()$ : leeres Wörterbuch erzeugen
- $lookup(f, x)$ : falls  $x \in S$ :  $f(x)$  ausgeben (sonst: „undefined“)
- $insert(f, x, r)$ :  $f(x) := r$  setzen (für  $x$  neu in  $S$ , oder  $f(x)$  auf  $r$  aktualisieren)
- $delete(f, x)$ :  $x$  aus  $S = \text{Def}(f)$  löschen, falls  $x \in S$

Hier **zusätzlich** nötig:  $U$  ist durch  $<$  (**total geordnet**).

---

*Beispiel:*  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung)

---

*Beispiel:*  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

---

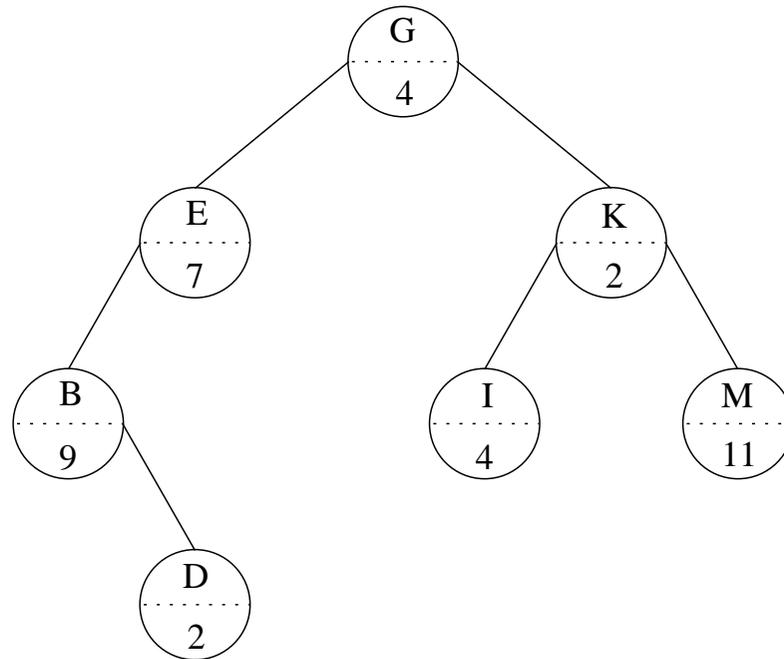
*Beispiel:*  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

Ein binärer Suchbaum  $T$ :

---

Beispiel:  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

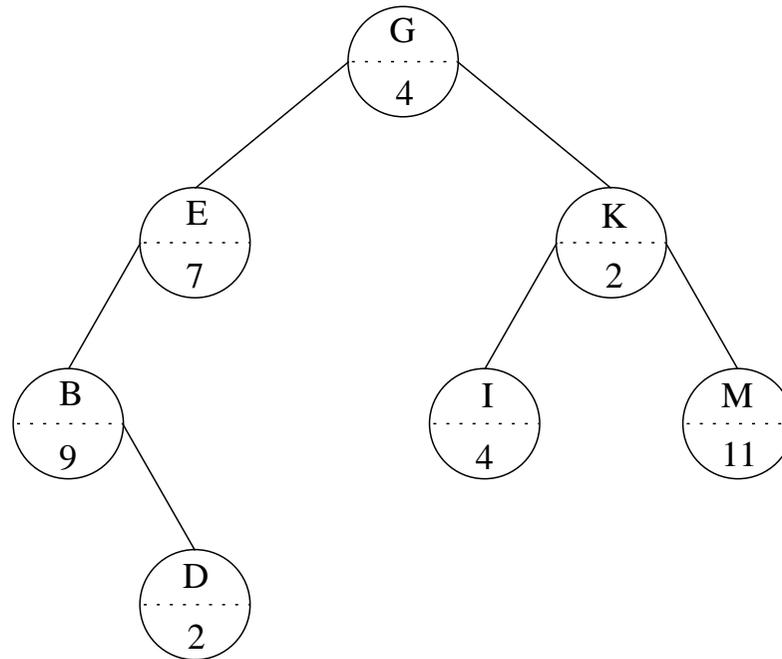
Ein binärer Suchbaum  $T$ :



---

Beispiel:  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

Ein binärer Suchbaum  $T$ :

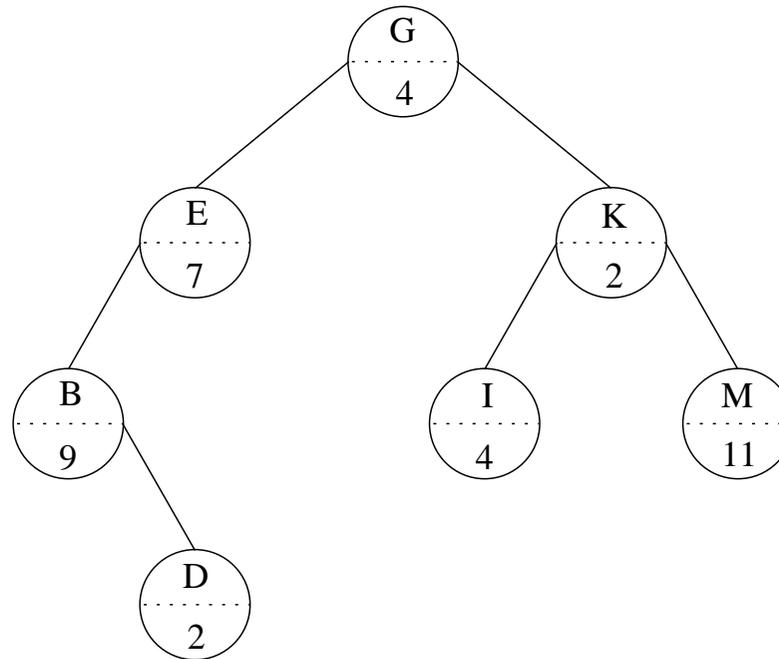


Dargestellte Funktion  $f = f_T$ :

---

Beispiel:  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

Ein binärer Suchbaum  $T$ :



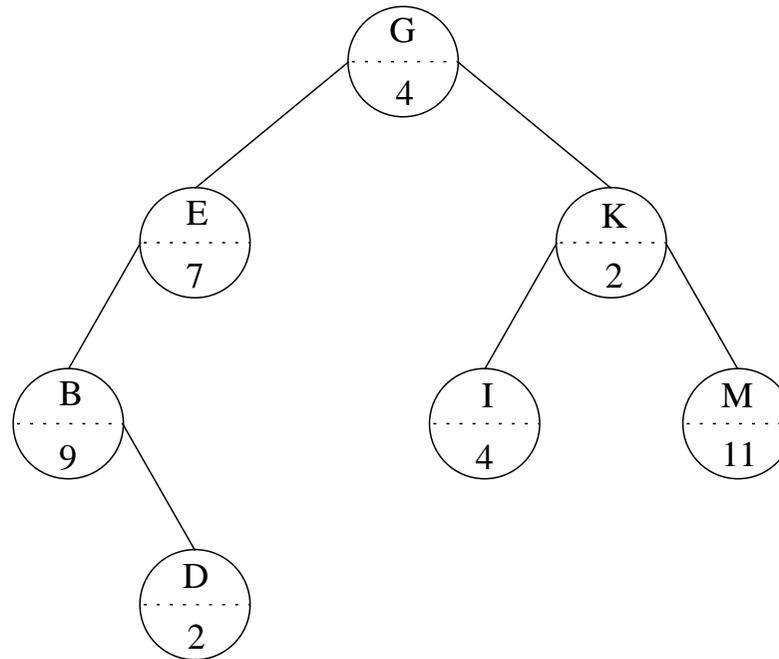
Dargestellte Funktion  $f = f_T$ :

$S = \text{Def}(f) = \{B, D, E, G, I, K, M\}$ ,

---

Beispiel:  $U = \{A, B, C, \dots, Z\}$  (Standardsortierung),  $R = \mathbb{N}$ .

Ein binärer Suchbaum  $T$ :



Dargestellte Funktion  $f = f_T$ :

$S = \text{Def}(f) = \{B, D, E, G, I, K, M\}$ ,  $f(B) = 9$ ,  $f(D) = 2$ , usw.

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

## Definition 4.1.1

Ein Binärbaum  $T$  mit Knoteneinträgen aus  $U \times R$

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

## Definition 4.1.1

Ein Binärbaum  $T$  mit Knoteneinträgen aus  $U \times R$  heißt ein **binärer Suchbaum**<sup>\*</sup>, falls **für jeden Knoten**  $v$  in  $T$  gilt:

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

## Definition 4.1.1

Ein Binärbaum  $T$  mit Knoteneinträgen aus  $U \times R$  heißt ein **binärer Suchbaum\***, falls **für jeden Knoten**  $v$  in  $T$  gilt:

für alle Knoten  $w$  im **linken** Unterbaum von  $T_v$ :  $key(w) < key(v)$

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

## Definition 4.1.1

Ein Binärbaum  $T$  mit Knoteneinträgen aus  $U \times R$  heißt ein **binärer Suchbaum\***, falls **für jeden Knoten  $v$  in  $T$  gilt:**

für alle Knoten  $w$  im **linken** Unterbaum von  $T_v$ :  $key(w) < key(v)$

für alle Knoten  $w$  im **rechten** Unterbaum von  $T_v$ :  $key(v) < key(w)$

---

# Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten  $v$ ):

- $key(v)$ : Schlüssel (aus  $U$ , mit  $<$  Totalordnung auf  $U$ )
- $data(v)$ : Daten, Wert (aus  $R$ ) (oft: Zeiger, Referenz)

## Definition 4.1.1

Ein Binärbaum  $T$  mit Knoteneinträgen aus  $U \times R$  heißt ein **binärer Suchbaum\***, falls **für jeden Knoten  $v$**  in  $T$  gilt:

für alle Knoten  $w$  im **linken** Unterbaum von  $T_v$ :  $key(w) < key(v)$

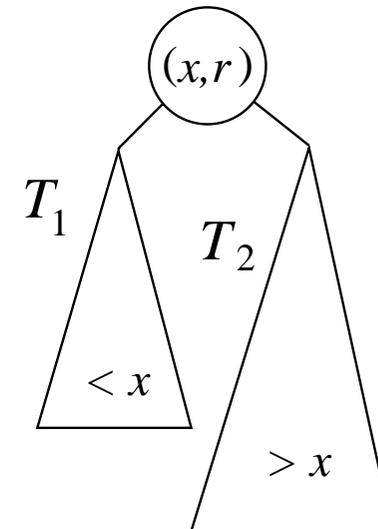
für alle Knoten  $w$  im **rechten** Unterbaum von  $T_v$ :  $key(v) < key(w)$

---

engl.: *binary search tree*. Abk.:  **$(U, R)$ -BSB** oder  **$(U, R)$ -BST**

---

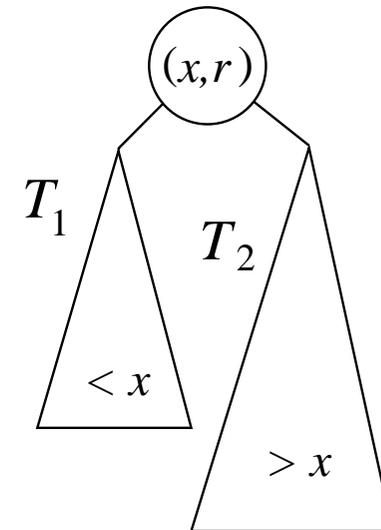
## Alternative Definition: Rekursive Auffassung.



---

**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

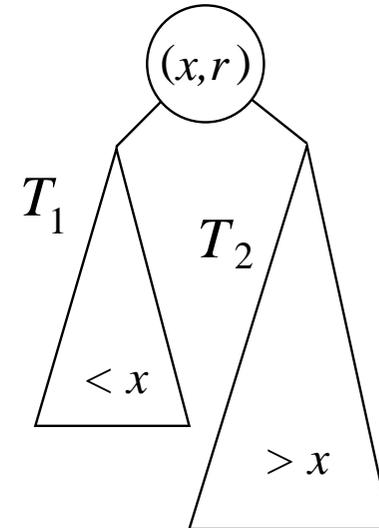


---

**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmengen  $S_1$  bzw.  $S_2 \subseteq U$ ,



---

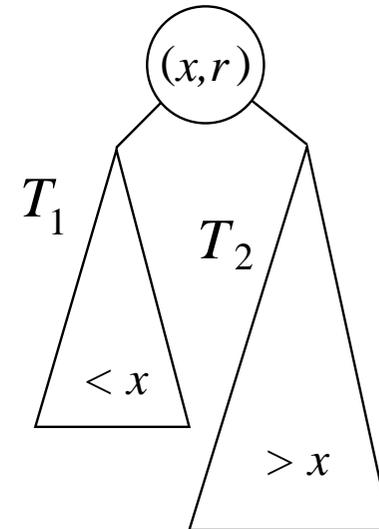
**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1$  bzw.  $S_2 \subseteq U$ ,

**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$



---

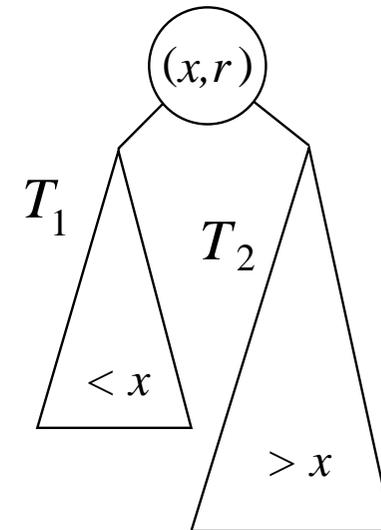
**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1$  bzw.  $S_2 \subseteq U$ ,

**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$  ein  $(U, R)$ -BSB



---

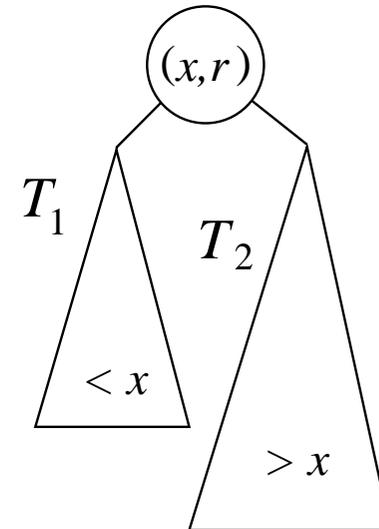
**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1$  bzw.  $S_2 \subseteq U$ ,

**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$  ein  $(U, R)$ -BSB  
mit Schlüsselmenge



---

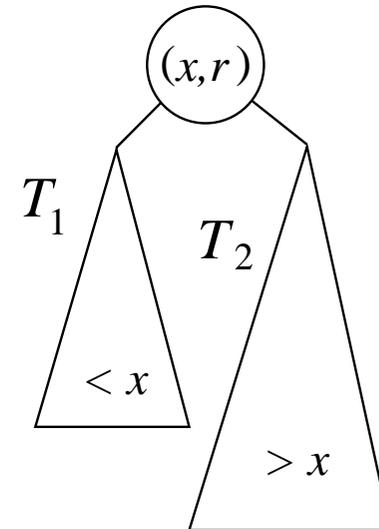
**Alternative Definition:** Rekursive Auffassung.

(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmengen  $S_1$  bzw.  $S_2 \subseteq U$ ,

**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$  ein  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1 \cup \{x\} \cup S_2$ .



---

**Alternative Definition:** Rekursive Auffassung.

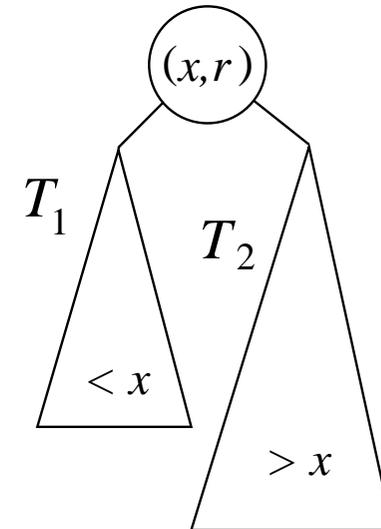
(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1$  bzw.  $S_2 \subseteq U$ ,

**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$  ein  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1 \cup \{x\} \cup S_2$ .

[(iii) Nichts sonst ist ein  $(U, R)$ -BSB.]



---

**Alternative Definition:** Rekursive Auffassung.

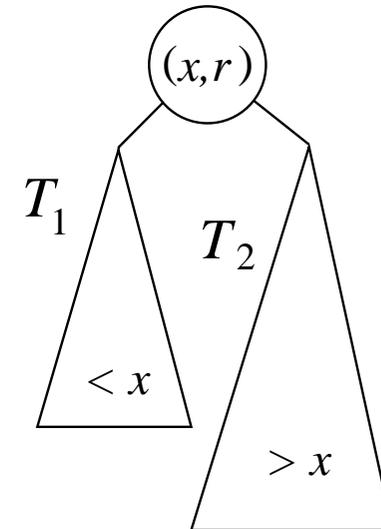
(i)  $\square$  ist  $(U, R)$ -BSB, mit Schlüsselmenge  $S = \emptyset$ .  $\square$

(ii) **Wenn**  $x \in U$  und  $r \in R$  **und**  
 $T_1, T_2$  sind  $(U, R)$ -BSB  
mit Schlüsselmengen  $S_1$  bzw.  $S_2 \subseteq U$ ,

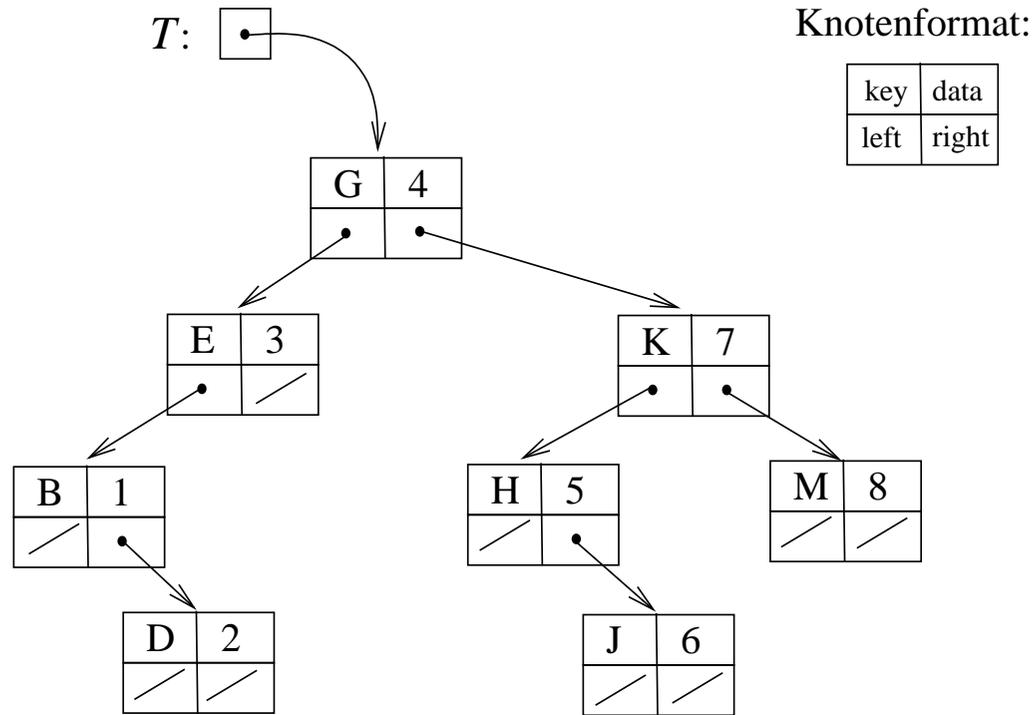
**und** wenn  $y < x$  für alle  $y \in S_1$   
und  $x < z$  für alle  $z \in S_2$ ,

**dann** ist  $(T_1, (x, r), T_2)$  ein  $(U, R)$ -BSB  
mit Schlüsselmenge  $S_1 \cup \{x\} \cup S_2$ .

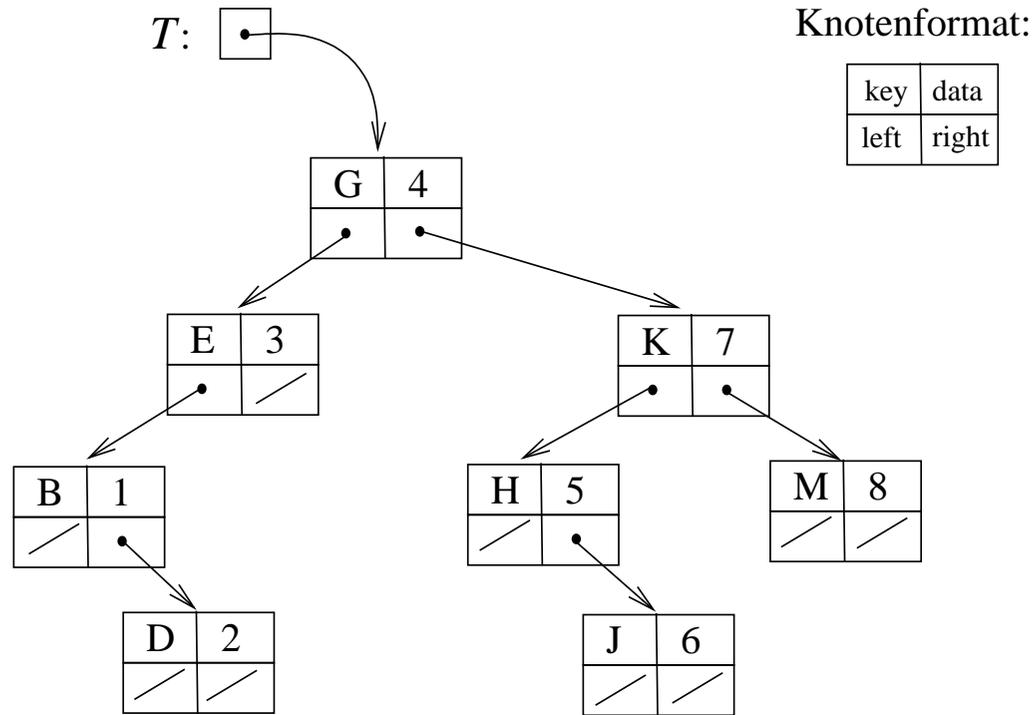
[(iii) Nichts sonst ist ein  $(U, R)$ -BSB.]



Falls (ii):  $key(T) := x$  und  $data(T) := r$ .

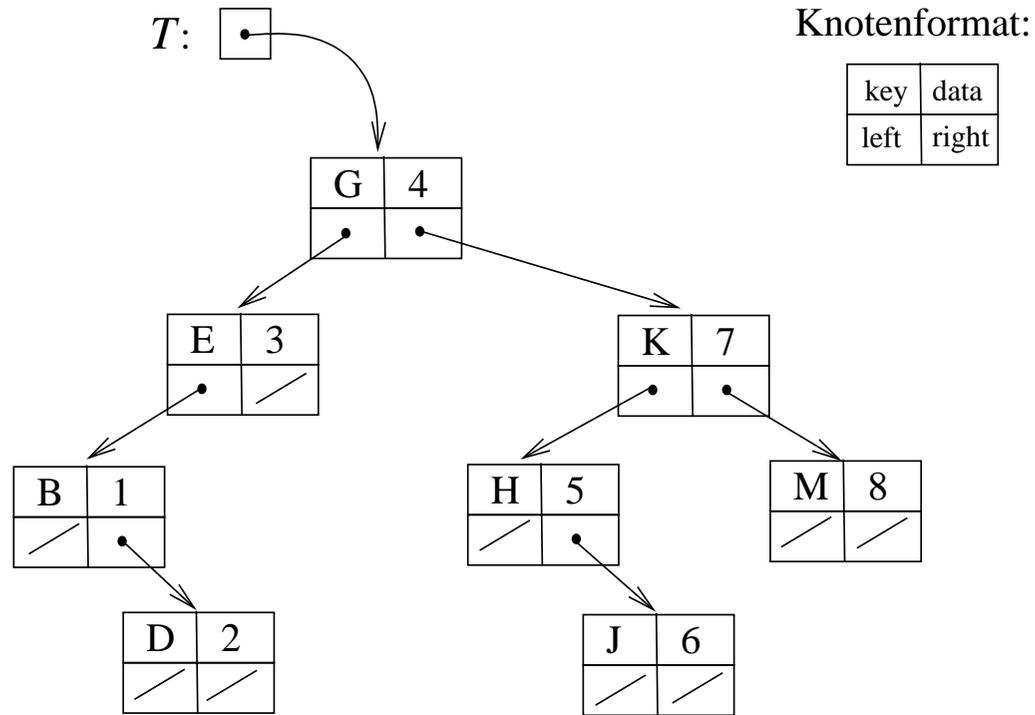


**Implementierung** von binären Suchbäumen: Verzeigerte Struktur.

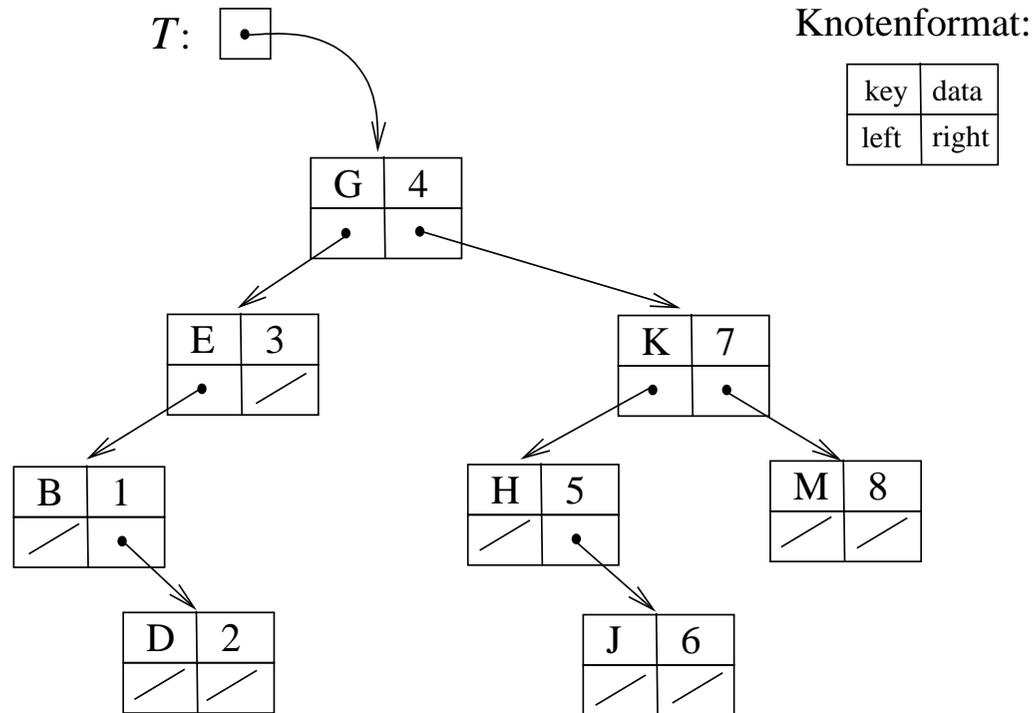


## Implementierung von binären Suchbäumen: Verzeigerte Struktur.

Knotenobjekt hat Komponenten (Attribute) für Schlüssel, Daten und Referenzen/Zeiger auf linkes und rechtes Kind:  $T.key$ ,  $T.data$ ,  $T.left(child)$ ,  $T.right(child)$

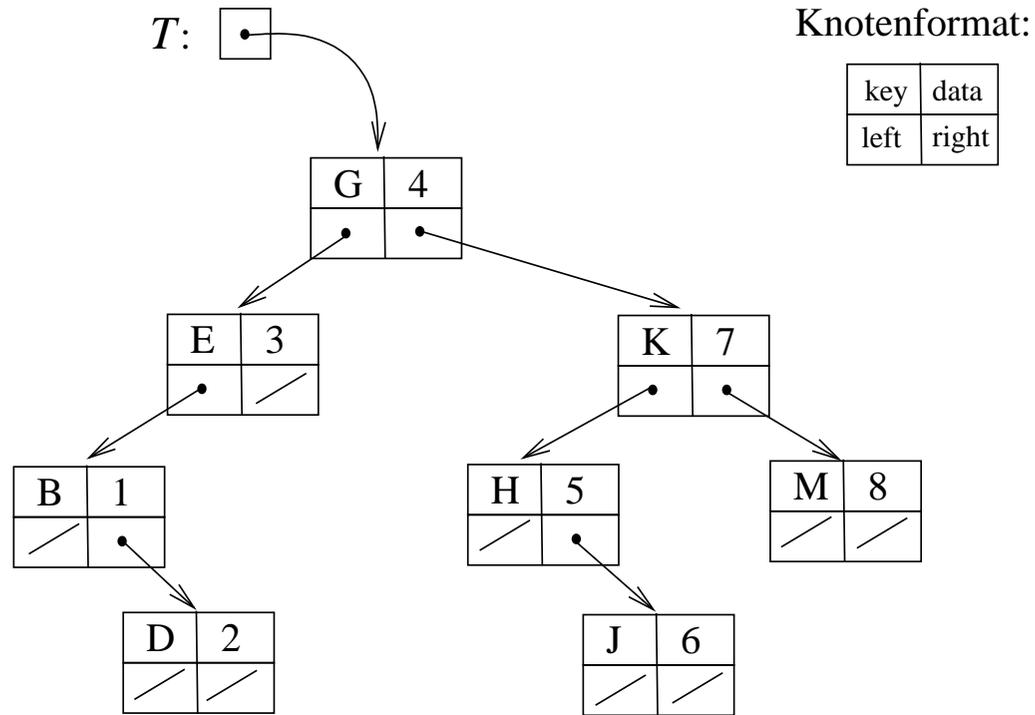


**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.



**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.

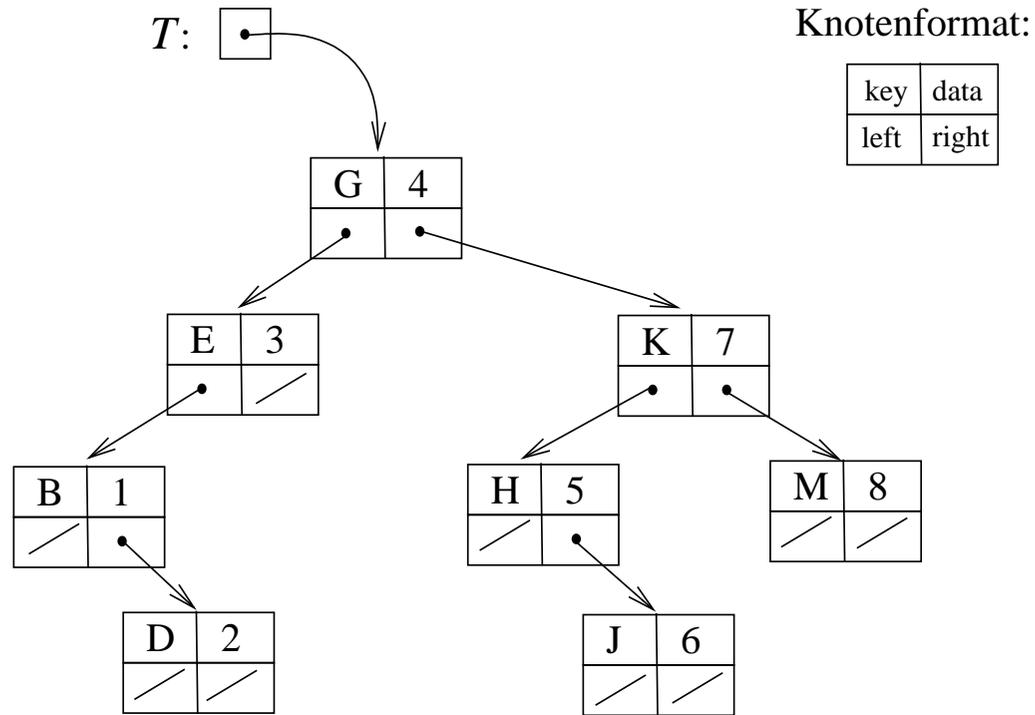
Nun: Implementierung der Wörterbuchoperationen (schon kurz in [\[AuP\]](#) betrachtet):



**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.

Nun: Implementierung der Wörterbuchoperationen (schon kurz in [\[AuP\]](#) betrachtet):

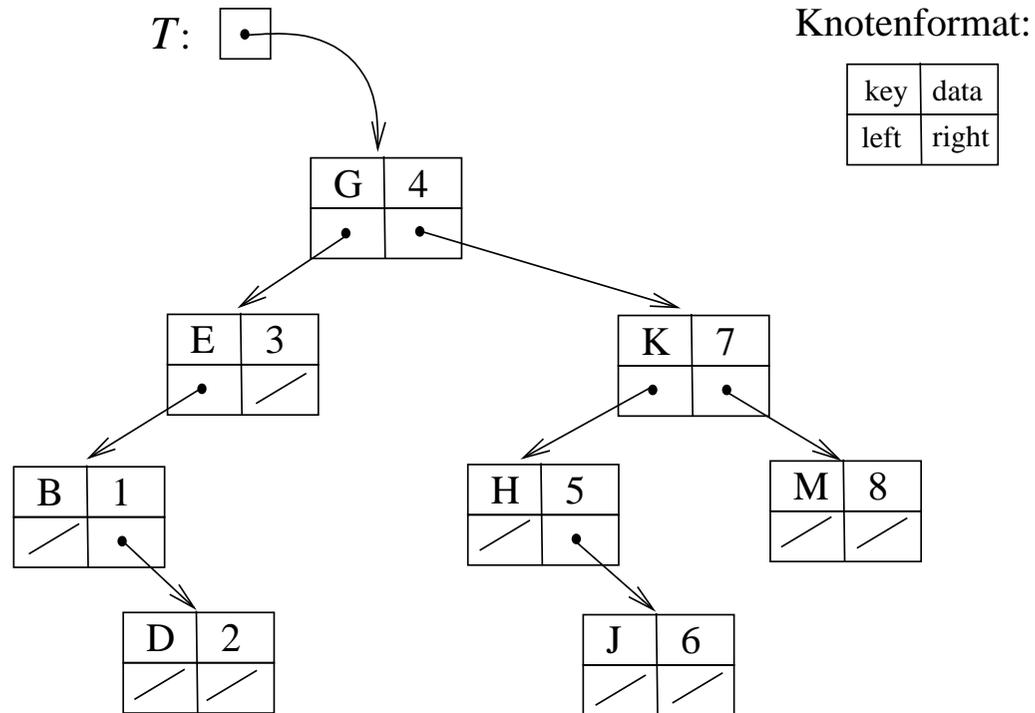
Erzeugen



**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.

Nun: Implementierung der Wörterbuchoperationen (schon kurz in [\[AuP\]](#) betrachtet):

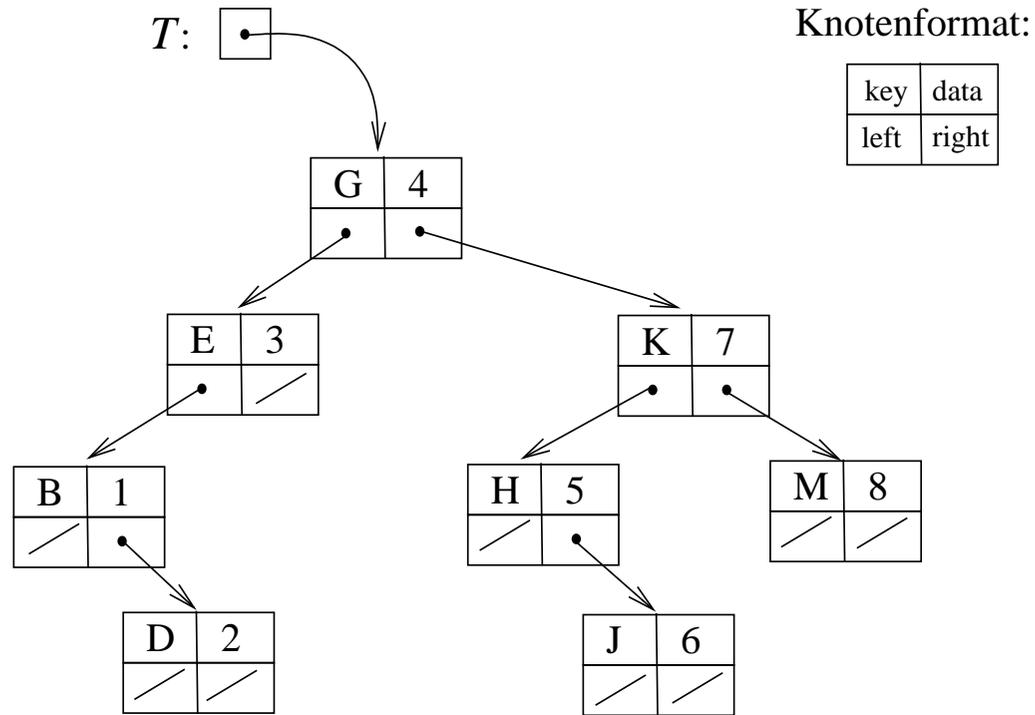
Erzeugen, Suchen



**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.

Nun: Implementierung der Wörterbuchoperationen (schon kurz in [\[AuP\]](#) betrachtet):

Erzeugen, Suchen, Einfügen



**Baum** ist gegeben durch Zeiger/Referenz auf Wurzel.

Nun: Implementierung der Wörterbuchoperationen (schon kurz in [\[AuP\]](#) betrachtet):

Erzeugen, Suchen, Einfügen, Löschen.

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T, x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T, x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ :

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ :

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ : **return**  $r$ .

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ : **return**  $r$ .

**3. Fall:**  $x < y$ :

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ : **return**  $r$ .

**3. Fall:**  $x < y$ : **return lookup**( $T.left, x$ ) // rekursiver Aufruf.

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ : **return**  $r$ .

**3. Fall:**  $x < y$ : **return lookup**( $T.left, x$ ) // rekursiver Aufruf.

**4. Fall:**  $x > y$ :

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x = y$ : **return**  $r$ .

**3. Fall:**  $x < y$ : **return lookup**( $T.left, x$ ) // rekursiver Aufruf.

**4. Fall:**  $x > y$ : **return lookup**( $T.right, x$ ) // rekursiver Aufruf.

---

## Suchen, rekursiv

**lookup**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis:  $r$ , wenn  $(x, r)$  im Baum  $T$  vorhanden

// „nicht vorhanden“, wenn es kein solches Paar gibt.

**1. Fall:**  $T = \square$ : **return** „nicht vorhanden“

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

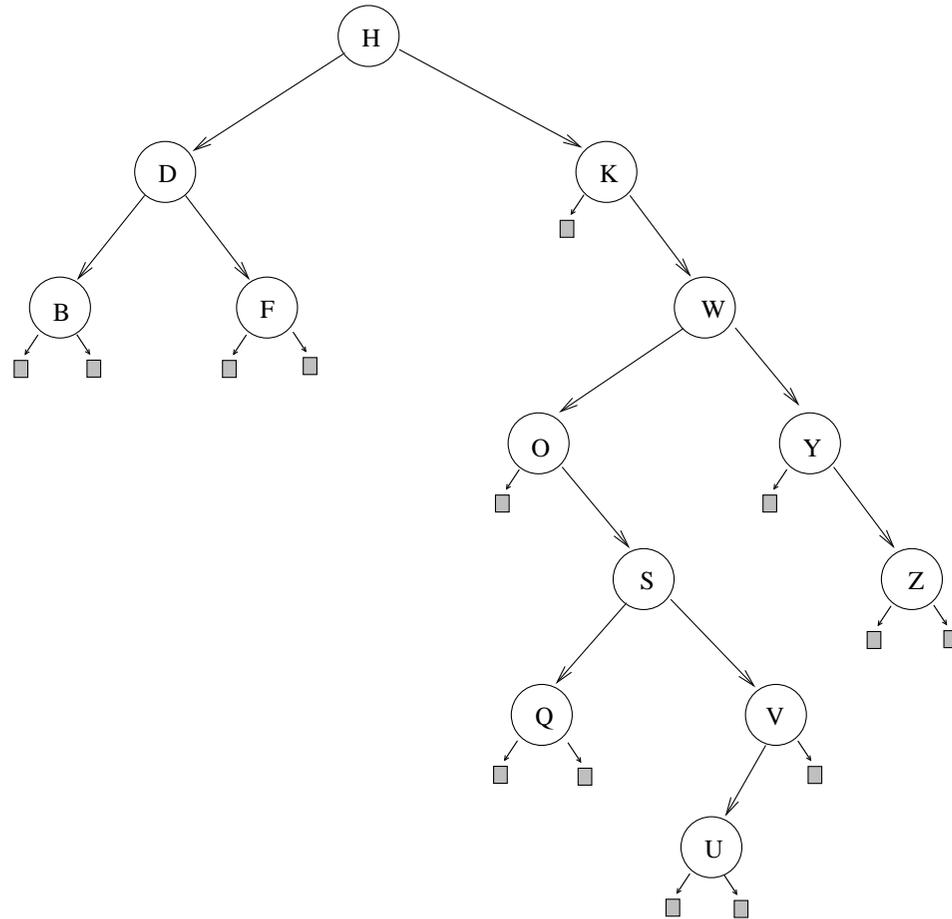
**2. Fall:**  $x = y$ : **return**  $r$ .

**3. Fall:**  $x < y$ : **return lookup**( $T.left, x$ ) // rekursiver Aufruf.

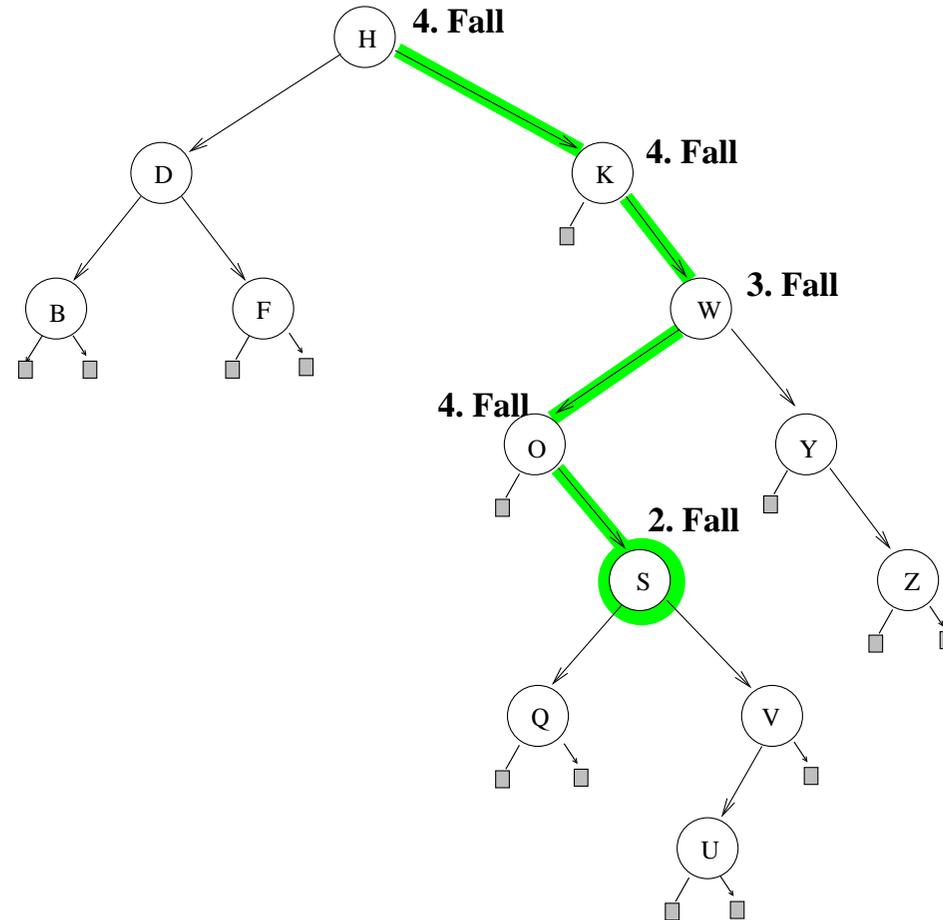
**4. Fall:**  $x > y$ : **return lookup**( $T.right, x$ ) // rekursiver Aufruf.

(**Beachte** die einfache Programmstruktur, die sich an rekursiver Definition von BSBen orientiert!)

*Beispiel:*

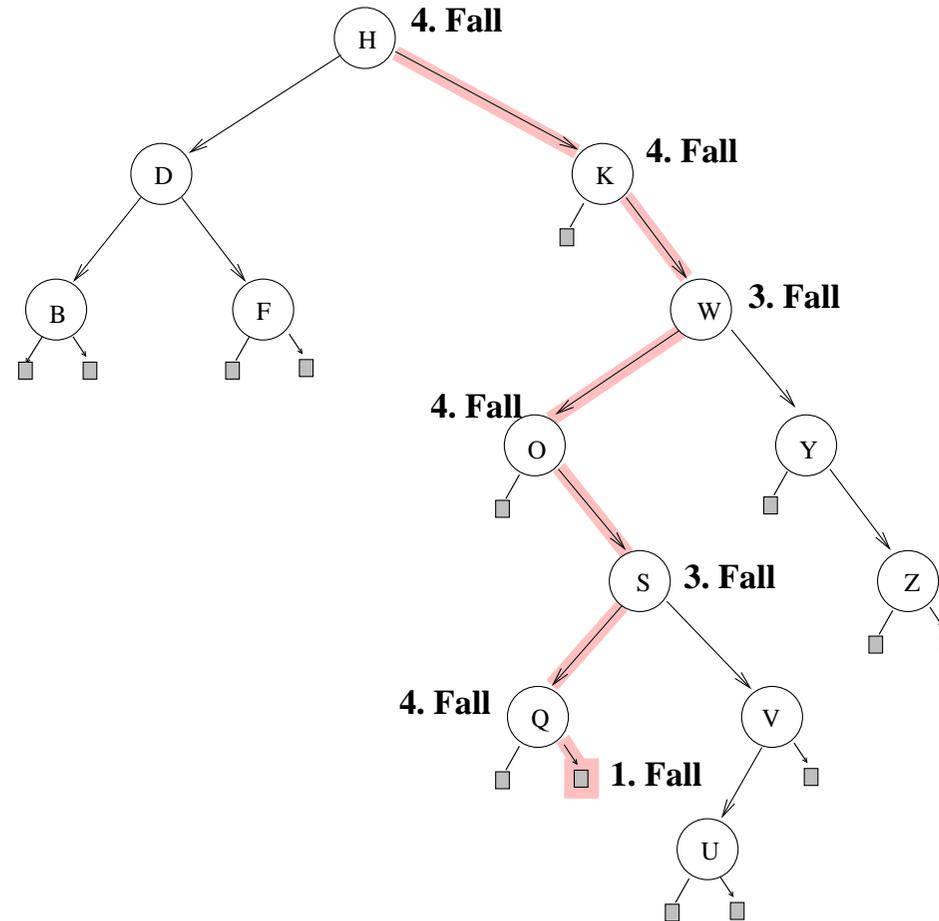


Beispiel:



Suche:  $S \in S$ .

Beispiel:



Suche:  $R \notin S$ .

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:**

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ .

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

Wenn  $x$  in  $T$  nicht vorkommt:

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

Wenn  $x$  in  $T$  nicht vorkommt: Suche endet in externem Knoten  $l_x$ .

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

Wenn  $x$  in  $T$  nicht vorkommt: Suche endet in externem Knoten  $l_x$ . Kosten  $O(1)$  für jeden Knoten auf dem Weg.

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

Wenn  $x$  in  $T$  nicht vorkommt: Suche endet in externem Knoten  $l_x$ . Kosten  $O(1)$  für jeden Knoten auf dem Weg. Also:

$$\text{Kosten} \quad O(d(l_x)) = O(d(T)).$$

---

**Korrektheit:** Für BSB  $T$  definieren wir  $f_T(x) := r$ , wenn es in  $T$  einen Knoten  $v$  mit  $v.key = x$  und  $v.data = r$  gibt; wenn es keinen Knoten  $v$  mit  $v.key = x$  gibt, ist  $x \notin \text{Def}(f_T)$ . D. h.:  $f_T$  ist die von  $T$  dargestellte Funktion. Per Induktion über den Aufbau von Binärbäumen zeigt man: Die Ausgabe von **lookup**( $T, x$ ) ist  $f_T(x)$ .

(Später gezeigt: **insert** und **delete** verändern  $f_T$  wie vom mathematischen Modell vorgeschrieben.)

**Zeitaufwand:** Wenn  $x$  in Knoten  $v_x$  sitzt:

Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  Kosten  $O(1)$ . Also:

$$\text{Kosten}^* \quad O(d(v_x)) = O(d(T)).$$

Wenn  $x$  in  $T$  nicht vorkommt: Suche endet in externem Knoten  $l_x$ . Kosten  $O(1)$  für jeden Knoten auf dem Weg. Also:

$$\text{Kosten} \quad O(d(l_x)) = O(d(T)).$$

\* Lies (überall)  $O(1 + d(v_x)) = O(1 + d(T))$  usw. – **Übung:** Suche **iterativ** programmieren.

---

**Einfügen, rekursiv**  
**insert**( $T, x, r$ ), für BSB  $T, x \in U, r \in R$ .

---

**Einfügen, rekursiv**  
**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ :

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

**3. Fall:**  $x < y$ :

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

**3. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{insert}(T.\text{left}, x, r)$ ; **return**  $T$ .

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

**3. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{insert}(T.\text{left}, x, r)$ ; **return**  $T$ .

**4. Fall:**  $y < x$ :

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

**3. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{insert}(T.\text{left}, x, r)$ ; **return**  $T$ .

**4. Fall:**  $y < x$ :  $T.\text{right} \leftarrow \text{insert}(T.\text{right}, x, r)$ ; **return**  $T$ .

---

## Einfügen, rekursiv

**insert**( $T, x, r$ ), für BSB  $T$ ,  $x \in U, r \in R$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{insert}(f_T, x, r)$ .

**1. Fall:**  $T = \square$ : Erzeuge neuen (Wurzel-)Knoten  $v$  mit Eintrag  $(x, r)$ ;  
 $T \leftarrow v$ ; **return**  $T$ .

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r'), T_2)$ .

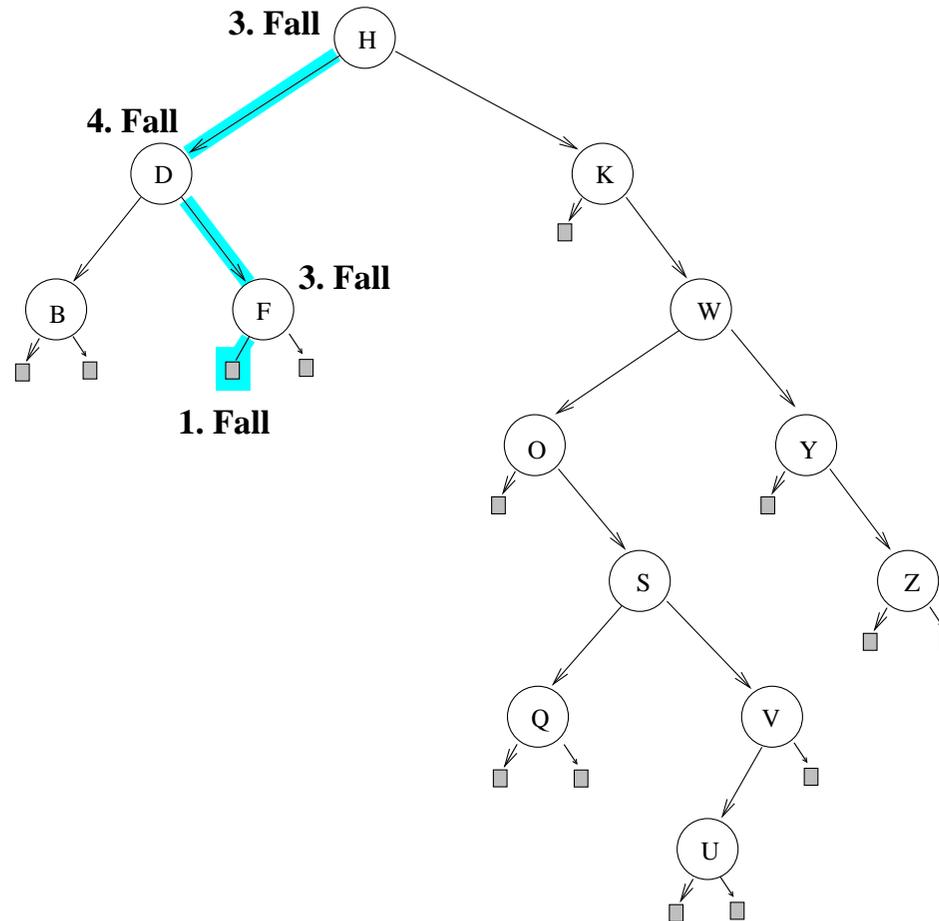
**2. Fall:**  $x = y$ :  $T.\text{data} \leftarrow r$ ; **return**  $T$ . // **Update-Situation!**

**3. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{insert}(T.\text{left}, x, r)$ ; **return**  $T$ .

**4. Fall:**  $y < x$ :  $T.\text{right} \leftarrow \text{insert}(T.\text{right}, x, r)$ ; **return**  $T$ .

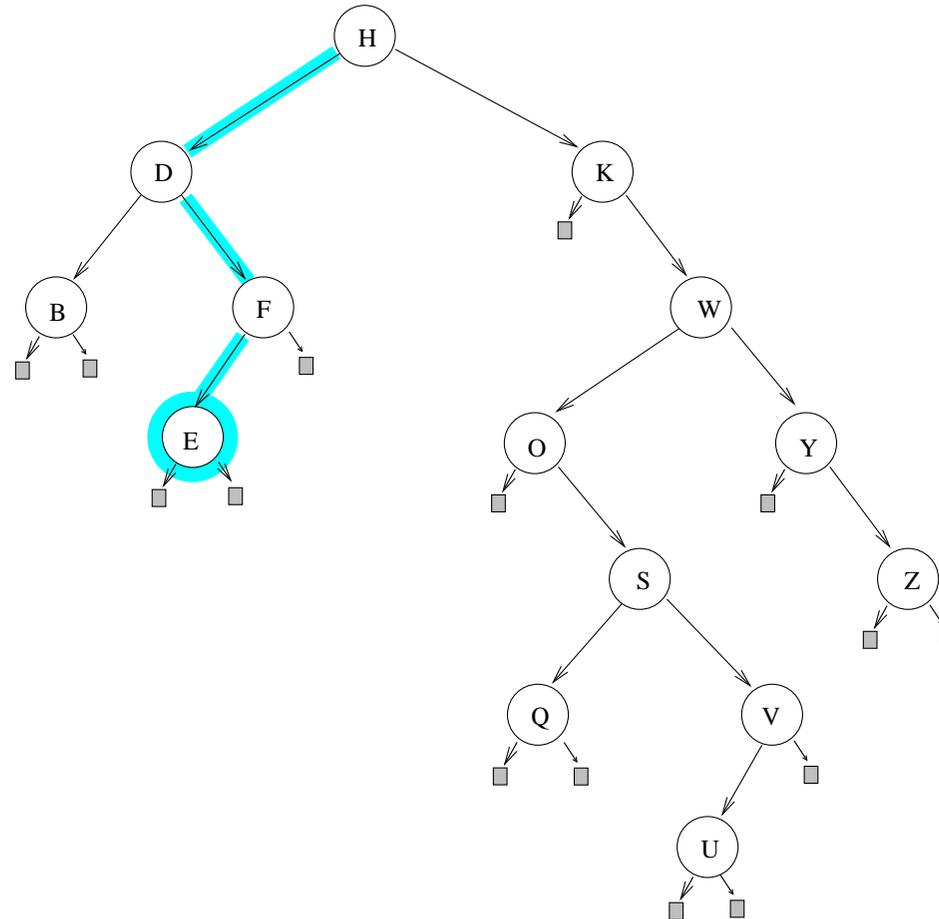
Beachte im 3. und im 4. Fall, dass implizit der Unterbaum „abgehängt“ und der Behandlung durch den rekursiven Aufruf übergeben wird. Durch den rekursiven Aufruf kann sich die Wurzel des Unterbaums verändern (oder aus einem null-Zeiger ein Zeiger auf einen echten Knoten werden). Der veränderte Unterbaum wird wieder an die richtige Stelle angehängt.

Beispiel:



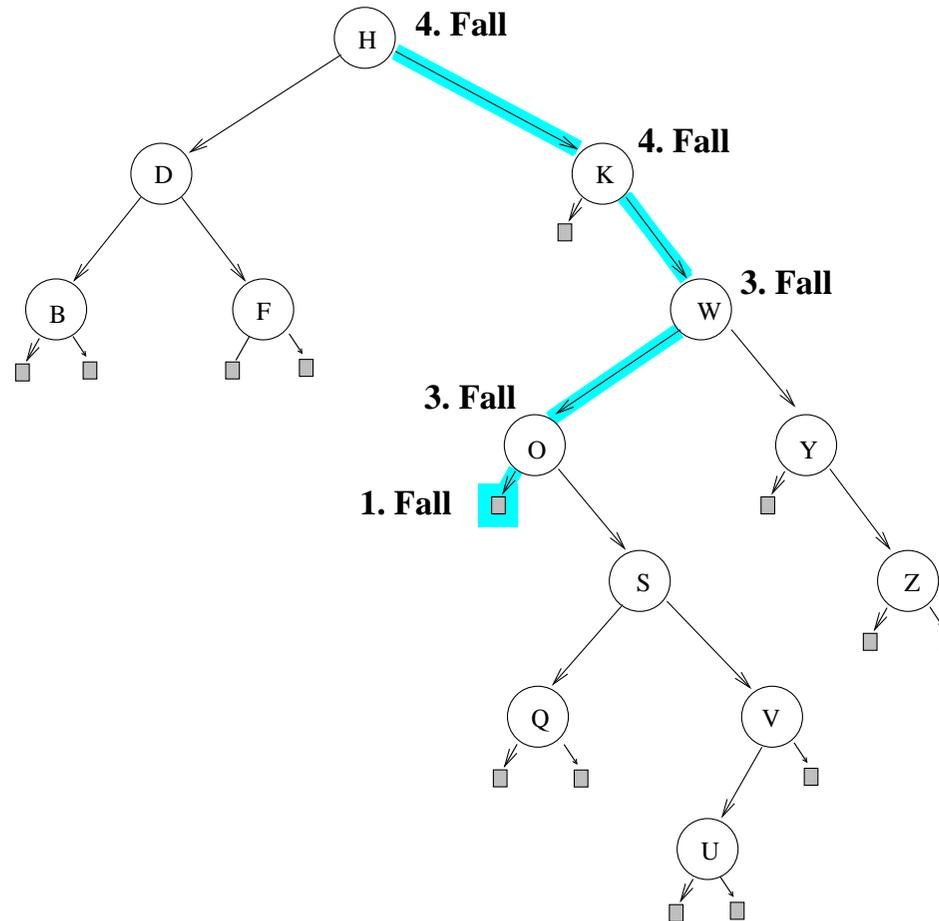
Füge ein:  $E \notin S$ .

Beispiel:



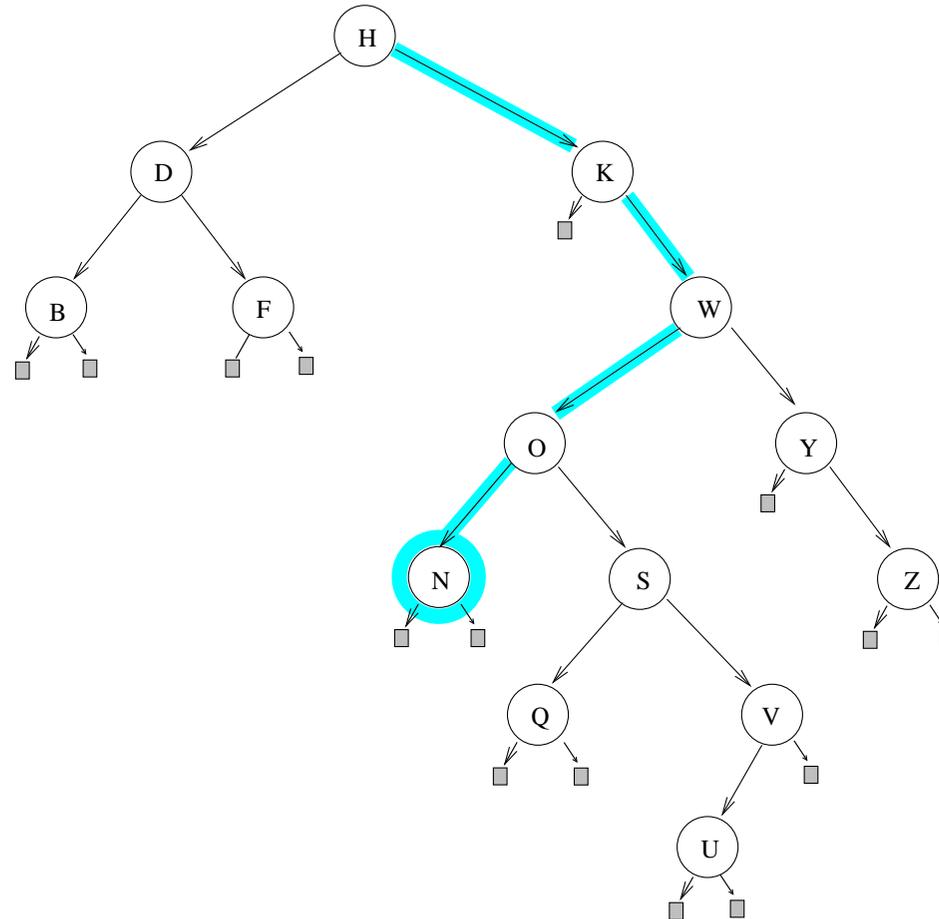
Füge ein:  $E \notin S$ .

Beispiel:



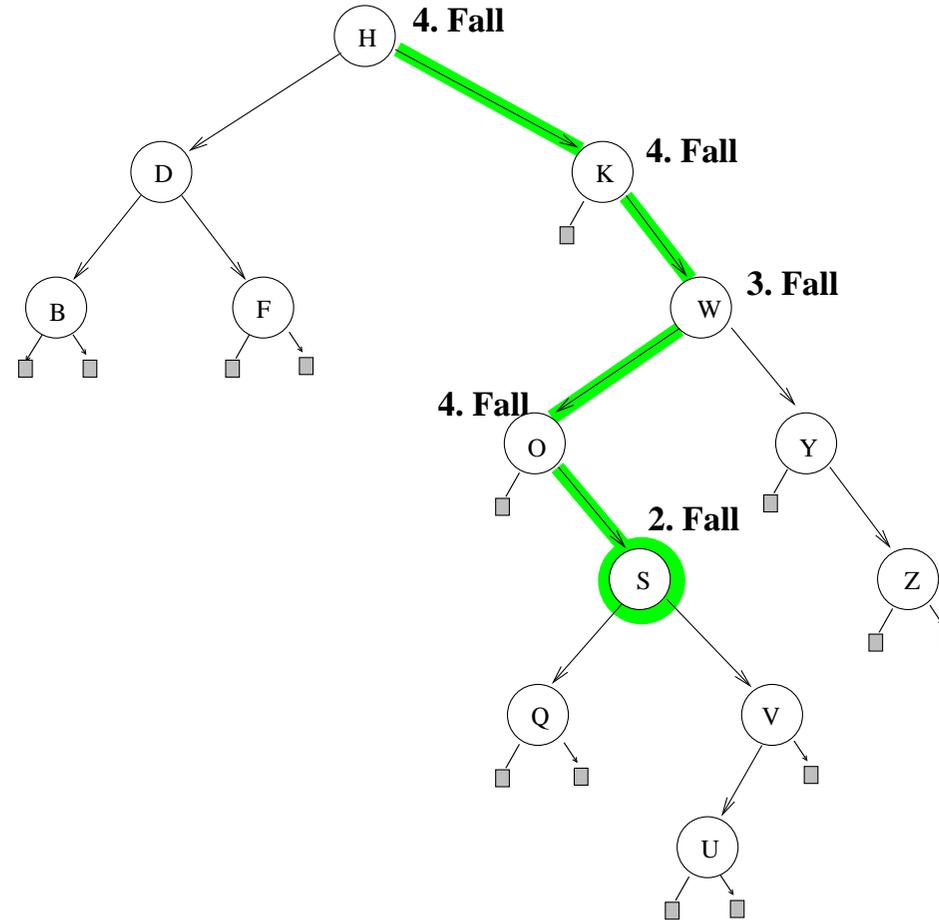
Füge ein:  $N \notin S$ .

Beispiel:



Füge ein:  $N \notin S$ .

Beispiel:



Füge ein:  $S \in S$ .

---

**Korrektheit:**

---

**Korrektheit:**

**Behauptung:** Aufruf **insert**( $T, x, r$ ) erzeugt einen binären Suchbaum  $T'$ ,

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

## Zeitaufwand:

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  schon vorhanden:

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  schon vorhanden:

$$\text{Kosten } O(d(v_x)) = O(d(T)).$$

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  schon vorhanden:

$$\text{Kosten } O(d(v_x)) = O(d(T)).$$

Wenn  $x$  nicht in  $T$ :  $O(1)$  für jeden Knoten auf dem Weg von Wurzel zu  $l_x$ .

---

## Korrektheit:

**Behauptung:** Aufruf  $\text{insert}(T, x, r)$  erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $\text{insert}(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  schon vorhanden:

$$\text{Kosten } O(d(v_x)) = O(d(T)).$$

Wenn  $x$  nicht in  $T$ :  $O(1)$  für jeden Knoten auf dem Weg von Wurzel zu  $l_x$ . Also:

$$\text{Kosten } O(d(l_x)) = O(d(T)).$$

---

## Korrektheit:

**Behauptung:** Aufruf **insert**( $T, x, r$ ) erzeugt einen binären Suchbaum  $T'$ , der das modifizierte Wörterbuch  $insert(f_T, x, r)$  darstellt. ( $f_T$ : die von  $T$  dargestellte Funktion.)

*Beweis:* Induktion über den Aufbau von Binärbäumen. (Siehe Druckfolien.)

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  schon vorhanden:

$$\text{Kosten } O(d(v_x)) = O(d(T)).$$

Wenn  $x$  nicht in  $T$ :  $O(1)$  für jeden Knoten auf dem Weg von Wurzel zu  $l_x$ . Also:

$$\text{Kosten } O(d(l_x)) = O(d(T)).$$

**Übung:** Einfügung **iterativ** programmieren. *Hinweis:* Man muss immer auch den Vorgänger des aktuellen Unterbaums kennen.

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T, x \in U$ .

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ :

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{delete}(T.\text{left}, x)$ ; **return**  $T$ ;

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{delete}(T.\text{left}, x)$ ; **return**  $T$ ;

**3. Fall:**  $y < x$ :

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T$ ,  $x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{delete}(T.\text{left}, x)$ ; **return**  $T$ ;

**3. Fall:**  $y < x$ :  $T.\text{right} \leftarrow \text{delete}(T.\text{right}, x)$ ; **return**  $T$ ;

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T, x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{delete}(T.\text{left}, x)$ ; **return**  $T$ ;

**3. Fall:**  $y < x$ :  $T.\text{right} \leftarrow \text{delete}(T.\text{right}, x)$ ; **return**  $T$ ;

**4. Fall:**  $x = y$ :

---

## Löschen, rekursiv

**delete**( $T, x$ ), für BSB  $T, x \in U$ .

// Ergebnis: Baum  $T'$  mit  $f_{T'} = \text{delete}(f_T, x)$ .

**1. Fall:**  $T = \square$ : **return**  $T$ . // Nichts zu tun ( $x$  nicht vorhanden)

// Ab hier:  $T \neq \square$ , also  $T = (T_1, (y, r), T_2)$ .

**2. Fall:**  $x < y$ :  $T.\text{left} \leftarrow \text{delete}(T.\text{left}, x)$ ; **return**  $T$ ;

**3. Fall:**  $y < x$ :  $T.\text{right} \leftarrow \text{delete}(T.\text{right}, x)$ ; **return**  $T$ ;

**4. Fall:**  $x = y$ : **Lösche Wurzel** von  $T$  (Unterfälle **4a–4c**, s. u.).

---

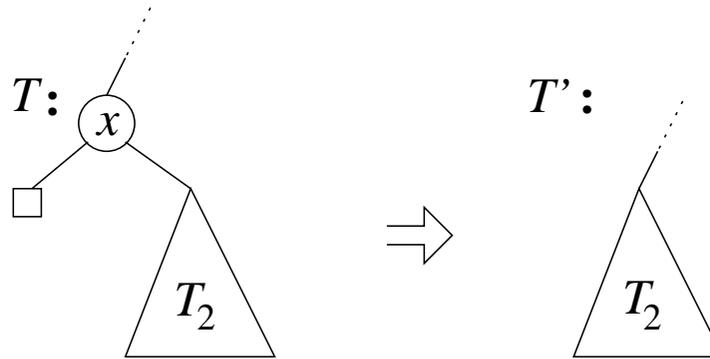
**Lösche Wurzel** von  $T = (T_1, (x, r), T_2)$ !

**Fall 4a:**  $T.\text{left} = \square$ :

---

**Lösche Wurzel** von  $T = (T_1, (x, r), T_2)$ !

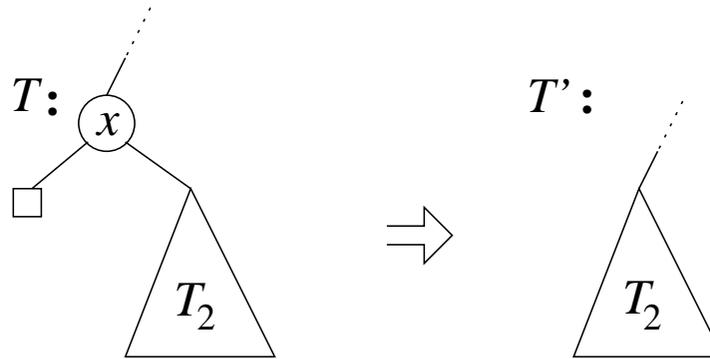
**Fall 4a:**  $T.left = \square$ : **return**  $T.right$  (auch wenn  $T.right = \square$  ist!)



---

**Lösche Wurzel** von  $T = (T_1, (x, r), T_2)$ !

**Fall 4a:**  $T.left = \square$ : **return**  $T.right$  (auch wenn  $T.right = \square$  ist!)

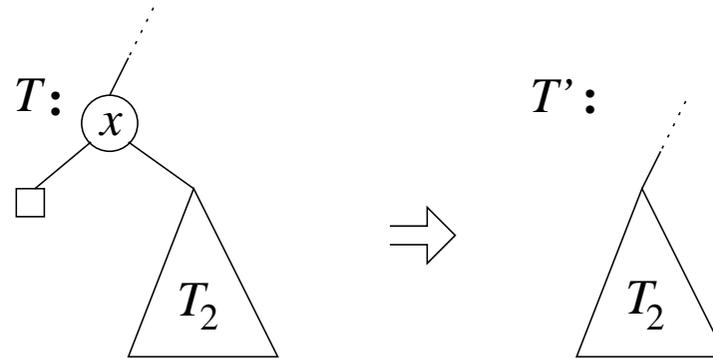


**Fall 4b:**  $T.right = \square$ :

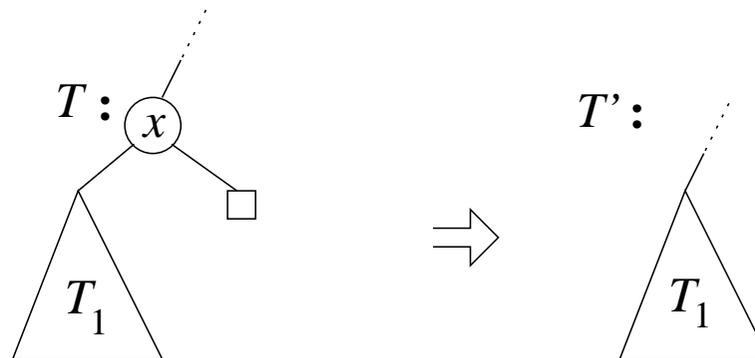
---

**Lösche Wurzel** von  $T = (T_1, (x, r), T_2)$ !

**Fall 4a:**  $T.left = \square$ : **return**  $T.right$  (auch wenn  $T.right = \square$  ist!)



**Fall 4b:**  $T.right = \square$ : **return**  $T.left$

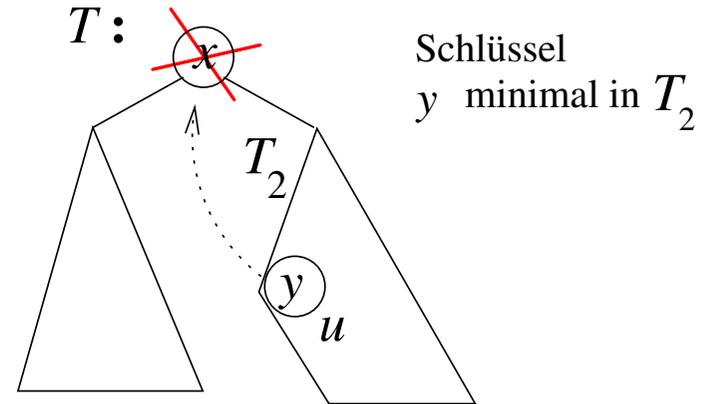


---

**Fall 4c:**  $T.left \neq \square$  und  $T.right \neq \square$ .

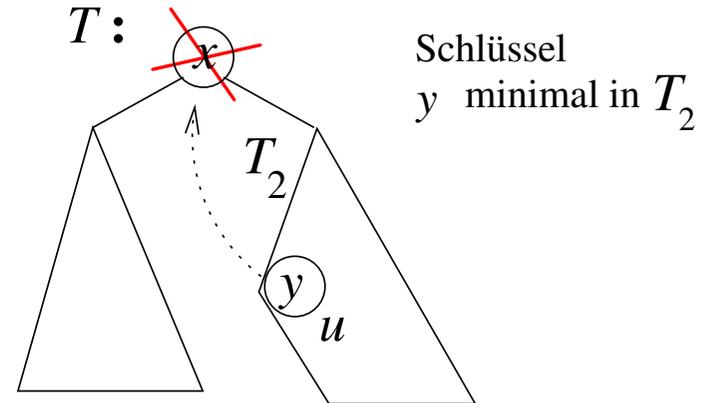
---

Fall 4c:  $T.left \neq \square$  und  $T.right \neq \square$ .



---

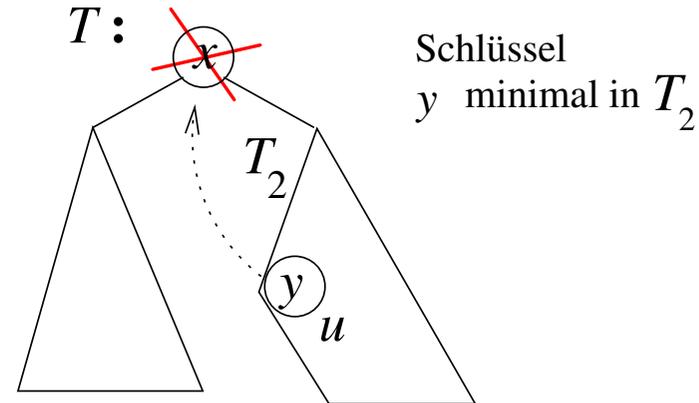
Fall 4c:  $T.left \neq \square$  und  $T.right \neq \square$ .



Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .

---

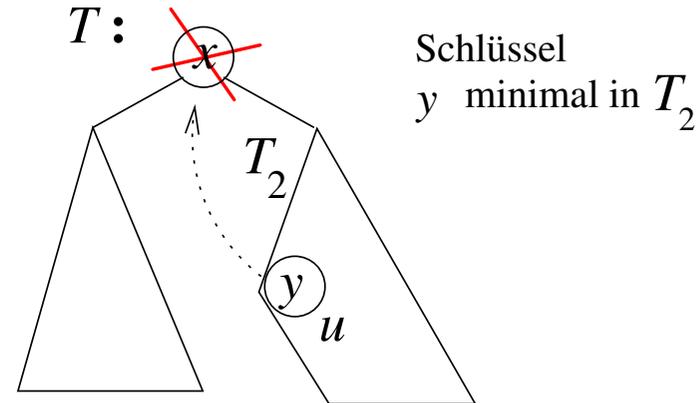
**Fall 4c:**  $T.left \neq \square$  und  $T.right \neq \square$ .



Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

---

Fall 4c:  $T.left \neq \square$  und  $T.right \neq \square$ .

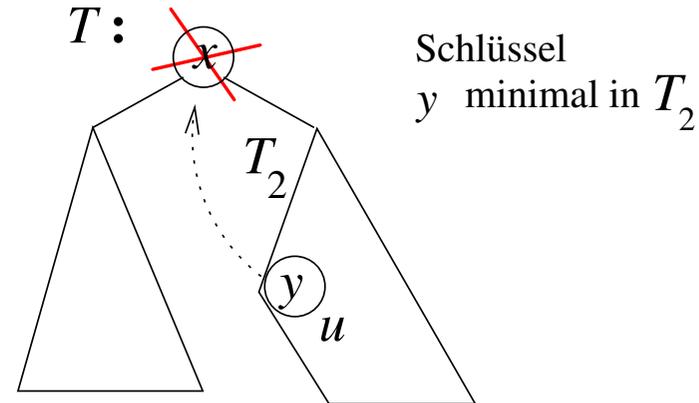


Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

$T'_2 := T_2$  nach Entfernen von Knoten  $u$ ; // siehe unten **ExtractMin**

---

Fall 4c:  $T.left \neq \square$  und  $T.right \neq \square$ .



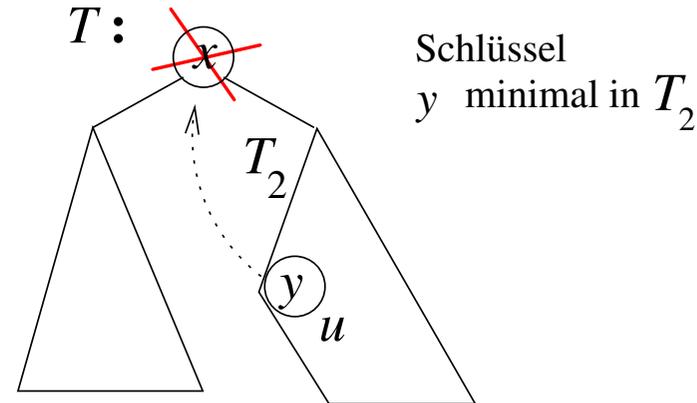
Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

$T'_2 := T_2$  nach Entfernen von Knoten  $u$ ; // siehe unten **ExtractMin**

$u.left \leftarrow T.left$ ;

---

**Fall 4c:**  $T.left \neq \square$  und  $T.right \neq \square$ .

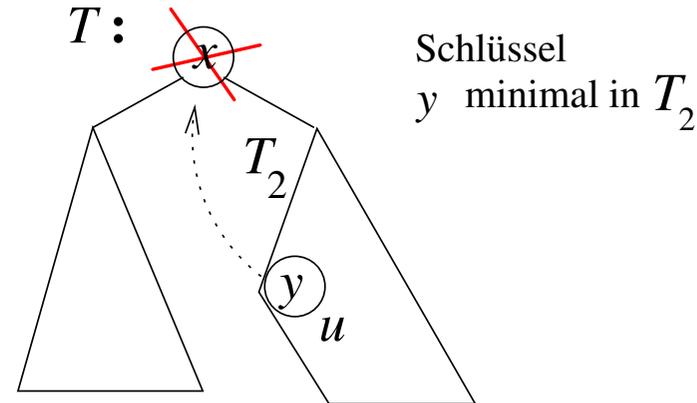


Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

$T'_2 := T_2$  nach Entfernen von Knoten  $u$ ; // siehe unten **ExtractMin**  
 $u.left \leftarrow T.left$ ;  $u.right \leftarrow T'_2$ ;

---

**Fall 4c:**  $T.left \neq \square$  und  $T.right \neq \square$ .

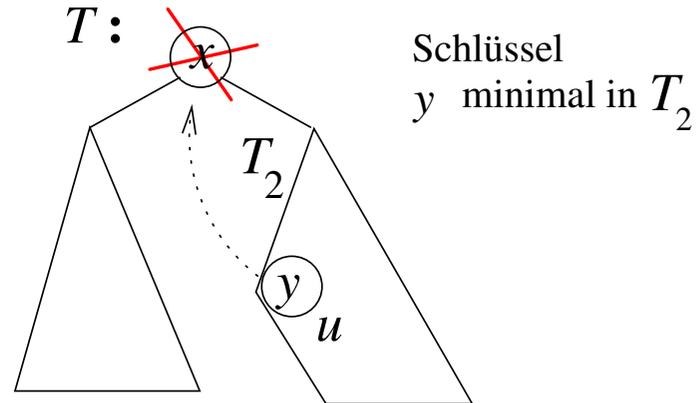


Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

$T'_2 := T_2$  nach Entfernen von Knoten  $u$ ; // siehe unten **ExtractMin**  
 $u.left \leftarrow T.left$ ;  $u.right \leftarrow T'_2$ ; **return**  $u$ .

---

**Fall 4c:**  $T.left \neq \square$  und  $T.right \neq \square$ .



Suche in  $T_2$  Eintrag  $(y, r')$  mit minimalem Schlüssel  $y$ , in Knoten  $u$ .  
(Dies ist der **Inorder-Nachfolger** der Wurzel von  $T$ .)

$T'_2 := T_2$  nach Entfernen von Knoten  $u$ ; // siehe unten **ExtractMin**  
 $u.left \leftarrow T.left$ ;  $u.right \leftarrow T'_2$ ; **return**  $u$ .

(Zeiger/Referenzen umhängen, nicht Daten/Schlüssel kopieren!)

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$   
den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$   
den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$   
den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$   
den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

**Fall „leer“:**  $T.left = \square$ .

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$  den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

**Fall „leer“:**  $T.left = \square$ . // Schlüssel  $z$  in der Wurzel ist minimal in  $T$ .

$T' \leftarrow T.right$ ;  $T.right \leftarrow \square$ ;  $u \leftarrow T$ ; **return**( $T', u$ ) // Knoten  $u$  war Wurzel.

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$  den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

**Fall „leer“:**  $T.left = \square$ . // Schlüssel  $z$  in der Wurzel ist minimal in  $T$ .

$T' \leftarrow T.right$ ;  $T.right \leftarrow \square$ ;  $u \leftarrow T$ ; **return**( $T', u$ ) // Knoten  $u$  war Wurzel.

**Fall „nichtleer“:**  $T.left \neq \square$ .

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$  den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

**Fall „leer“:**  $T.left = \square$ . // Schlüssel  $z$  in der Wurzel ist minimal in  $T$ .

$T' \leftarrow T.right$ ;  $T.right \leftarrow \square$ ;  $u \leftarrow T$ ; **return**( $T', u$ ) // Knoten  $u$  war Wurzel.

**Fall „nichtleer“:**  $T.left \neq \square$ .

$(T', u) \leftarrow \mathbf{extractMin}(T.left)$ ;  $T.left \leftarrow T'$ ; **return**( $T, u$ ).

---

## Extrahieren des Minimums, rekursiv

**extractMin**( $T$ ) // zieht aus einem (nichtleeren) BSB  $T = (T_1, (z, r), T_2)$  den Knoten  $u$  mit dem kleinsten Schlüssel heraus.

Dies ist ebenfalls rekursiv zu realisieren.

Resultat: Kleinerer Baum  $T'$  und ein separater Baumknoten  $u$  (via Zeiger auf  $u$ ).

**Fall „leer“:**  $T.left = \square$ . // Schlüssel  $z$  in der Wurzel ist minimal in  $T$ .

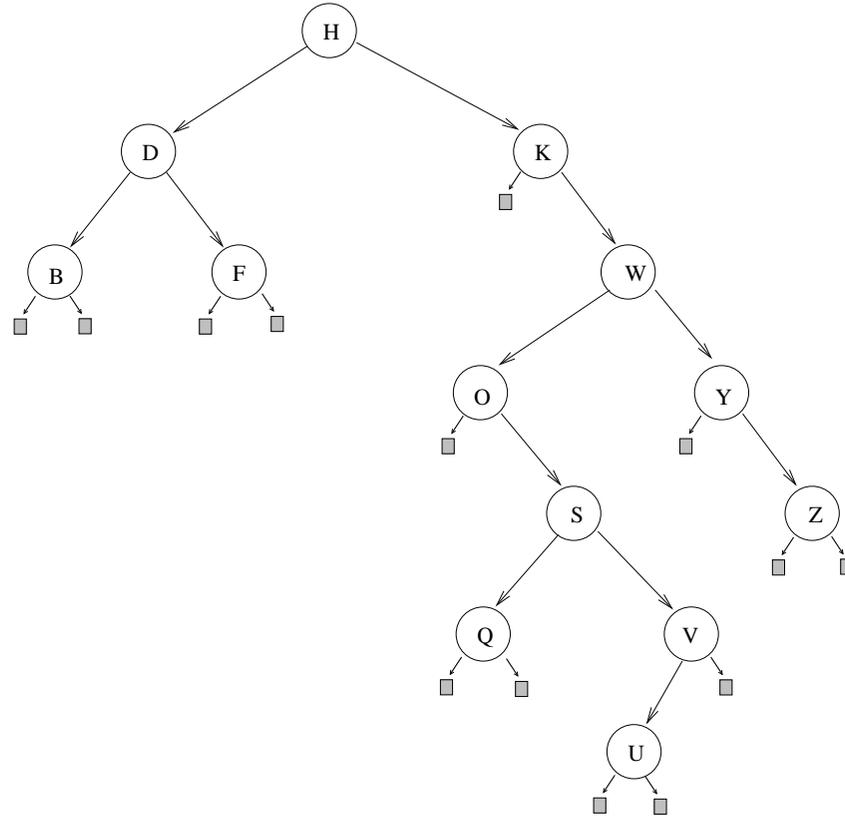
$T' \leftarrow T.right$ ;  $T.right \leftarrow \square$ ;  $u \leftarrow T$ ; **return**( $T', u$ ) // Knoten  $u$  war Wurzel.

**Fall „nichtleer“:**  $T.left \neq \square$ .

$(T', u) \leftarrow \mathbf{extractMin}(T.left)$ ;  $T.left \leftarrow T'$ ; **return**( $T, u$ ).

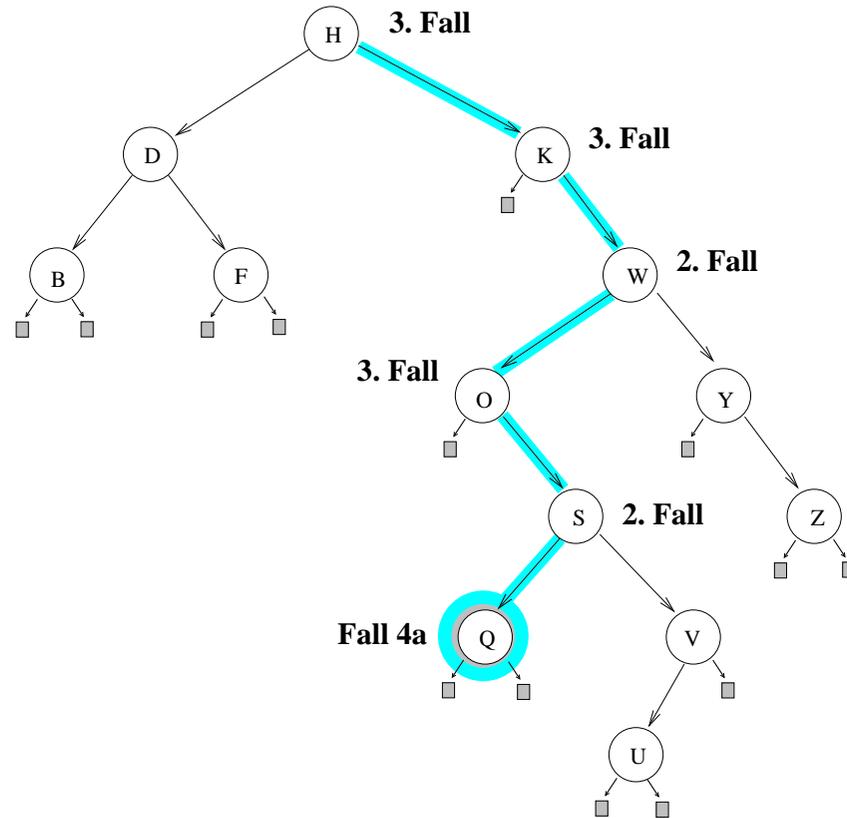
Anschaulich, iterativ: Starte in der Wurzel von  $T$ . Laufe den „Weg der linken Kinder“ hinab, bis ein Knoten  $u$  erreicht ist, der kein linkes Kind hat. Knoten  $u$  enthält den kleinsten Schlüssel in  $T$ . Lenke Zeiger auf  $u$  auf den rechten Unterbaum von  $u$  um, und gib den so modifizierten Baum und  $u$  zurück.

*Beispiel:*



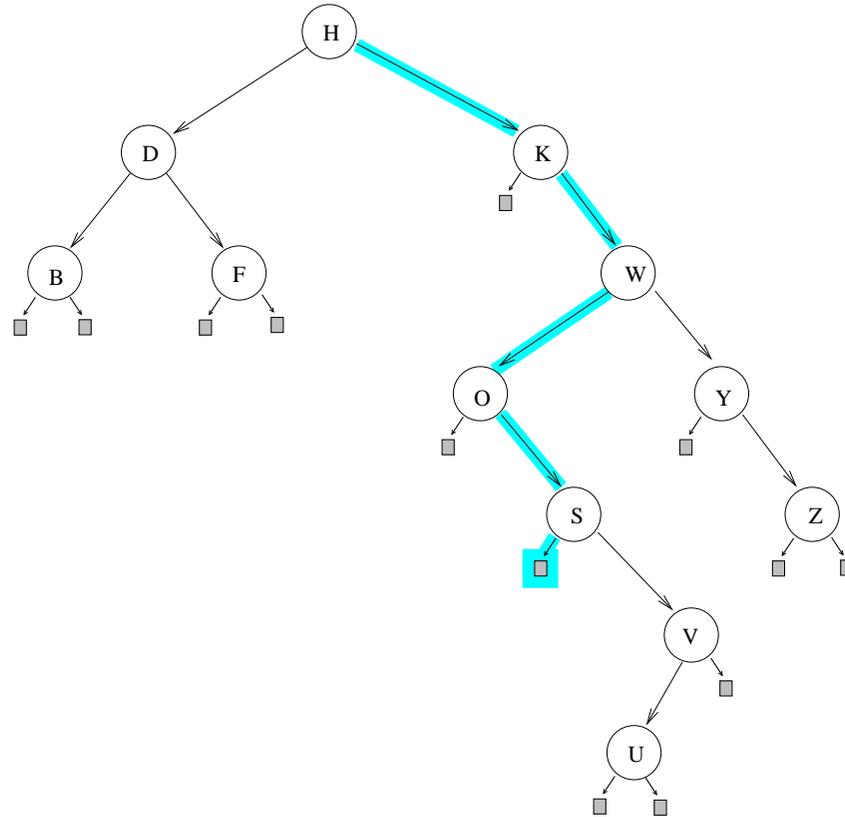
Lösche: Q.

Beispiel:



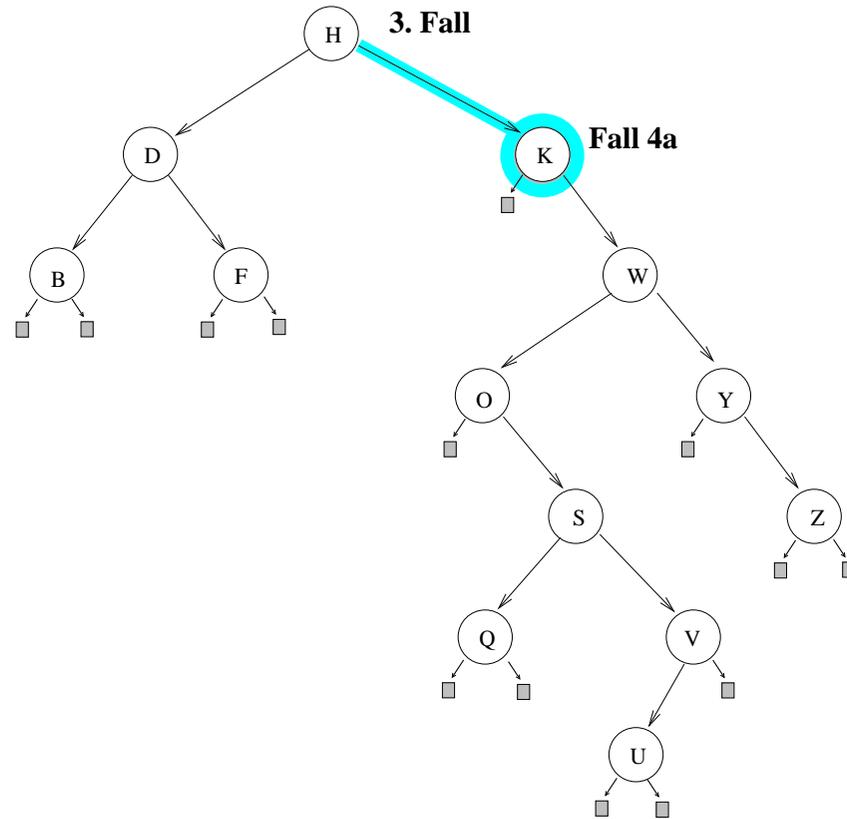
Lösche: Q.

Beispiel:



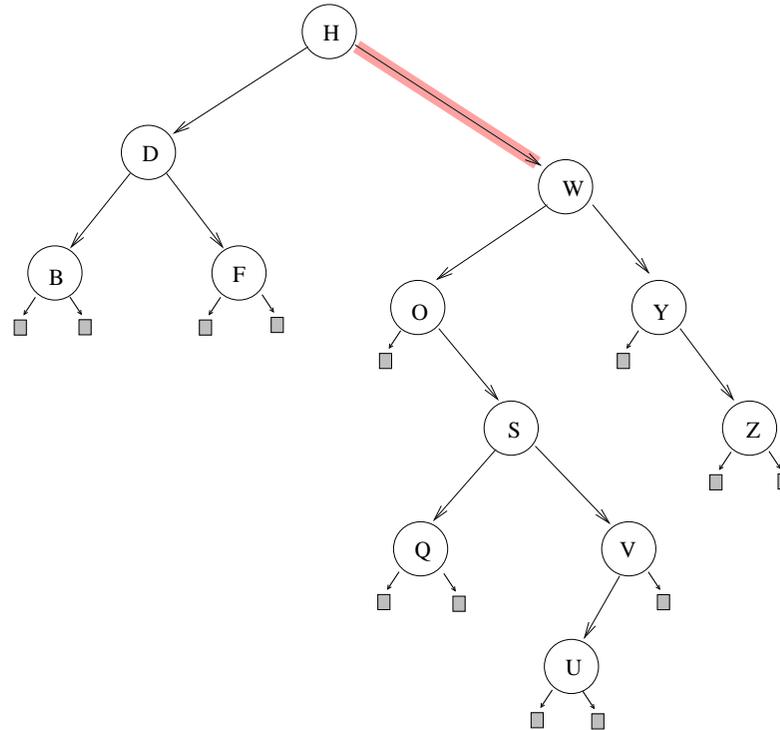
Lösche: Q.

Beispiel:



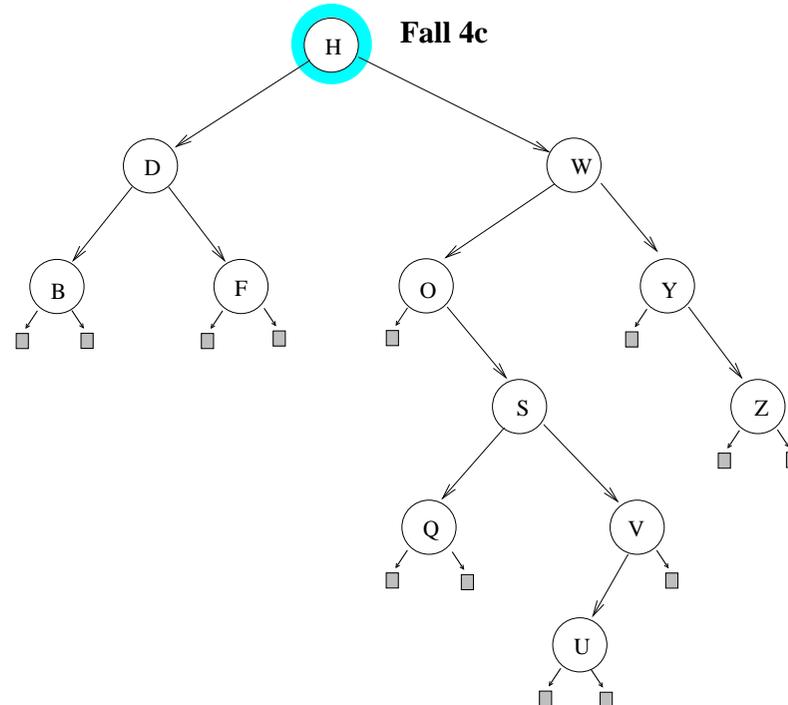
Lösche: K.

Beispiel:



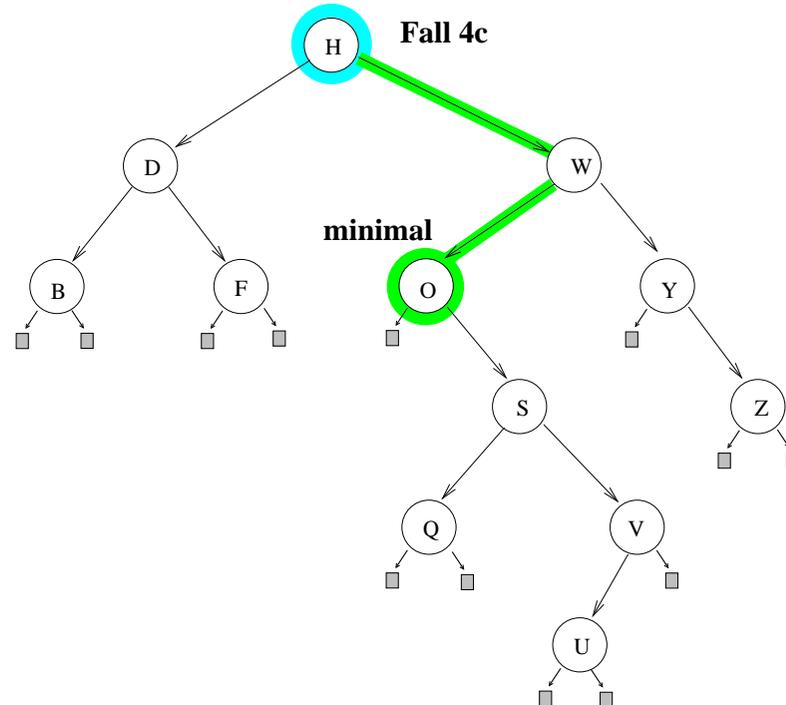
Lösche: K.

Beispiel:



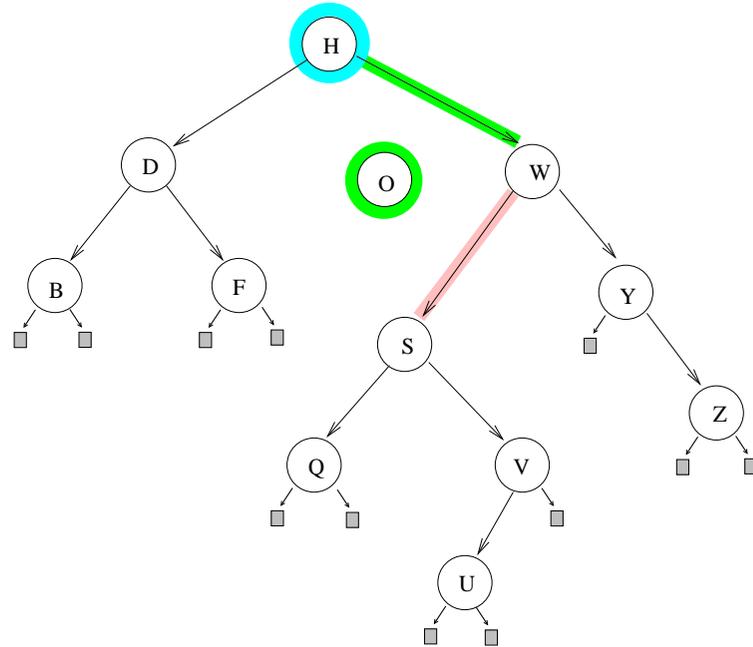
Lösche: H.

Beispiel:



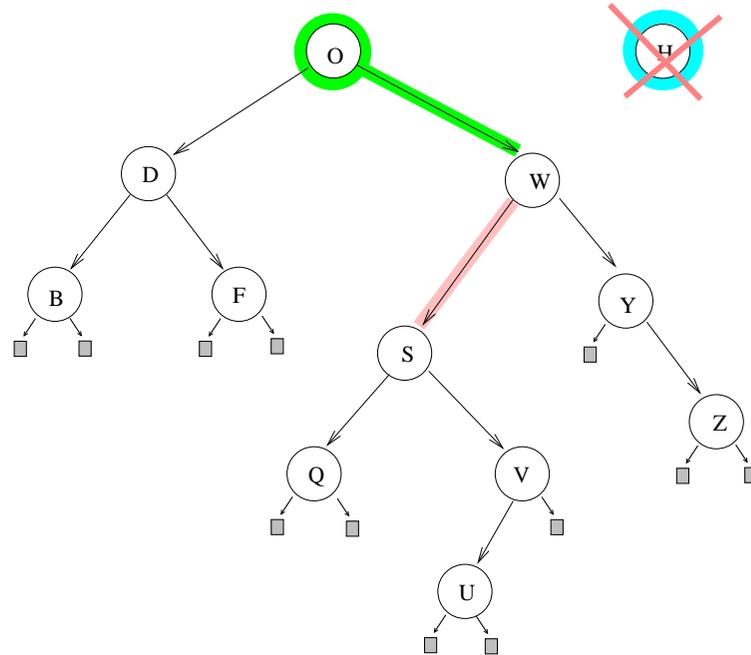
Lösche: H.

Beispiel:



Lösche: H.

Beispiel:



Lösche: H.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:  
Suche Schlüssel  $x$  in  $T$ .

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:  
Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ .

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ . (Gehe zum rechten Kind von  $v$  und dann „immer links“, bis Knoten  $u$  erreicht, der kein linkes Kind hat.)

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ . (Gehe zum rechten Kind von  $v$  und dann „immer links“, bis Knoten  $u$  erreicht, der kein linkes Kind hat.) **Der linke Unterbaum von  $u$  ist immer leer!**

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ . (Gehe zum rechten Kind von  $v$  und dann „immer links“, bis Knoten  $u$  erreicht, der kein linkes Kind hat.) **Der linke Unterbaum von  $u$  ist immer leer!**

Lenke den Zeiger vom Vorgänger von  $u$  auf den rechten Unterbaum von  $u$  um, behalte  $u$  als einzelnen Knoten.

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ . (Gehe zum rechten Kind von  $v$  und dann „immer links“, bis Knoten  $u$  erreicht, der kein linkes Kind hat.) **Der linke Unterbaum von  $u$  ist immer leer!**

Lenke den Zeiger vom Vorgänger von  $u$  auf den rechten Unterbaum von  $u$  um, behalte  $u$  als einzelnen Knoten.

Ersetze in  $T$  den Knoten  $v$  durch  $u$  (Zeiger zu den Kindern kopieren, Zeiger auf  $v$  nach  $u$  umlenken).

---

Intuitive, **iterative** Beschreibung des Löschvorgangs:

Suche Schlüssel  $x$  in  $T$ . Falls nicht gefunden, ist nichts zu tun. Falls  $x$  in Knoten  $v$ , drei Fälle:

a)  $v$  hat leeren linken Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den rechten Unterbaum von  $v$  um.“

(**Achtung!** Der Fall, dass  $v$  Blatt ist, ist hier inbegriffen.)

b)  $v$  hat leeren rechten Unterbaum.

„Biege Zeiger vom Vorgänger von  $v$  zu  $v$  auf den linken Unterbaum von  $v$  um.“

c) Beide Unterbäume von  $v$  sind nicht leer.

Suche **Inorder-Nachfolger**  $u$  von  $v$ , d. h. den Knoten mit kleinstem Schlüssel im rechten Unterbaum von  $v$ . (Gehe zum rechten Kind von  $v$  und dann „immer links“, bis Knoten  $u$  erreicht, der kein linkes Kind hat.) **Der linke Unterbaum von  $u$  ist immer leer!**

Lenke den Zeiger vom Vorgänger von  $u$  auf den rechten Unterbaum von  $u$  um, behalte  $u$  als einzelnen Knoten.

Ersetze in  $T$  den Knoten  $v$  durch  $u$  (Zeiger zu den Kindern kopieren, Zeiger auf  $v$  nach  $u$  umlenken).

### Übung:

Löschung **iterativ** programmieren. *Hinweis:* Erst  $v$  mit  $v.key = x$  finden, aber man benötigt auch den Vorgänger von  $v$ . Im Fall c):  $u$  finden, aber man benötigt auch den Vorgänger von  $u$ .

---

## Korrektheit:

---

**Korrektheit:**

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

- $x$  im Knoten  $v_x$ , Fälle 2/3:  $O(d(v_x)) = O(d(T))$ .

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

- $x$  im Knoten  $v_x$ , Fälle 2/3:  $O(d(v_x)) = O(d(T))$ .
- $x$  im Knoten  $v_x$ , Fall 4:

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

- $x$  im Knoten  $v_x$ , Fälle 2/3:  $O(d(v_x)) = O(d(T))$ .
- $x$  im Knoten  $v_x$ , Fall 4:  
 $O(d(v_x^+)) = O(d(T))$ , für Inorder-Nachfolger  $v_x^+$  von  $v_x$ .

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

- $x$  im Knoten  $v_x$ , Fälle 2/3:  $O(d(v_x)) = O(d(T))$ .
- $x$  im Knoten  $v_x$ , Fall 4:  
 $O(d(v_x^+)) = O(d(T))$ , für Inorder-Nachfolger  $v_x^+$  von  $v_x$ .
- $x$  in  $T$  nicht vorhanden:

---

## Korrektheit:

**Behauptung:** Aufruf **delete**( $T, x$ ) erzeugt einen binären Suchbaum für das modifizierte Wörterbuch  $delete(f_T, x)$ .

*Beweis:* Induktion über den Aufbau von Binärbäumen.

Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**( $T$ ), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

## Zeitaufwand:

- $x$  im Knoten  $v_x$ , Fälle 2/3:  $O(d(v_x)) = O(d(T))$ .
- $x$  im Knoten  $v_x$ , Fall 4:  
 $O(d(v_x^+)) = O(d(T))$ , für Inorder-Nachfolger  $v_x^+$  von  $v_x$ .
- $x$  in  $T$  nicht vorhanden:  
 $O(d(l_x)) = O(d(T))$ , für externen Knoten  $l_x$ .

**Vorlesungsvideo:**

**Erwartete mittlere Tiefe  
in zufälligen BSBs**

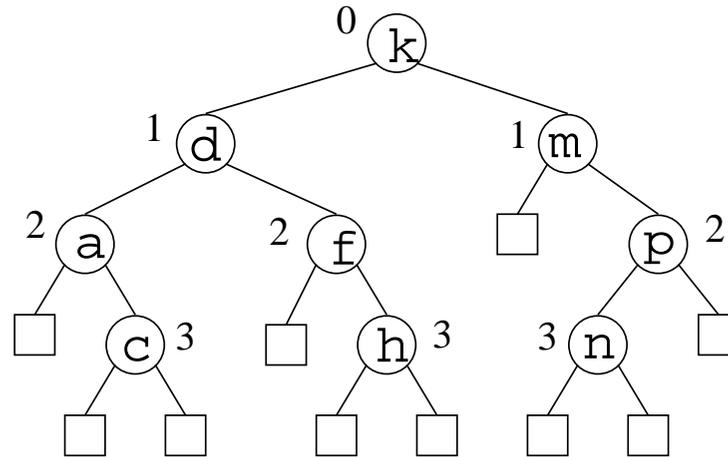
---

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

---

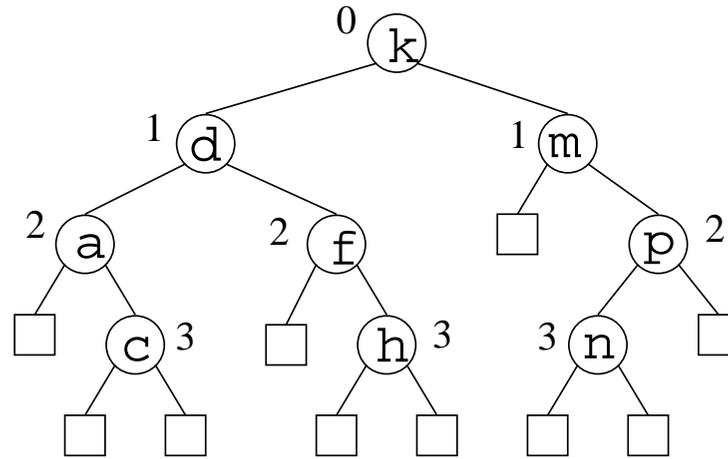
## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:  
Summe der Knotentiefen.



## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

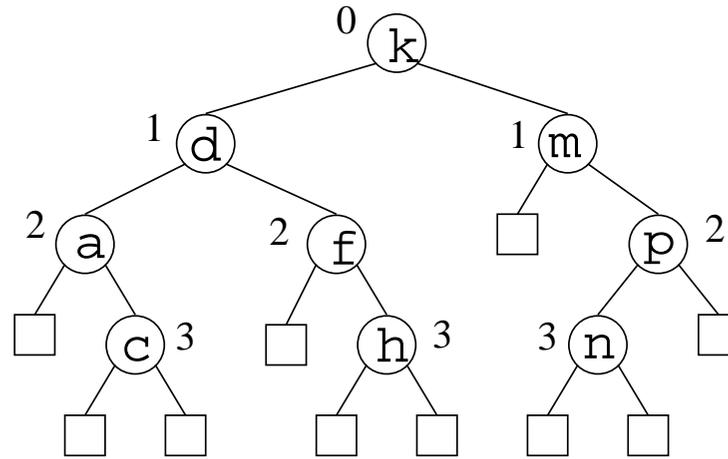
Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge**  $\text{TIPL}(T) = \sum_{v \in V} d(v) (= 0)$

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

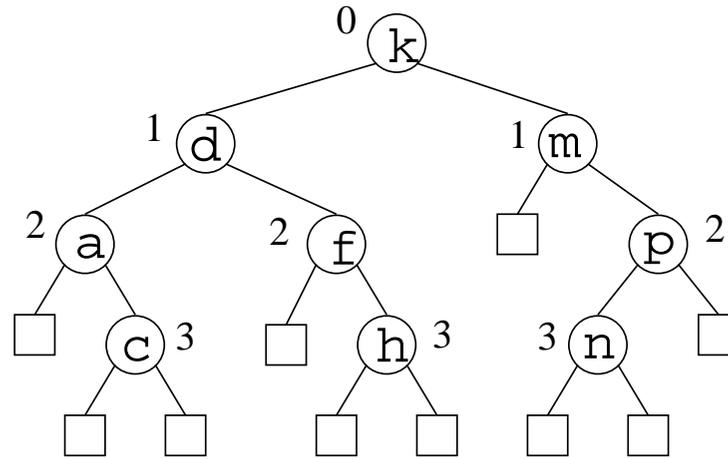
Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge  $TIPL(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1$**

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

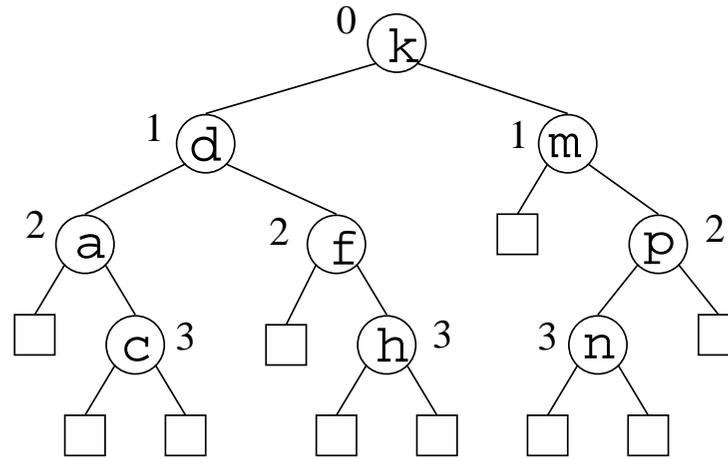
Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge  $TIPL(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2)$**

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

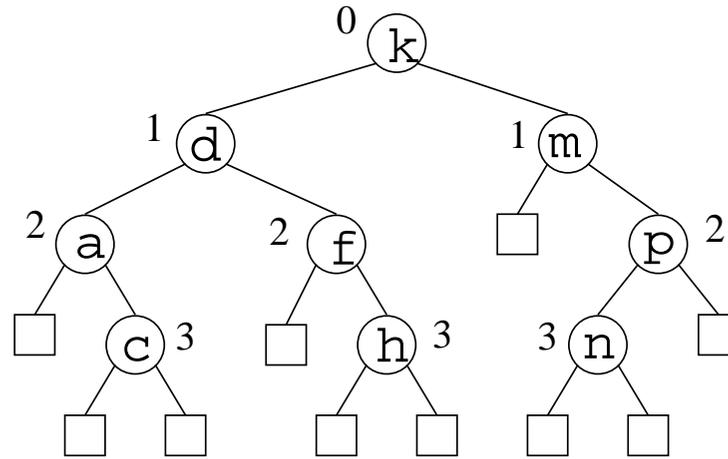
Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge  $TIPL(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3)$**

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

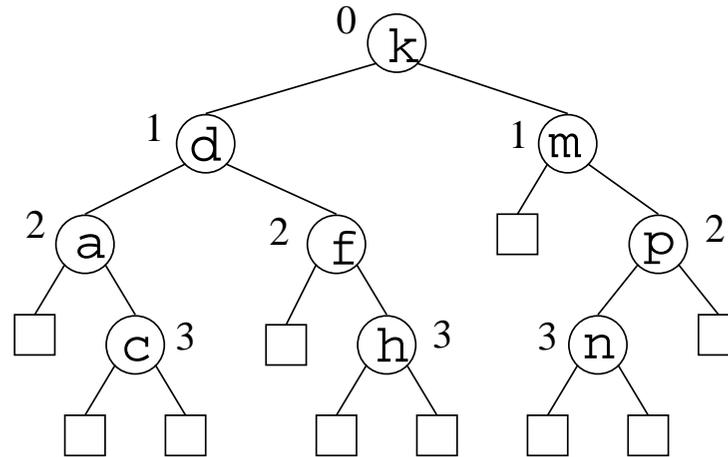
Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge  $TIPL(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17)$ .**

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:  
Summe der Knotentiefen.

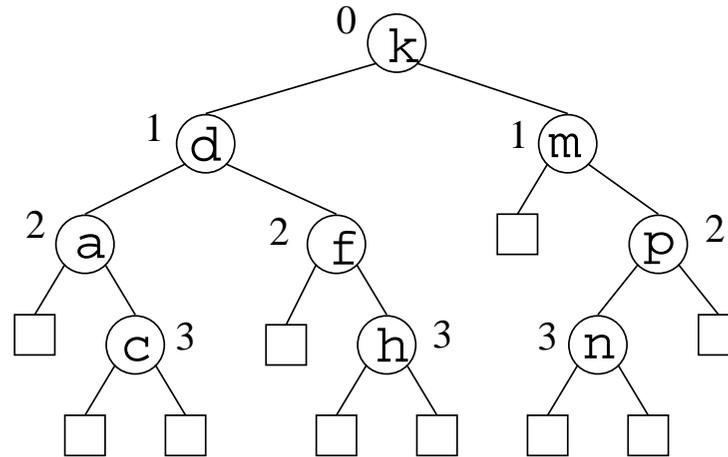


**Totale innere Weglänge**  $\text{TIPL}(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17)$ .

**Mittlere** innere Weglänge:  $\frac{1}{n} \text{TIPL}(T)$  (hier:  $\frac{17}{9}$ ).

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:  
Summe der Knotentiefen.



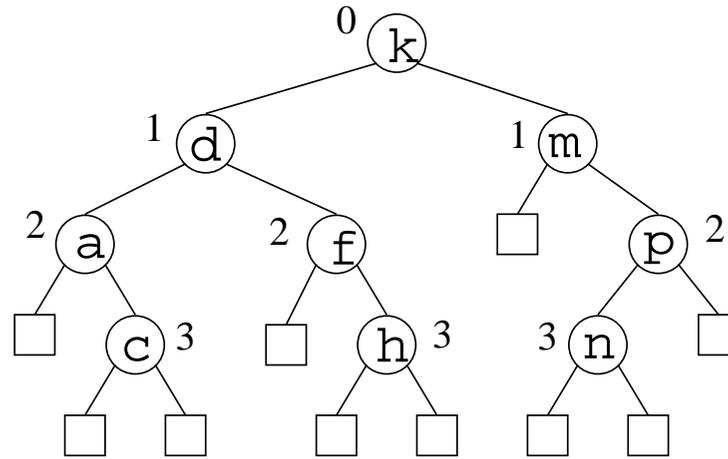
**Totale innere Weglänge**  $\text{TIPL}(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17)$ .

**Mittlere** innere Weglänge:  $\frac{1}{n} \text{TIPL}(T)$  (hier:  $\frac{17}{9}$ ).

Totale Anzahl von durchlaufenen Knoten, wenn man zu jedem einmal hinläuft:  $n + \text{TIPL}(T)$ .

## 4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:  
Summe der Knotentiefen.



**Totale innere Weglänge**  $\text{TIPL}(T) = \sum_{v \in V} d(v) (= 0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17)$ .

**Mittlere** innere Weglänge:  $\frac{1}{n} \text{TIPL}(T)$  (hier:  $\frac{17}{9}$ ).

Totale Anzahl von durchlaufenen Knoten, wenn man zu jedem einmal hinläuft:  $n + \text{TIPL}(T)$ .

Mittelwert  $1 + \frac{1}{n} \text{TIPL}(T)$ : erwartete Kosten, wenn jeder Schlüssel mit Wahrscheinlichkeit  $\frac{1}{n}$  gesucht wird.

---

# Zufällig erzeugte binäre Suchbäume

---

# Zufällig erzeugte binäre Suchbäume

$T$  „zufällig erzeugt“:

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein.

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ .

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**)

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**)

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

$$\mathbf{E} \left( O \left( 1 + \frac{1}{n} \text{TIPL}(T) \right) \right) = O \left( 1 + \frac{1}{n} A(n) \right).$$

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

$$\mathbf{E} \left( O \left( 1 + \frac{1}{n} \text{TIPL}(T) \right) \right) = O \left( 1 + \frac{1}{n} A(n) \right).$$

Was ist  $A(n)$ ?

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

$$\mathbf{E} \left( O \left( 1 + \frac{1}{n} \text{TIPL}(T) \right) \right) = O \left( 1 + \frac{1}{n} A(n) \right).$$

Was ist  $A(n)$ ?  $A(0) = 0$

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

$$\mathbf{E} \left( O \left( 1 + \frac{1}{n} \text{TIPL}(T) \right) \right) = O \left( 1 + \frac{1}{n} A(n) \right).$$

Was ist  $A(n)$ ?  $A(0) = 0, \quad A(1) = 0$

---

## Zufällig erzeugte binäre Suchbäume

$T$  „**zufällig erzeugt**“: Gegeben:  $S = \{x_1, \dots, x_n\} \subseteq U$  mit  $x_1 < \dots < x_n$ .

Starte mit leerem Baum, füge Schlüssel aus  $S$  nacheinander ein. Dabei:

Eingabereihenfolge der Schlüssel ist „rein zufällig“, d. h. für jede Permutation  $\pi$  hat die Reihenfolge  $x_{\pi(1)}, \dots, x_{\pi(n)}$  Wahrscheinlichkeit  $1/n!$ . Definiere:

$$A(n) := \mathbf{E}(\text{TIPL}(T)) \quad (\text{Erwartungswert}).$$

Dann ist der **erwartete** (über **Eingabereihenfolge zufällig**) **mittlere** (über  $n$  **Schlüssel**  $x \in S$  **gemittelte**) Aufwand für **lookup**( $x$ ):

$$\mathbf{E} \left( O \left( 1 + \frac{1}{n} \text{TIPL}(T) \right) \right) = O \left( 1 + \frac{1}{n} A(n) \right).$$

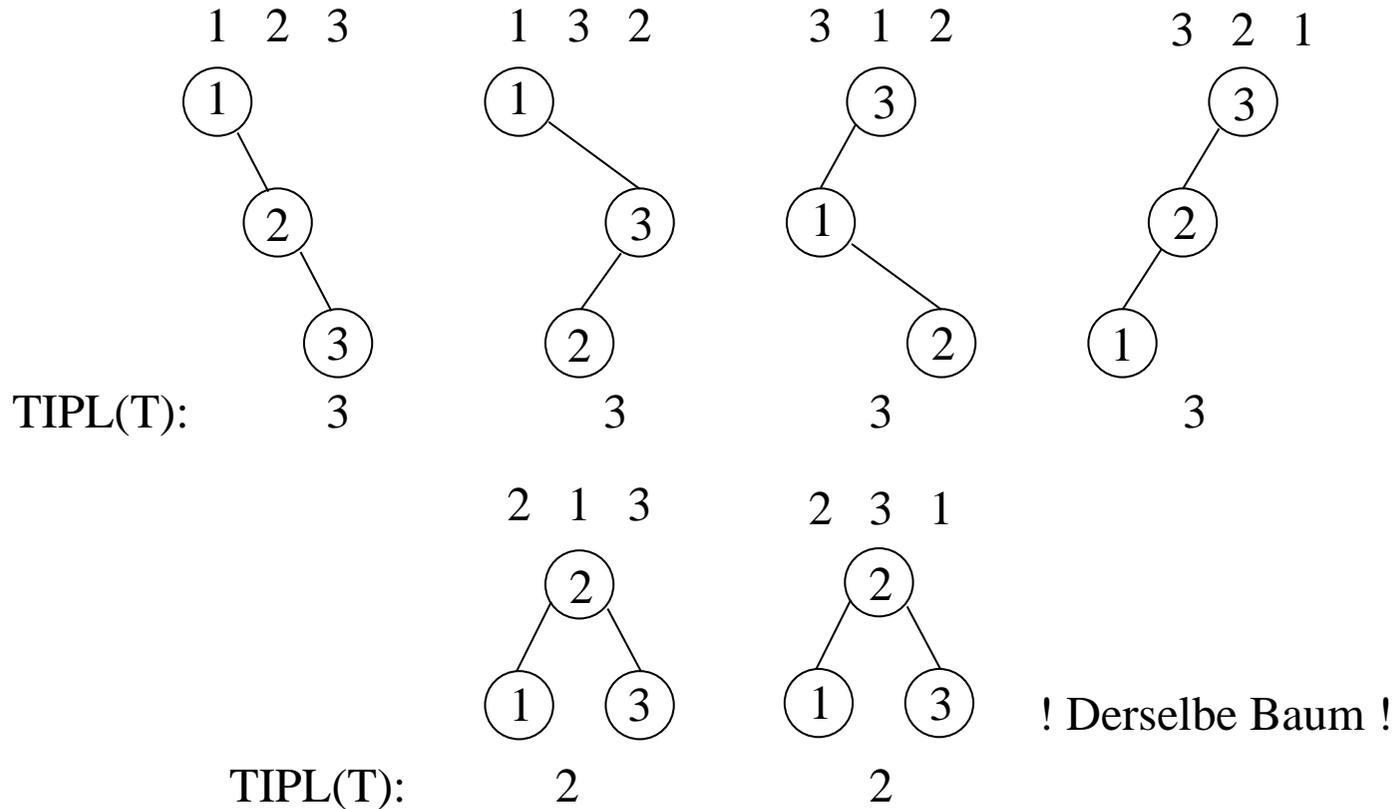
Was ist  $A(n)$ ?  $A(0) = 0, A(1) = 0, A(2) = 1, \dots$

---

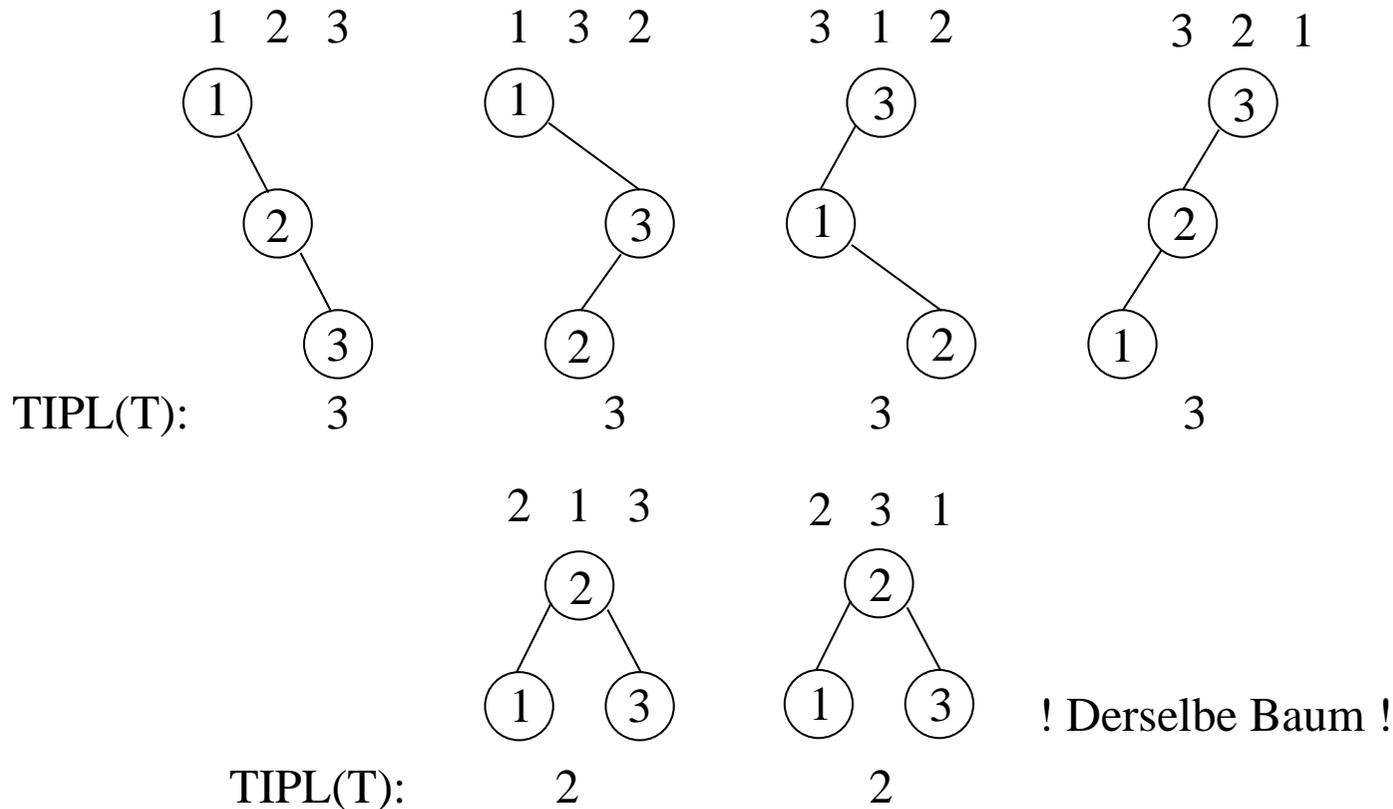
$A(3) = ?$  – 6 Einfügereihenfolgen für  $(x_1, x_2, x_3) = (1, 2, 3)$ :

---

$A(3) = ?$  – 6 Einfügereihenfolgen für  $(x_1, x_2, x_3) = (1, 2, 3)$ :



$A(3) = ?$  – 6 Einfügereihenfolgen für  $(x_1, x_2, x_3) = (1, 2, 3)$ :



$$A(3) = \mathbf{E}(\text{TIPL}(T)) = \frac{1}{6} (3 + 3 + 3 + 3 + 2 + 2) = \frac{8}{3}.$$

---

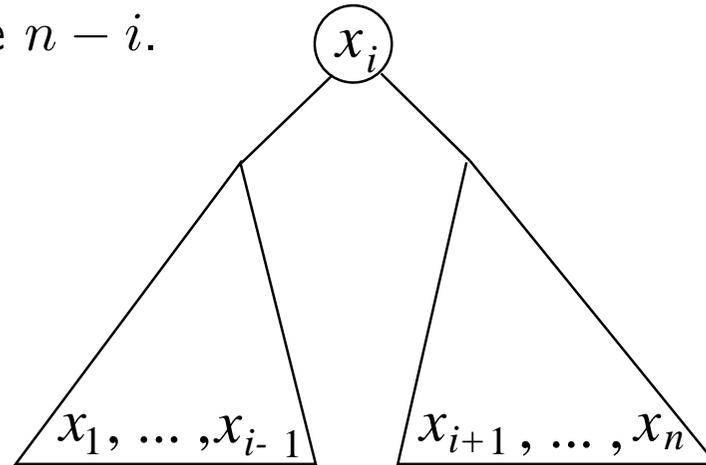
## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .

---

## Rekursionsformel

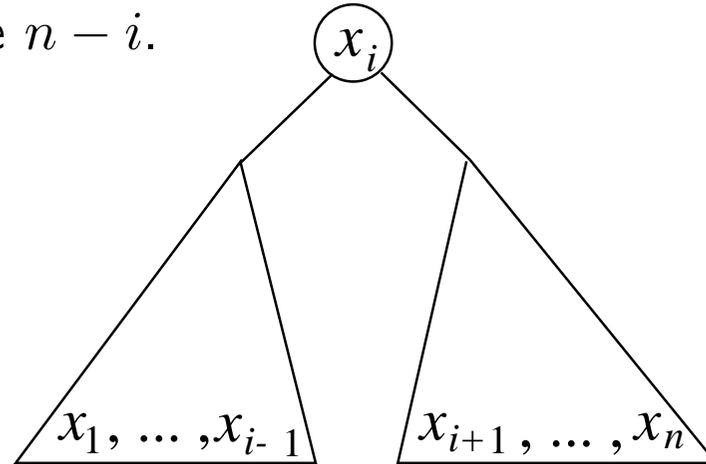
Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .



---

## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .

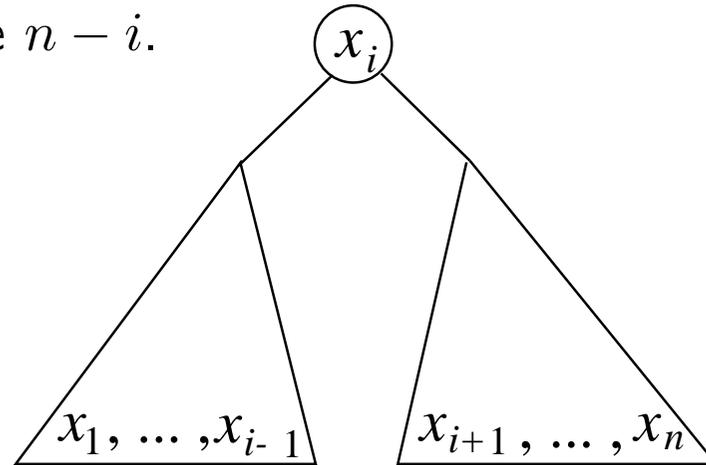


Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume.

---

## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .



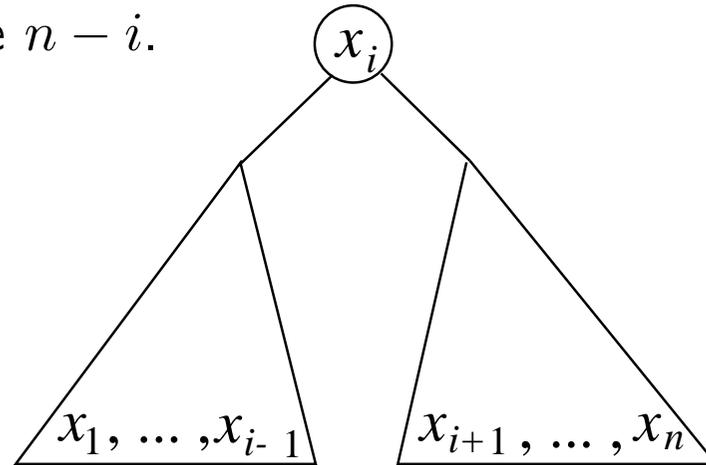
Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = \dots$$

---

## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .



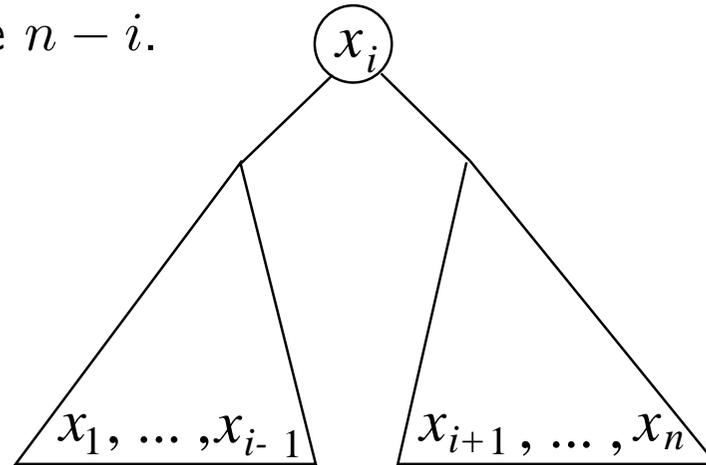
Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = \dots = ((i - 1) + A(i - 1))$$

---

## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .



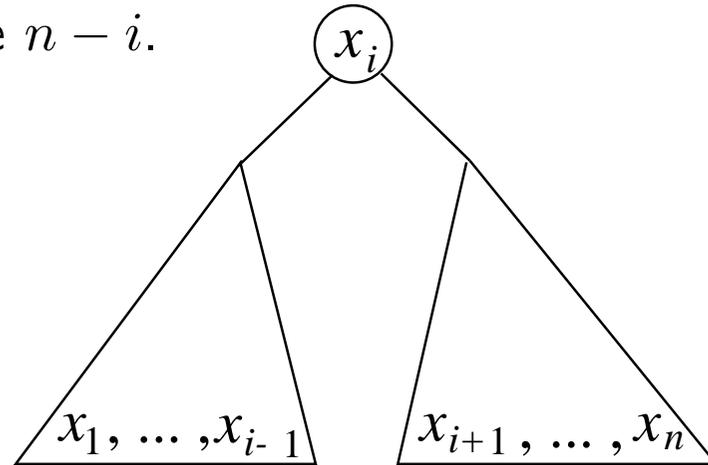
Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = \dots = ((i - 1) + A(i - 1)) + ((n - i) + A(n - i)).$$

---

## Rekursionsformel

Wenn  $x_i$  der erste eingefügte Schlüssel ist, dann hat der linke Unterbaum von  $T$  genau  $i - 1$  Schlüssel, der rechte  $n - i$ .



Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = \dots = ((i - 1) + A(i - 1)) + ((n - i) + A(n - i)).$$

**Begründung:** Der Weg in  $T$  zu jedem der  $n - 1$  Knoten in den Teilbäumen ist um 1 länger als der Weg im Teilbaum; daher die Summanden  $(i - 1)$ ,  $(n - i)$ .

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden.

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n)$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n}$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i))$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (A(i - 1) + A(n - i))$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (A(i - 1) + A(n - i)) = n - 1$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (A(i - 1) + A(n - i)) = n - 1 + \frac{2}{n}$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (A(i - 1) + A(n - i)) = n - 1 + \frac{2}{n} \cdot \sum_{0 \leq j \leq n-1} A(j).$$

---

Was wir hatten, etwas vereinfacht:

$$\mathbf{E}(\text{TIPL}(T) \mid \text{falls } x_i \text{ erster}) = n - 1 + A(i - 1) + A(n - i).$$

Jeder der  $n$  Schlüssel  $x_1, \dots, x_n$  hat dieselbe Wahrscheinlichkeit  $\frac{1}{n}$ , als erster eingefügt zu werden und die Wurzel zu bilden. Mittelung liefert den E.-wert:

$$A(n) = \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (n - 1 + A(i - 1) + A(n - i)).$$

Das heißt:

$$A(n) = n - 1 + \frac{1}{n} \cdot \sum_{1 \leq i \leq n} (A(i - 1) + A(n - i)) = n - 1 + \frac{2}{n} \cdot \sum_{0 \leq j \leq n-1} A(j).$$

(Begründung:  $i - 1$  durchläuft  $j = 0, 1, \dots, n - 1$ ;  $n - i$  durchläuft  $j = n - 1, \dots, 1, 0$ .)

---

Wir haben also die folgende Rekurrenzgleichung:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

$$A(2) = 1 + \frac{2}{2} \cdot (0) = 1;$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

$$A(2) = 1 + \frac{2}{2} \cdot (0) = 1;$$

$$A(3) = 2 + \frac{2}{3} \cdot (0 + 1) = \frac{8}{3};$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

$$A(2) = 1 + \frac{2}{2} \cdot (0) = 1;$$

$$A(3) = 2 + \frac{2}{3} \cdot (0 + 1) = \frac{8}{3};$$

$$A(4) = 3 + \frac{2}{4} \cdot (0 + 1 + \frac{8}{3}) = \frac{29}{6};$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

$$A(2) = 1 + \frac{2}{2} \cdot (0) = 1;$$

$$A(3) = 2 + \frac{2}{3} \cdot (0 + 1) = \frac{8}{3};$$

$$A(4) = 3 + \frac{2}{4} \cdot (0 + 1 + \frac{8}{3}) = \frac{29}{6};$$

$$A(5) = 4 + \frac{2}{5} \cdot (0 + 1 + \frac{8}{3} + \frac{29}{6}) = \frac{37}{5}; \text{ usw.}$$

---

Wir haben also die folgende **Rekurrenzgleichung**:

$$A(0) = 0 \quad \text{und} \quad A(n) = n - 1 + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j) \quad \text{für } n \geq 1.$$

Damit können wir nacheinander Werte berechnen:

$$A(1) = 0 + \frac{2}{1} \cdot \sum_{1 \leq j \leq 0} A(j) = 0;$$

$$A(2) = 1 + \frac{2}{2} \cdot (0) = 1;$$

$$A(3) = 2 + \frac{2}{3} \cdot (0 + 1) = \frac{8}{3};$$

$$A(4) = 3 + \frac{2}{4} \cdot (0 + 1 + \frac{8}{3}) = \frac{29}{6};$$

$$A(5) = 4 + \frac{2}{5} \cdot (0 + 1 + \frac{8}{3} + \frac{29}{6}) = \frac{37}{5}; \text{ usw.}$$

Finden wir eventuell eine geschlossene Form?

---

$$A(n) = (n - 1) + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j)$$

---

$$A(n) = (n - 1) + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Multiplizieren mit  $n$ :

$$nA(n) = n(n - 1) + 2 \cdot \sum_{1 \leq j \leq n-1} A(j)$$

---

$$A(n) = (n - 1) + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Multiplizieren mit  $n$ :

$$nA(n) = n(n - 1) + 2 \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Dasselbe für  $n - 1$ :

$$(n - 1)A(n - 1) = (n - 1)(n - 2) + 2 \cdot \sum_{1 \leq j \leq n-2} A(j)$$

---

$$A(n) = (n - 1) + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Multiplizieren mit  $n$ :

$$nA(n) = n(n - 1) + 2 \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Dasselbe für  $n - 1$ :

$$(n - 1)A(n - 1) = (n - 1)(n - 2) + 2 \cdot \sum_{1 \leq j \leq n-2} A(j)$$

Subtrahieren (dieser Trick eliminiert die Summe!):

$$nA(n) - (n - 1)A(n - 1) = n(n - 1) - (n - 1)(n - 2) + 2A(n - 1)$$

---

$$A(n) = (n - 1) + \frac{2}{n} \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Multiplizieren mit  $n$ :

$$nA(n) = n(n - 1) + 2 \cdot \sum_{1 \leq j \leq n-1} A(j)$$

Dasselbe für  $n - 1$ :

$$(n - 1)A(n - 1) = (n - 1)(n - 2) + 2 \cdot \sum_{1 \leq j \leq n-2} A(j)$$

Subtrahieren (dieser Trick eliminiert die Summe!):

$$\begin{aligned} nA(n) - (n - 1)A(n - 1) &= n(n - 1) - (n - 1)(n - 2) + 2A(n - 1) \\ &= 2(n - 1) + 2A(n - 1). \end{aligned}$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1).$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

Also, für  $n \geq 0$ :

$$Z(n)$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

Also, für  $n \geq 0$ :

$$Z(n) = \sum_{1 \leq j \leq n} \frac{2(j - 1)}{j(j + 1)}$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

Also, für  $n \geq 0$ :

$$Z(n) = \sum_{1 \leq j \leq n} \frac{2(j - 1)}{j(j + 1)} = \sum_{1 \leq j \leq n}$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

Also, für  $n \geq 0$ :

$$Z(n) = \sum_{1 \leq j \leq n} \frac{2(j - 1)}{j(j + 1)} = \sum_{1 \leq j \leq n} \binom{2}{j}$$

---

Wir haben:  $nA(n) - (n - 1)A(n - 1) = 2A(n - 1) + 2(n - 1)$ , d. h.

$$nA(n) - (n + 1)A(n - 1) = 2(n - 1). \quad | : n(n + 1)$$

$$\frac{A(n)}{n + 1} - \frac{A(n - 1)}{n} = \frac{2(n - 1)}{n(n + 1)}.$$

Mit der Abkürzung  $Z(n) := A(n)/(n + 1)$  gilt

$$Z(0) = 0 \quad \text{und} \quad Z(n) - Z(n - 1) = \frac{2(n - 1)}{n(n + 1)}, \quad \text{für } n \geq 1.$$

Also, für  $n \geq 0$ :

$$Z(n) = \sum_{1 \leq j \leq n} \frac{2(j - 1)}{j(j + 1)} = \sum_{1 \leq j \leq n} \left( \frac{2}{j} - \frac{4}{j(j + 1)} \right).$$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j}$$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**)

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und  
 $\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und

$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$  ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und

$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$  ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

Mit  $A(n) = Z(n) \cdot (n+1)$  erhalten wir die „geschlossene Form“

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und

$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$  ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

Mit  $A(n) = Z(n) \cdot (n+1)$  erhalten wir die „geschlossene Form“

$$A(n) = 2(n+1)H_n - 4n$$

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und

$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$  ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

Mit  $A(n) = Z(n) \cdot (n+1)$  erhalten wir die „geschlossene Form“

$$A(n) = 2(n+1)H_n - 4n.$$

**Erwartete mittlere Knotentiefe** (Einfügereihenfolge **zufällig**; **Mittelung** über  $x_1, \dots, x_n$ ):

---

Wir haben:

$$Z(n) = 2 \sum_{1 \leq j \leq n} \frac{1}{j} - 4 \sum_{1 \leq j \leq n} \frac{1}{j(j+1)}.$$

Mit  $H_n := 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$  ( **$n$ -te harmonische Zahl**) und

$\frac{1}{1 \cdot 2} + \frac{1}{2 \cdot 3} + \dots + \frac{1}{n \cdot (n+1)} = (1 - \frac{1}{2}) + (\frac{1}{2} - \frac{1}{3}) + \dots + (\frac{1}{n} - \frac{1}{n+1}) = 1 - \frac{1}{n+1}$  ergibt das:

$$Z(n) = 2H_n - \frac{4n}{n+1}.$$

Mit  $A(n) = Z(n) \cdot (n+1)$  erhalten wir die „geschlossene Form“

$$A(n) = 2(n+1)H_n - 4n.$$

**Erwartete mittlere Knotentiefe** (Einfügereihenfolge **zufällig**; **Mittlung** über  $x_1, \dots, x_n$ ):

$$A(n)/n = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

---

Was ist  $H_n$ ?

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t} .$$

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n$$

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n \leq 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} < H_n.$$

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n \leq 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1} < H_n.$$

Also:

$\ln n < H_n < 1 + \ln n, \text{ für } n \geq 2.$

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n \leq 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} < H_n.$$

Also:

$$\ln n < H_n < 1 + \ln n, \text{ für } n \geq 2.$$

Genauer (ohne Bew.):  $(H_n - \ln n) \nearrow \gamma$ , für  $n \rightarrow \infty$ ,

---

Was ist  $H_n$ ? Für jedes  $i \geq 2$ :

$$\frac{1}{i} < \int_{i-1}^i \frac{dt}{t}.$$

Für jedes  $i \geq 1$ :

$$\int_i^{i+1} \frac{dt}{t} < \frac{1}{i}.$$

Daraus:

$$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n \leq 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} < H_n.$$

Also:

$\ln n < H_n < 1 + \ln n, \text{ für } n \geq 2.$

Genauer (ohne Bew.):  $(H_n - \ln n) \nearrow \gamma$ , für  $n \rightarrow \infty$ , wobei  $\gamma = 0,57721 \dots$  („Euler-Konstante“).

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

### Satz 4.2.1

$$\frac{1}{n}A(n) = 2 \ln n - (4 - 2\gamma) + o(1)$$

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

### Satz 4.2.1

$$\frac{1}{n}A(n) = 2 \ln n - (4 - 2\gamma) + o(1)$$

Wenn man Zweierlogarithmen bevorzugt:

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

### Satz 4.2.1

$$\frac{1}{n}A(n) = 2 \ln n - (4 - 2\gamma) + o(1)$$

Wenn man Zweierlogarithmen bevorzugt:

$$\frac{1}{n}A(n) = (2 \ln 2) \log n - 2,846 \dots + o(1),$$

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

### Satz 4.2.1

$$\frac{1}{n}A(n) = 2 \ln n - (4 - 2\gamma) + o(1)$$

Wenn man Zweierlogarithmen bevorzugt:

$$\frac{1}{n}A(n) = (2 \ln 2) \log n - 2,846 \dots + o(1), \text{ mit } 2 \ln 2 = 1,386 \dots$$

---

Hatten schon:

$$\frac{1}{n}A(n) = 2H_n - 4 + O\left(\frac{H_n}{n}\right).$$

Eben gesehen:  $\gamma = \lim_{n \rightarrow \infty} (H_n - \ln n) = \gamma \approx 0,57721$ .

### Satz 4.2.1

$$\frac{1}{n}A(n) = 2 \ln n - (4 - 2\gamma) + o(1)$$

Wenn man Zweierlogarithmen bevorzugt:

$$\frac{1}{n}A(n) = (\mathbf{2 \ln 2}) \log n - 2,846 \dots + o(1), \text{ mit } 2 \ln 2 = \mathbf{1,386 \dots}$$

**Satz 4.2.1** (Kurzform) In zufällig erzeugten binären Suchbäumen ist die **erwartete mittlere** Knotentiefe etwa  $1,39 \log n$ .

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  
 $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  
 $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  
 $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

Ungünstig: Daten perfekt oder partiell sortiert. (Und das ist nicht ungewöhnlich.)

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

Ungünstig: Daten perfekt oder partiell sortiert. (Und das ist nicht ungewöhnlich.)

Zudem: Häufiges Löschen mit **extractMin** zerstört die Balance, mittlere Suchzeit wird  $\Omega(\sqrt{n})$ .

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

Ungünstig: Daten perfekt oder partiell sortiert. (Und das ist nicht ungewöhnlich.)

Zudem: Häufiges Löschen mit **extractMin** zerstört die Balance, mittlere Suchzeit wird  $\Omega(\sqrt{n})$ .

**Abhilfe hierfür:** Bei Löschungen von Knoten mit nichtleeren Unterbäumen **abwechselnd kleinsten Schlüssel aus rechtem Unterbaum und größten Schlüssel aus linkem Unterbaum** entnehmen.

---

## Mitteilungen:

Erwartete **Baumtiefe**  $d(T)$  für zufällig erzeugtes  $T$  ist ebenfalls  $O(\log n)$ , genauer:  $\mathbf{E}(d(T)) \leq 3 \log n + O(1/n)$ .

Leider ist sehr oft die **Einfügereihenfolge nicht zufällig**.

Ungünstig: Daten perfekt oder partiell sortiert. (Und das ist nicht ungewöhnlich.)

Zudem: Häufiges Löschen mit **extractMin** zerstört die Balance, mittlere Suchzeit wird  $\Omega(\sqrt{n})$ .

**Abhilfe hierfür:** Bei Löschungen von Knoten mit nichtleeren Unterbäumen **abwechselnd kleinsten Schlüssel aus rechtem Unterbaum und größten Schlüssel aus linkem Unterbaum** entnehmen.

Rest dieses Kapitels: **Balancierte** Suchbäume. (Erzwingen logarithmische Tiefe.)

**Vorlesungsvideo:**

# **AVL-Bäume (Einführung)**

---

## 4.3 Balancierte binäre Suchbäume

**Idee:**

---

## 4.3 Balancierte binäre Suchbäume

### Idee:

- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.

---

## 4.3 Balancierte binäre Suchbäume

### Idee:

- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.
- Strukturbedingungen **erzwingen geringe Tiefe** (z. B.  $O(\log n)$  bei  $n$  Einträgen).

---

## 4.3 Balancierte binäre Suchbäume

### Idee:

- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.
- Strukturbedingungen **erzwingen geringe Tiefe** (z. B.  $O(\log n)$  bei  $n$  Einträgen).
- **Implementiere** Wörterbuch-Operationen *insert* und *delete*

---

## 4.3 Balancierte binäre Suchbäume

### Idee:

- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.
- Strukturbedingungen **erzwingen geringe Tiefe** (z. B.  $O(\log n)$  bei  $n$  Einträgen).
- **Implementiere** Wörterbuch-Operationen *insert* und *delete* so, dass
  - (i) sie die **Strukturbedingungen erhalten** und
  - (ii) ihre **Kosten  $O(\text{Baumtiefe})$**  sind.

---

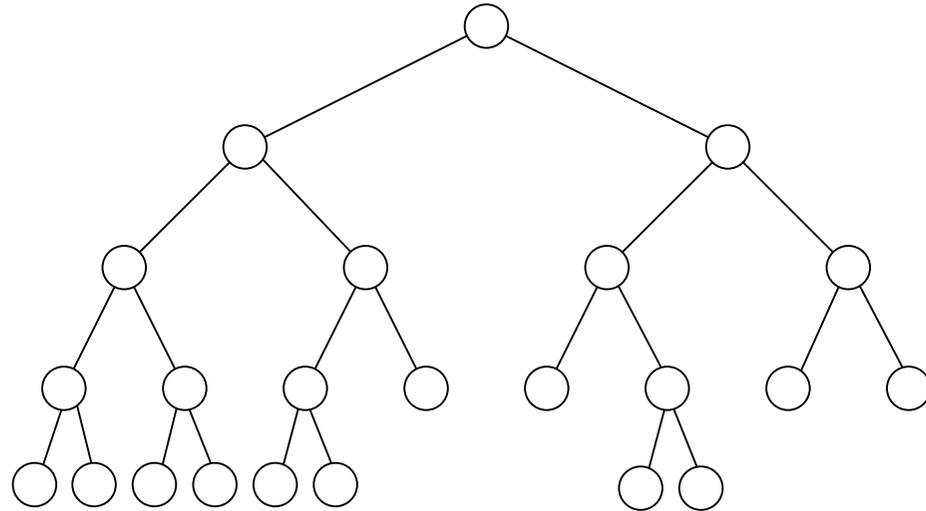
?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:

---

?? Attraktiv: Perfekte Balance.

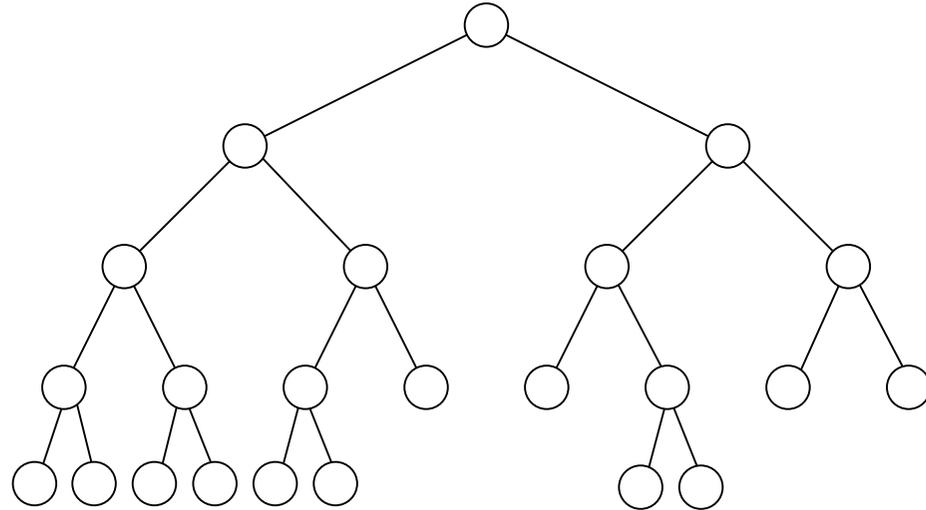
D. h.: Alle Blätter auf zwei benachbarten Levels:



---

?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:

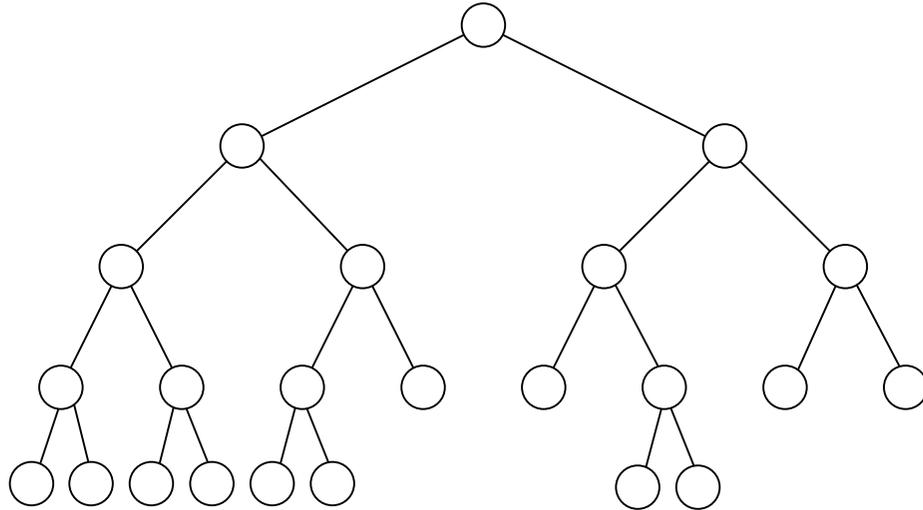


Tiefe  $d$  erfüllt  $2^d - 1 < n \leq 2^{d+1} - 1$

---

?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:

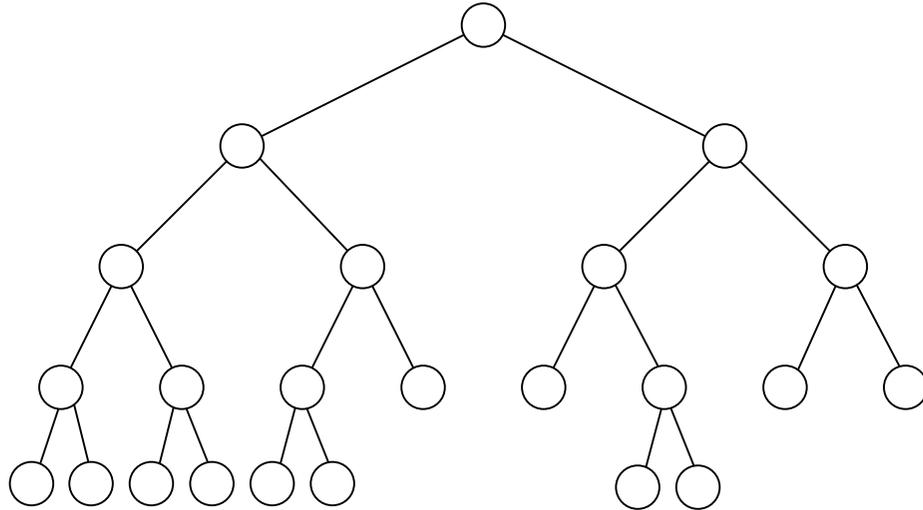


Tiefe  $d$  erfüllt  $2^d - 1 < n \leq 2^{d+1} - 1$ , also  $d < \log(n + 1) \leq d + 1$

---

?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:



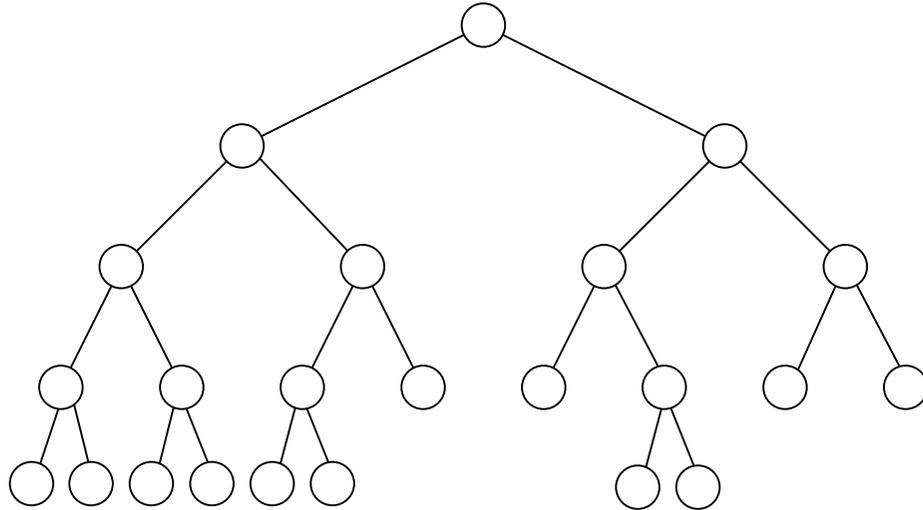
Tiefe  $d$  erfüllt  $2^d - 1 < n \leq 2^{d+1} - 1$ , also  $d < \log(n + 1) \leq d + 1$ , also

$d = \lceil \log(n + 1) \rceil - 1$ . (Bemerkung:  $\lceil \log(n + 1) \rceil = |\text{bin}(n)|$ , Länge der Binärdarst.)

---

?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:



Tiefe  $d$  erfüllt  $2^d - 1 < n \leq 2^{d+1} - 1$ , also  $d < \log(n + 1) \leq d + 1$ , also

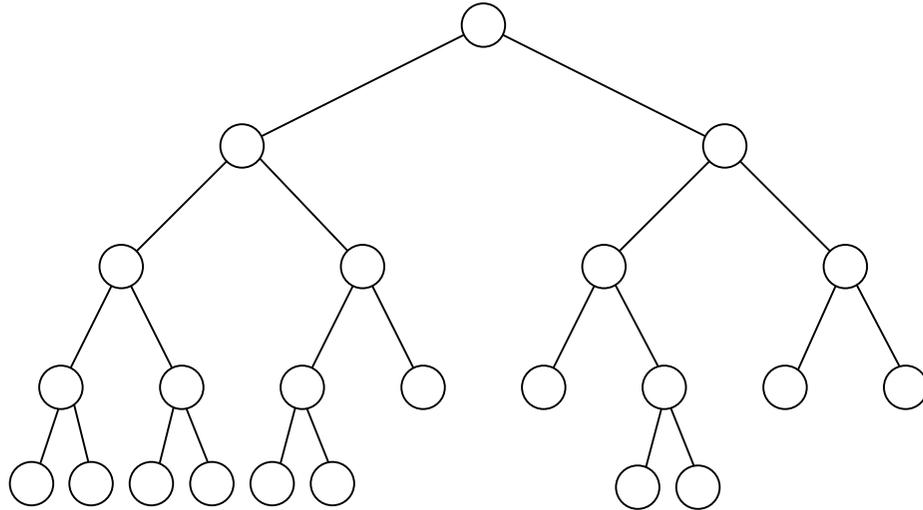
$d = \lceil \log(n + 1) \rceil - 1$ . (Bemerkung:  $\lceil \log(n + 1) \rceil = |\text{bin}(n)|$ , Länge der Binärdarst.)

Attraktiv??

---

?? Attraktiv: Perfekte Balance.

D. h.: Alle Blätter auf zwei benachbarten Levels:



Tiefe  $d$  erfüllt  $2^d - 1 < n \leq 2^{d+1} - 1$ , also  $d < \log(n + 1) \leq d + 1$ , also  $d = \lceil \log(n + 1) \rceil - 1$ . (Bemerkung:  $\lceil \log(n + 1) \rceil = |\text{bin}(n)|$ , Länge der Binärdarst.)

Attraktiv?? Nein! Einfüge- und Lösch-Operationen zu teuer.

---

## Häufig benutzte Strukturbedingungen:

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume**

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume**

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)
- **2-3-4-Bäume**

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)
- **2-3-4-Bäume** ([Cormen et al.], [Sedgewick],[D./Mehlhorn/Sanders])

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)
- **2-3-4-Bäume** ([Cormen et al.], [Sedgewick],[D./Mehlhorn/Sanders])
- **B-Bäume**

---

## Häufig benutzte Strukturbedingungen:

- **AVL-Bäume** (behandeln wir ausführlich)
- **Rot-Schwarz-Bäume** ([Cormen et al.], [Sedgewick])
- **2-3-Bäume** (behandeln wir prinzipiell)
- **2-3-4-Bäume** ([Cormen et al.], [Sedgewick],[D./Mehlhorn/Sanders])
- **B-Bäume** ([Cormen et al.], Vorlesungen zu Datenbanktechnologie)

---

## 4.3.1 AVL-Bäume

### Höhenbalancierte binäre Suchbäume

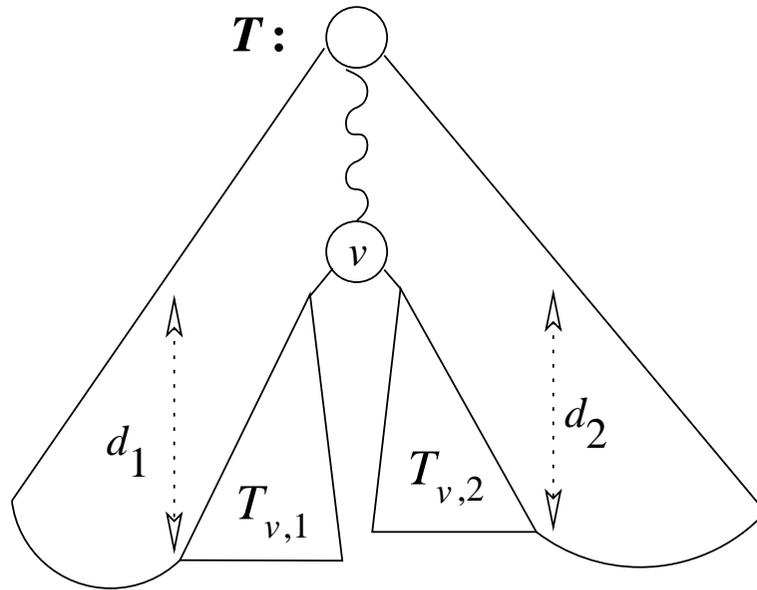
[G. M. Adelson-Velski und J. M. Landis 1962]

---

## 4.3.1 AVL-Bäume

### Höhenbalancierte binäre Suchbäume

[G. M. Adelson-Velski und J. M. Landis 1962]

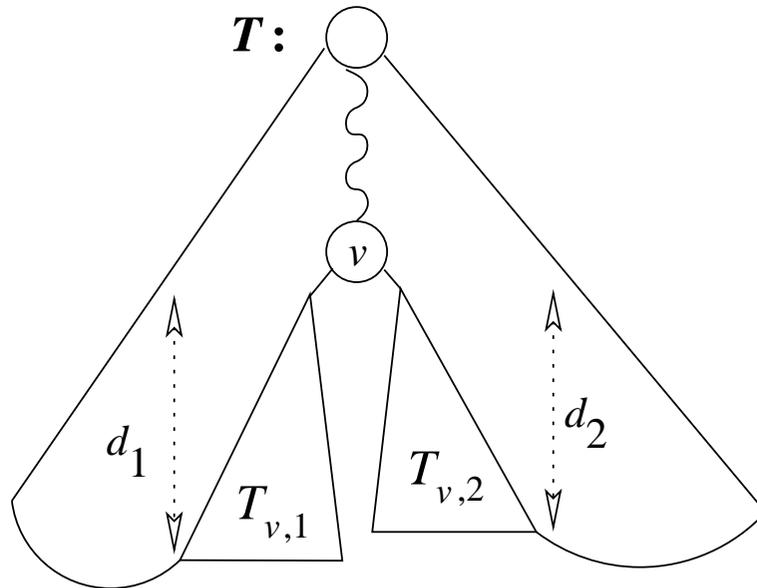


---

## 4.3.1 AVL-Bäume

### Höhenbalancierte binäre Suchbäume

[G. M. Adelson-Velski und J. M. Landis 1962]



$$|d_2 - d_1| \leq 1$$

---

## Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**,

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

(i) Der leere Baum  $\square$  ist höhenbalancierter BB über  $U$ .

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

- (i) Der leere Baum  $\square$  ist höhenbalancierter BB über  $U$ .
- (ii) Sind  $T_1$  und  $T_2$  höhenbalancierte BB

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

- (i) Der leere Baum  $\square$  ist höhenbalancierter BB über  $U$ .
- (ii) Sind  $T_1$  und  $T_2$  höhenbalancierte BB,  $x \in U$ ,

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

- (i) Der leere Baum  $\square$  ist höhenbalancierter BB über  $U$ .
- (ii) Sind  $T_1$  und  $T_2$  höhenbalancierte BB,  $x \in U$ , und ist  $bal = d(T_2) - d(T_1) \in \{-1, 0, 1\}$ ,

---

### Definition 4.3.1

Ein **Binärbaum**  $T$  heißt **höhenbalanciert**, falls in **jedem Knoten**  $v$  in  $T$  für den Teilbaum  $T_v = (T_{v,1}, v, T_{v,2})$  mit Wurzel  $v$  gilt:

$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} .$$

Äquivalent ist die folgende **rekursive** Charakterisierung:

- (i) Der leere Baum  $\square$  ist höhenbalancierter BB über  $U$ .
- (ii) Sind  $T_1$  und  $T_2$  höhenbalancierte BB,  $x \in U$ , und ist  $bal = d(T_2) - d(T_1) \in \{-1, 0, 1\}$ ,  
so ist  $(T_1, x, T_2)$  höhenbalancierter BB über  $U$  (mit **Balancefaktor**  $bal$ ).

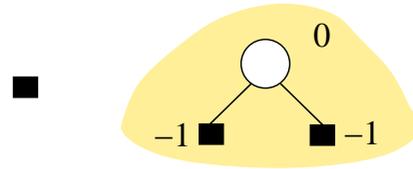
---

*Beispiele:*

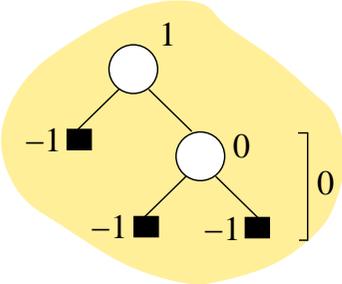
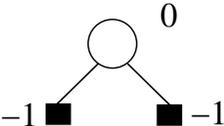


---

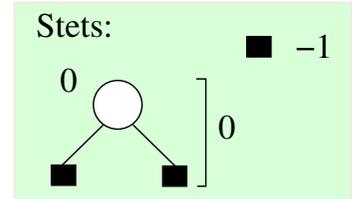
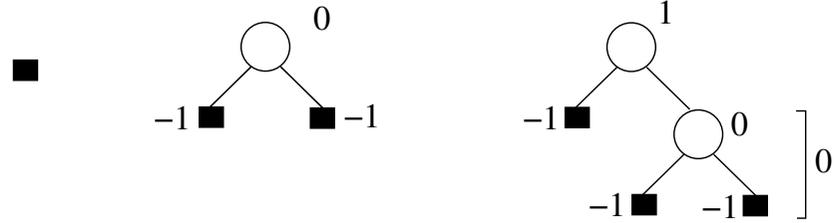
*Beispiele:*



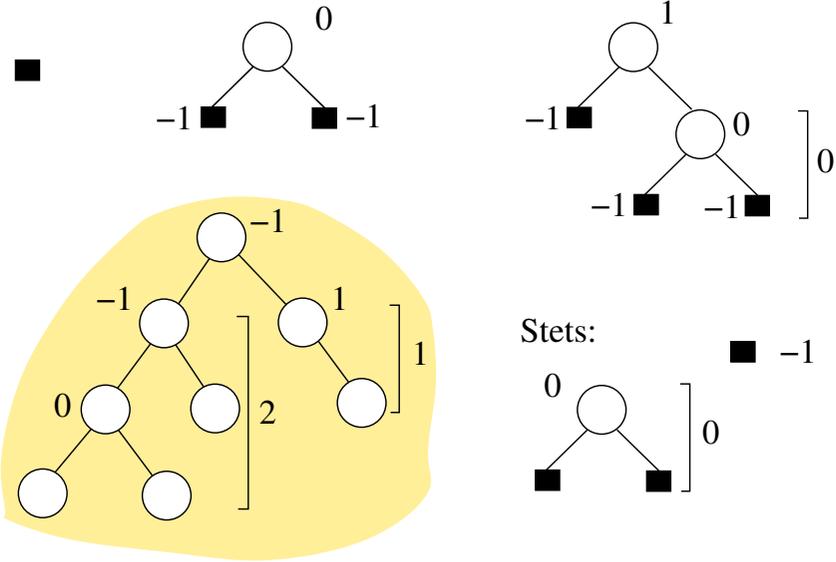
Beispiele:



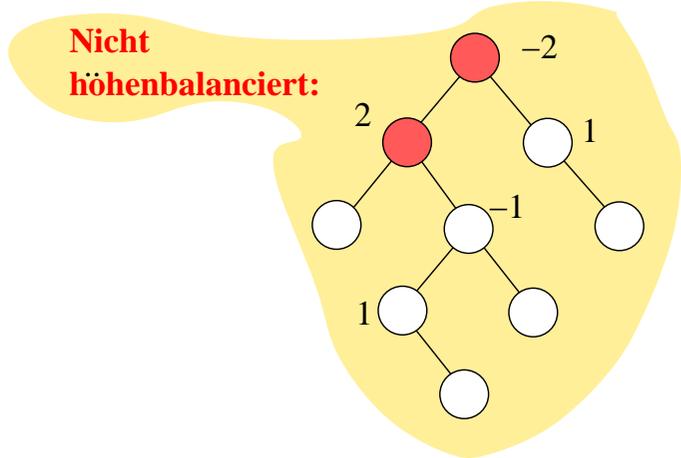
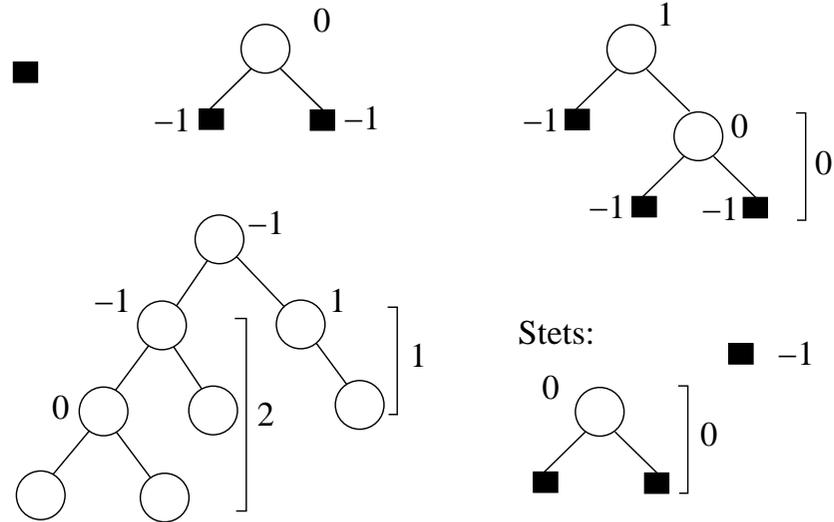
*Beispiele:*



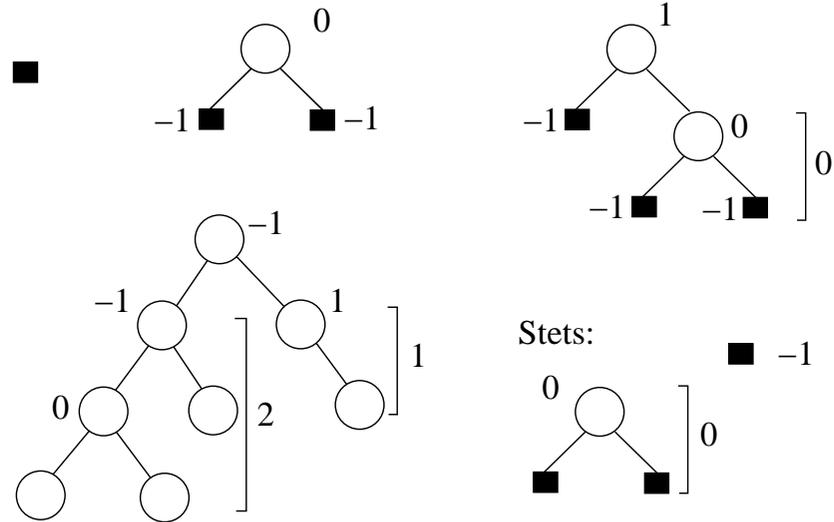
Beispiele:



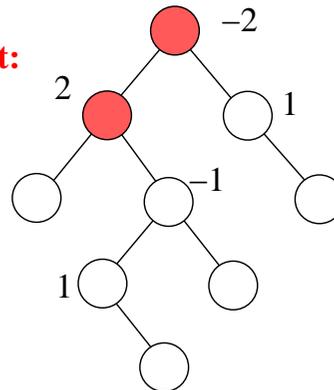
Beispiele:



Beispiele:



**Nicht  
höhenbalanciert:**



---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor  **$\text{bal}(v)$**  speichern.

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal( $v$ )** speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief.

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal( $v$ )** speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal( $v$ )** speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal**( $v$ ) speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

Ist  $T$  ein höhenbalancierter Baum mit  $n \geq 1$  Knoten, so gilt

$$d(T) \leq 1,4405 \cdot \log_2 n.$$

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal**( $v$ ) speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

Ist  $T$  ein höhenbalancierter Baum mit  $n \geq 1$  Knoten, so gilt

$$d(T) \leq 1,4405 \cdot \log_2 n.$$

D. h.: AVL-Bäume sind höchstens um den Faktor 1,4405 tiefer als vollständig balancierte binäre Suchbäume mit derselben Knotenzahl.

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal**( $v$ ) speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

Ist  $T$  ein höhenbalancierter Baum mit  $n \geq 1$  Knoten, so gilt

$$d(T) \leq 1,4405 \cdot \log_2 n.$$

D. h.: AVL-Bäume sind höchstens um den Faktor 1,4405 tiefer als vollständig balancierte binäre Suchbäume mit derselben Knotenzahl.

Beweisansatz: Wir zeigen, dass ein AVL-Baum mit Tiefe  $d$  exponentiell in  $d$  viele Knoten haben muss,

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal**( $v$ ) speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

Ist  $T$  ein höhenbalancierter Baum mit  $n \geq 1$  Knoten, so gilt

$$d(T) \leq 1,4405 \cdot \log_2 n.$$

D. h.: AVL-Bäume sind höchstens um den Faktor 1,4405 tiefer als vollständig balancierte binäre Suchbäume mit derselben Knotenzahl.

Beweisansatz: Wir zeigen, dass ein AVL-Baum mit Tiefe  $d$  exponentiell in  $d$  viele Knoten haben muss, nämlich mindestens  $\Phi^d$  viele für eine Konstante  $\Phi > 1$ ,

---

## Definition 4.3.2

Ein **AVL-Baum** ist ein **höhenbalancierter binärer Suchbaum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten  $v$  seinen Balancefaktor **bal**( $v$ ) speichern.

Hoffnung: AVL-Bäume sind nicht allzu tief. Tatsächlich: „. . . logarithmisch tief“!

## Satz 4.3.3

Ist  $T$  ein höhenbalancierter Baum mit  $n \geq 1$  Knoten, so gilt

$$d(T) \leq 1,4405 \cdot \log_2 n.$$

D. h.: AVL-Bäume sind höchstens um den Faktor 1,4405 tiefer als vollständig balancierte binäre Suchbäume mit derselben Knotenzahl.

Beweisansatz: Wir zeigen, dass ein AVL-Baum mit Tiefe  $d$  exponentiell in  $d$  viele Knoten haben muss, nämlich mindestens  $\Phi^d$  viele für eine Konstante  $\Phi > 1$ , mit  $\log_\Phi 2 \leq 1,4405$ .

---

*Beweis* von Satz 4.3.3:

---

*Beweis* von Satz 4.3.3: Für  $d = 0, 1, 2, \dots$  setze

$N(d) :=$  **minimale Anzahl innerer Knoten** in einem AVL-Baum mit Tiefe  $d$ .

---

*Beweis* von Satz 4.3.3: Für  $d = 0, 1, 2, \dots$  setze

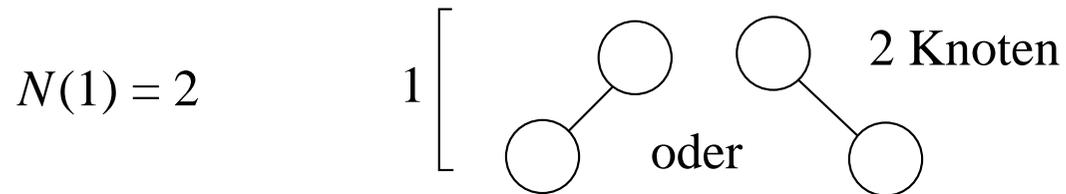
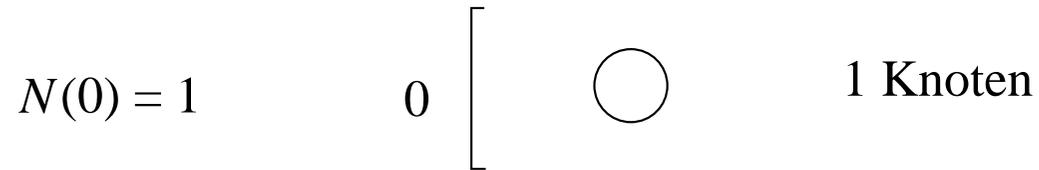
$N(d) :=$  **minimale Anzahl innerer Knoten** in einem AVL-Baum mit Tiefe  $d$ .

$$N(0) = 1 \quad 0 \left[ \begin{array}{c} \bigcirc \end{array} \right. \quad 1 \text{ Knoten}$$

---

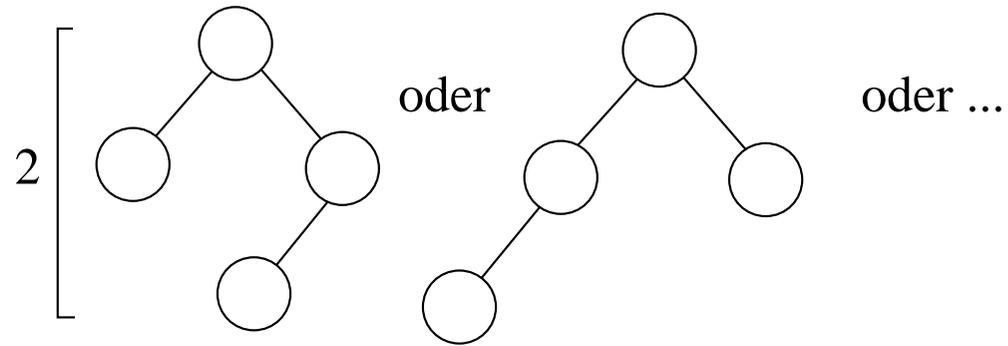
*Beweis* von Satz 4.3.3: Für  $d = 0, 1, 2, \dots$  setze

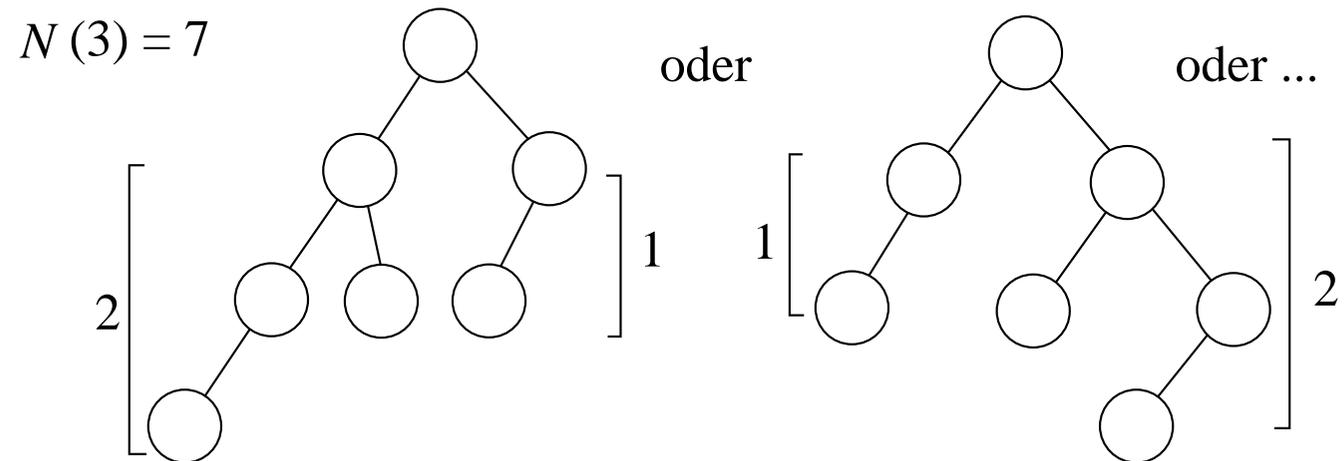
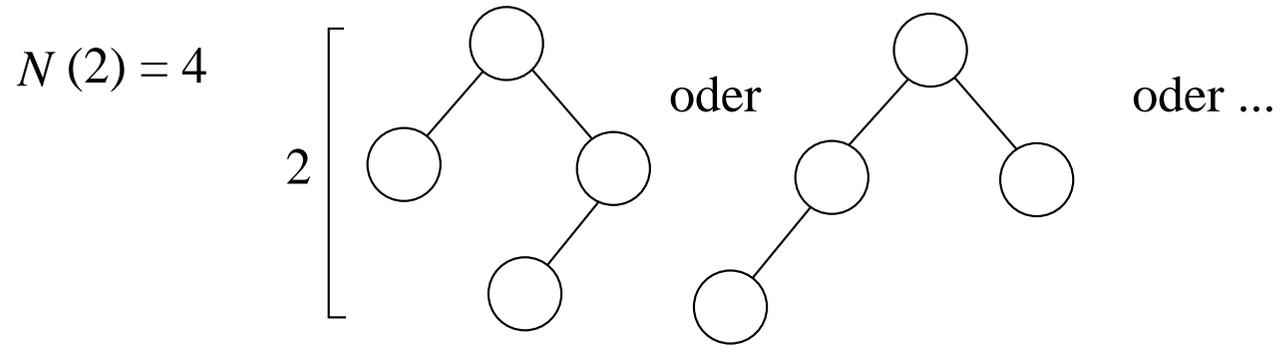
$N(d) :=$  **minimale Anzahl innerer Knoten** in einem AVL-Baum mit Tiefe  $d$ .



---

$N(2) = 4$



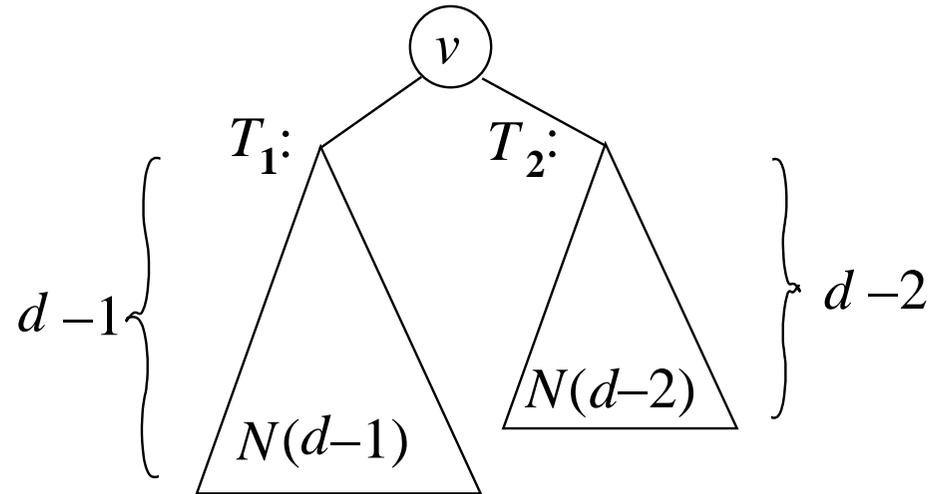


---

**Behauptung 1:**  $N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ .

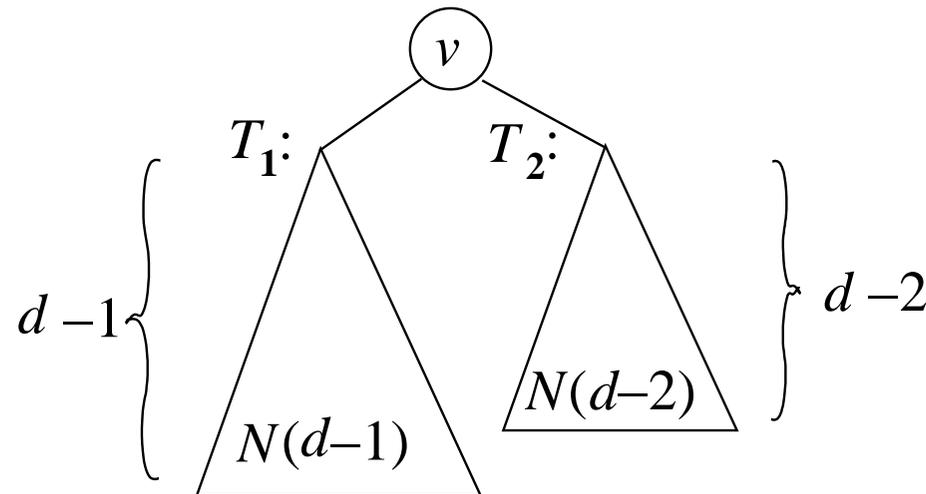
---

**Behauptung 1:**  $N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ .



---

**Behauptung 1:**  $N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ .



*Beweis:* Damit der höhenbalancierte Baum  $T$  der Tiefe  $d$  minimale Knotenzahl hat, muss ein Unterbaum von  $T$  Tiefe  $d - 1$  und der andere Tiefe  $d - 2$  haben. (Es ist klar, dass  $N(d - 1) > N(d - 2)$  gilt.) Weiter müssen die Unterbäume minimale Knotenzahl für ihre jeweilige Tiefe haben.  $\square$

---

$$N(d) = 1 + N(d - 1) + N(d - 2), \text{ für } d \geq 2.$$

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143

Eindruck: exponentielles Wachstum.

---

$N(d) = 1 + N(d - 1) + N(d - 2)$ , für  $d \geq 2$ . –  $N(d)$  für kleine  $d$ :

$d$	$N(d)$
0	1
1	2
2	4
3	7
4	12
5	20
6	33
7	54
8	88
9	143

Eindruck: exponentielles Wachstum.

Man kann zeigen:  $N(d) = F_{d+3} - 1$ , für die **Fibonacci-Zahlen**  $F_0, F_1, F_2, \dots$

---

## Behauptung 2:

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5})$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2:

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („**goldener Schnitt**“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ;

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ .

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ . **I.-Schritt:**

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ . **I.-Schritt:**

$$N(d)$$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ . **I.-Schritt:**

$$N(d) \stackrel{\text{Beh. 1}}{=} 1 + N(d-1) + N(d-2)$$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ . **I.-Schritt:**

$$\begin{aligned} N(d) &\stackrel{\text{Beh. 1}}{=} 1 + N(d-1) + N(d-2) \\ &\stackrel{\text{I.V.}}{\geq} \Phi^{d-1} + \Phi^{d-2} = \Phi^{d-2}(\Phi + 1) \end{aligned}$$

---

## Behauptung 2:

$$N(d) \geq \Phi^d, \text{ für } d \geq 0,$$

wobei  $\Phi = \frac{1}{2}(1 + \sqrt{5}) \approx 1,618\dots$  („goldener Schnitt“).

(\*)  $\Phi$  ist (eine, nämlich die positive) Lösung der Gleichung  $x^2 = x + 1$ .

*Beweis* von Beh. 2: Vollständige Induktion.

**I.A.:** Für  $d = 0$ :  $N(0) = 1 = \Phi^0$ ; für  $d = 1$ :  $N(1) = 2 > \Phi^1$ .

Sei nun  $d \geq 2$ . **I.V.:** Behauptung gilt für  $d' < d$ . **I.-Schritt:**

$$\begin{aligned} N(d) &\stackrel{\text{Beh. 1}}{=} 1 + N(d-1) + N(d-2) \\ &\stackrel{\text{I.V.}}{\geq} \Phi^{d-1} + \Phi^{d-2} = \Phi^{d-2}(\Phi + 1) \stackrel{(*)}{=} \Phi^{d-2} \cdot \Phi^2 = \Phi^d. \end{aligned}$$

□

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$ , d. h.

$$d(T) \leq (\log_{\Phi} 2) \cdot \log_2 n.$$

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$ , d. h.

$$d(T) \leq (\log_{\Phi} 2) \cdot \log_2 n.$$

Es gilt:  $\log_{\Phi} 2 =$

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$ , d. h.

$$d(T) \leq (\log_{\Phi} 2) \cdot \log_2 n.$$

Es gilt:  $\log_{\Phi} 2 = (\ln 2)/(\ln \Phi) =$

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$ , d. h.

$$d(T) \leq (\log_{\Phi} 2) \cdot \log_2 n.$$

Es gilt:  $\log_{\Phi} 2 = (\ln 2)/(\ln \Phi) = 1,440420 \dots < 1,4405$ .

---

Nun sei  $T$  ein höhenbalancierter Baum mit Höhe  $d(T) \geq 0$  und  $n$  Knoten.

Nach Behauptung 2:  $\Phi^{d(T)} \leq n$ .

**Logarithmieren** liefert  $d(T) \leq \log_{\Phi} n$ , d. h.

$$d(T) \leq (\log_{\Phi} 2) \cdot \log_2 n.$$

Es gilt:  $\log_{\Phi} 2 = (\ln 2)/(\ln \Phi) = 1,440420 \dots < 1,4405$ .

Damit ist Satz 4.3.3 bewiesen. □

---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

**Müssen noch zeigen:**

Man kann die Wörterbuchoperationen so implementieren,

---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

**Müssen noch zeigen:**

Man kann die Wörterbuchoperationen so implementieren,  
dass die AVL-Eigenschaft erhalten bleibt,

---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

**Müssen noch zeigen:**

Man kann die Wörterbuchoperationen so implementieren,  
dass die AVL-Eigenschaft erhalten bleibt,  
und der Zeitbedarf proportional zur Tiefe ist.

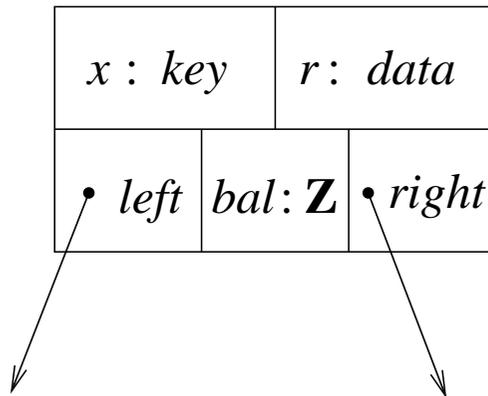
---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

**Müssen noch zeigen:**

Man kann die Wörterbuchoperationen so implementieren,  
dass die AVL-Eigenschaft erhalten bleibt,  
und der Zeitbedarf proportional zur Tiefe ist.

Knotenformat:



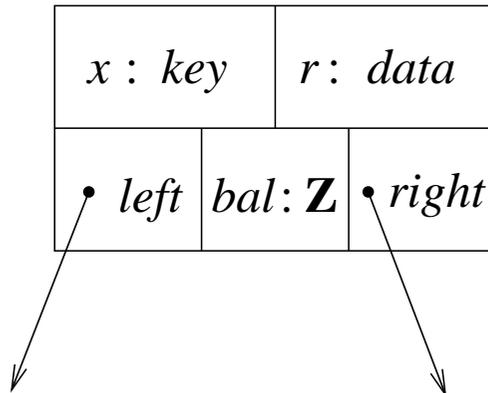
---

AVL-Bäume: höhenbalancierte binäre Suchbäume.  
Wissen: haben logarithmische Tiefe.

### Müssen noch zeigen:

Man kann die Wörterbuchoperationen so implementieren,  
dass die AVL-Eigenschaft erhalten bleibt,  
und der Zeitbedarf proportional zur Tiefe ist.

Knotenformat:



```
x: key;  
r: data;  
bal: integer; // Legal: {-1, 0, 1}  
left, right: AVL_Tree  
// Zeiger auf Baumknoten
```

**Vorlesungsvideo:**

# **AVL Rotationen**

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

Brauchen nur:

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

Brauchen nur: **AVL\_insert**( $T, x, r$ ) und **AVL\_delete**( $T, x$ ).

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

Brauchen nur: **AVL\_insert**( $T, x, r$ ) und **AVL\_delete**( $T, x$ ).

Grundansatz: Führe Operationen aus wie bei gewöhnlichem BSB.

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

Brauchen nur: **AVL\_insert**( $T, x, r$ ) und **AVL\_delete**( $T, x$ ).

Grundansatz: Führe Operationen aus wie bei gewöhnlichem BSB.

Eventuell wird dadurch Bedingung „Höhenbalancierung“ verletzt.

---

## 4.3.2 Implementierung der Operationen bei AVL-Bäumen

**AVL\_empty**: Erzeuge *NULL*-Zeiger. (Wie bei gewöhnlichem BSB.)

**AVL\_lookup**: Wie bei gewöhnlichem BSB.

Brauchen nur: **AVL\_insert**( $T, x, r$ ) und **AVL\_delete**( $T, x$ ).

Grundansatz: Führe Operationen aus wie bei gewöhnlichem BSB.

Eventuell wird dadurch Bedingung „Höhenbalancierung“ verletzt.

„Reparatur“: **Rebalancierung**, erfolgt rekursiv.

---

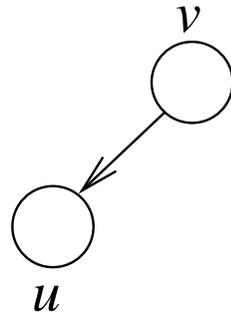
# **Rotationen: Hilfsoperationen für Rebalancierung**

**Rechtsrotation:** „kippe 1 Kante nach rechts“:

---

## Rotationen: Hilfsoperationen für Rebalancierung

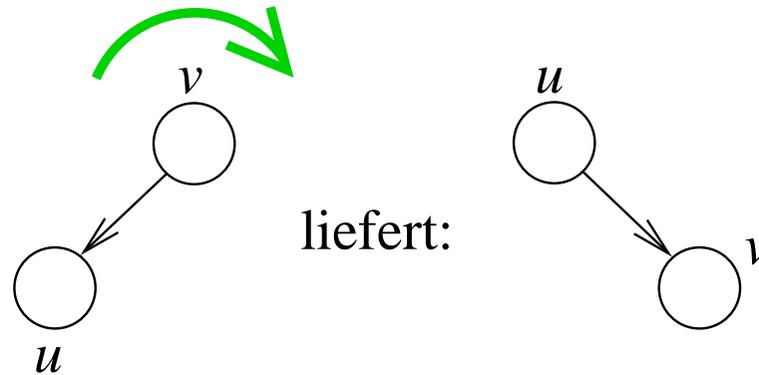
**Rechtsrotation:** „kippe 1 Kante nach rechts“:



---

# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** „kippe 1 Kante nach rechts“:



---

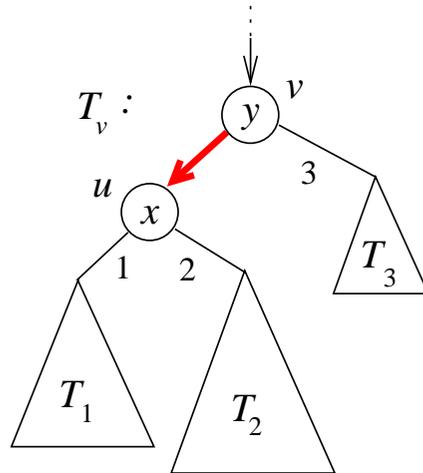
# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:

---

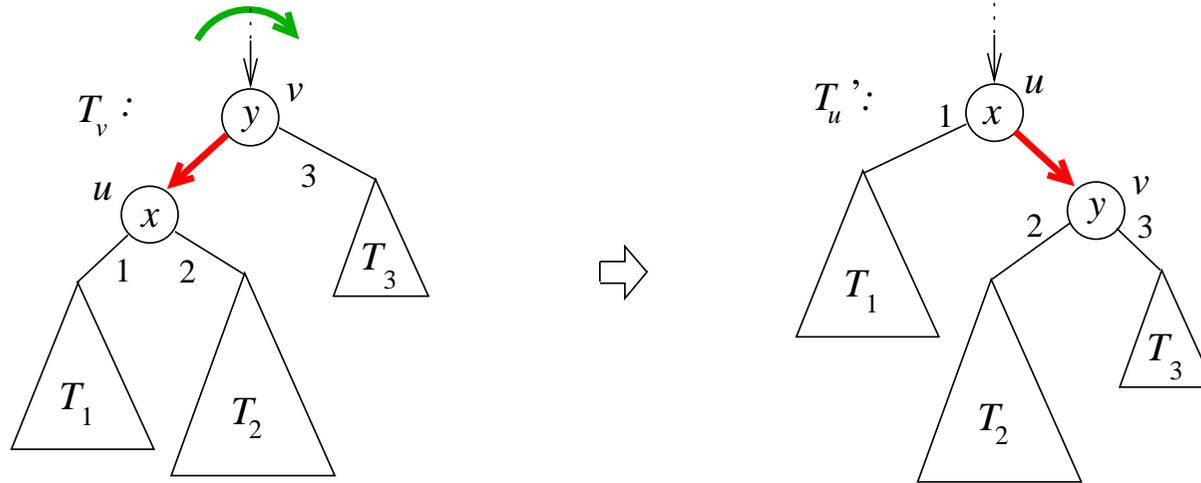
# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:



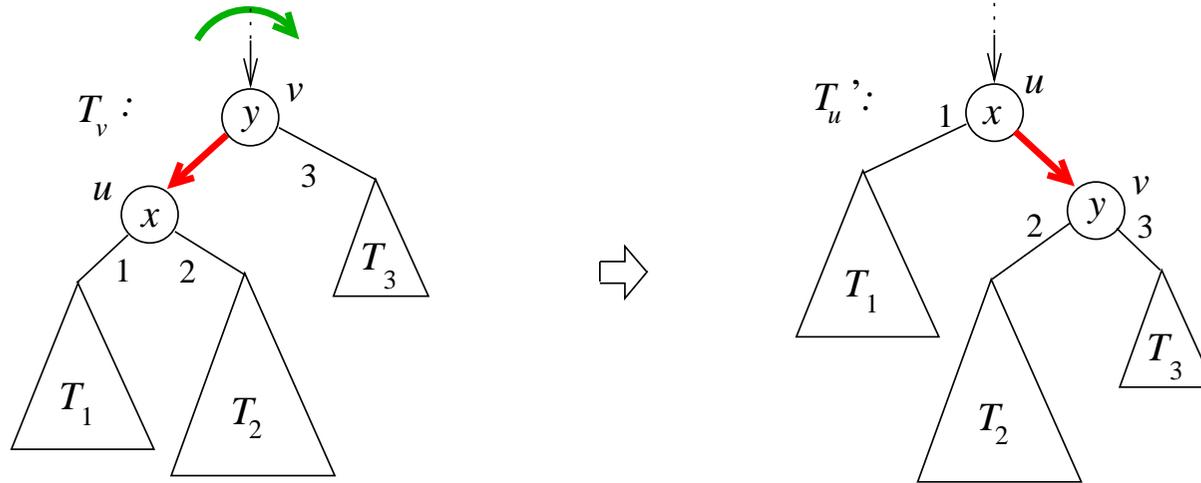
# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:



# Rotationen: Hilfsoperationen für Rebalancierung

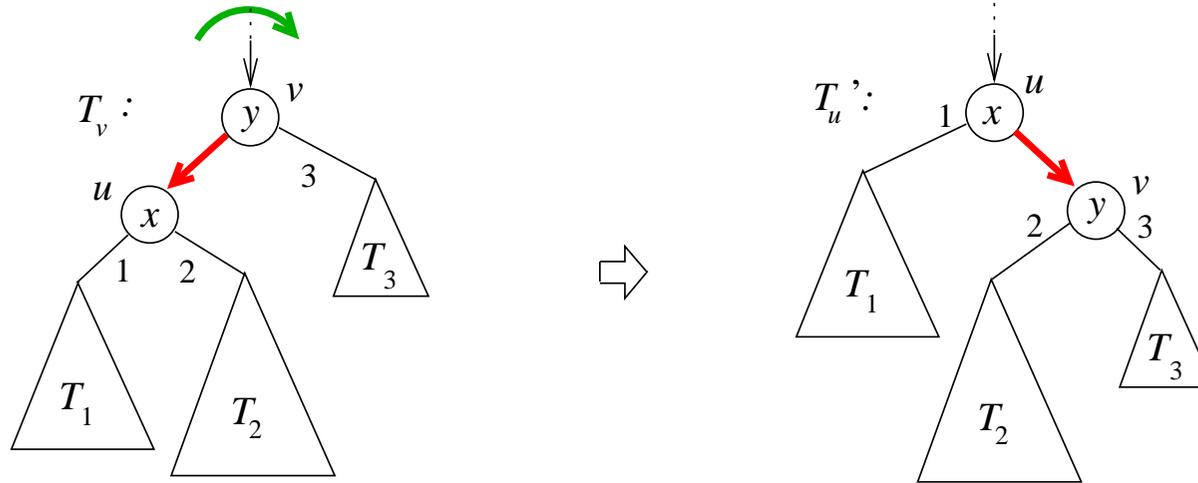
**Rechtsrotation:** Mit Unterbäumen:



**Beobachte:** Knotenmengen von  $T_v$  und  $T_u'$  sind gleich  $\wedge T_u'$  ist binärer Suchbaum.

# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:

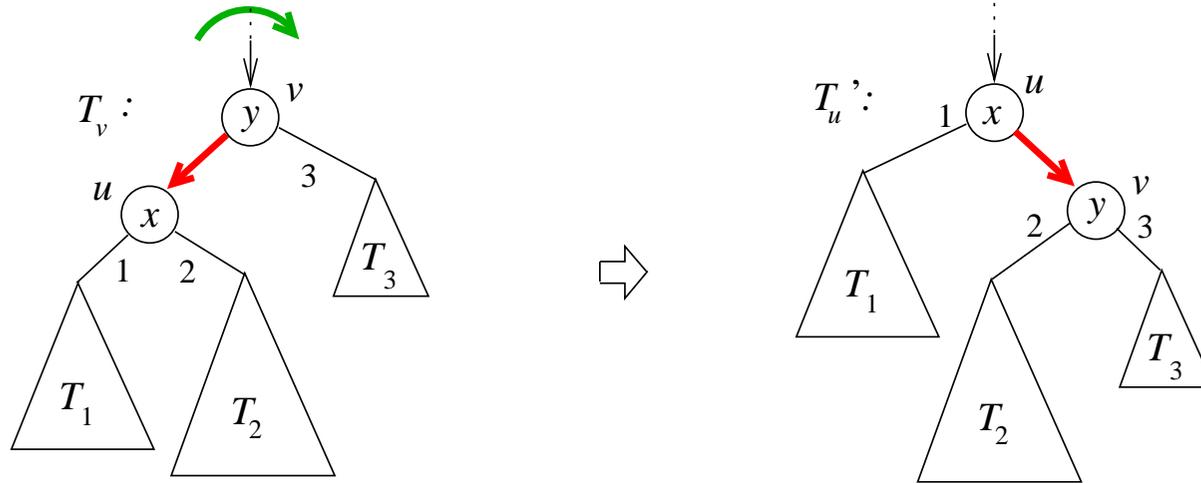


**Beobachte:** Knotenmengen von  $T_v$  und  $T'_u$  sind gleich  $\wedge T'_u$  ist binärer Suchbaum.

Denn: Knotenmengen sind klar. Weil  $T_v$  binärer Suchbaum ist,

# Rotationen: Hilfsoperationen für Rebalancierung

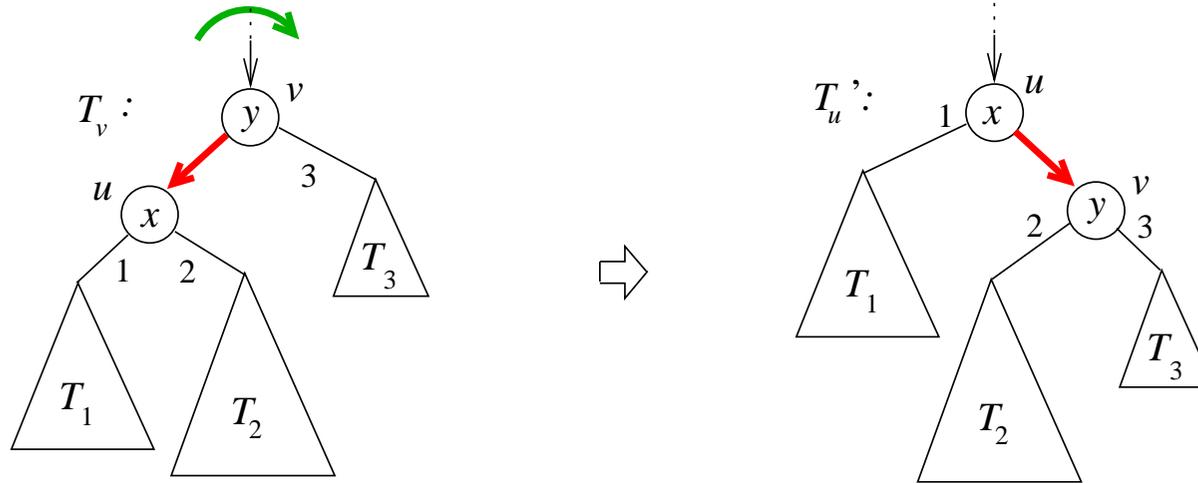
**Rechtsrotation:** Mit Unterbäumen:



**Beobachte:** Knotenmengen von  $T_v$  und  $T_u'$  sind gleich  $\wedge T_u'$  ist binärer Suchbaum.  
Denn: Knotenmengen sind klar. Weil  $T_v$  binärer Suchbaum ist, gilt für  $v_1$  in  $T_1$ ,  $v_2$  in  $T_2$ ,  $v_3$  in  $T_3$ :

# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:

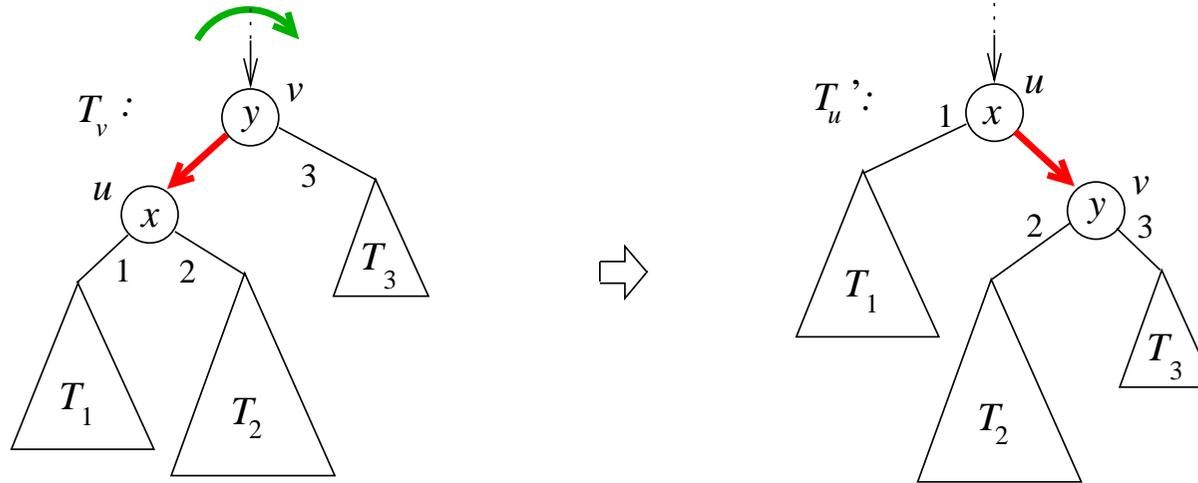


**Beobachte:** Knotenmengen von  $T_v$  und  $T'_u$  sind gleich  $\wedge T'_u$  ist binärer Suchbaum.

Denn: Knotenmengen sind klar. Weil  $T_v$  binärer Suchbaum ist, gilt für  $v_1$  in  $T_1$ ,  $v_2$  in  $T_2$ ,  $v_3$  in  $T_3$ :  $key(v_1) < x < key(v_2) < y < key(v_3)$ .

# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:

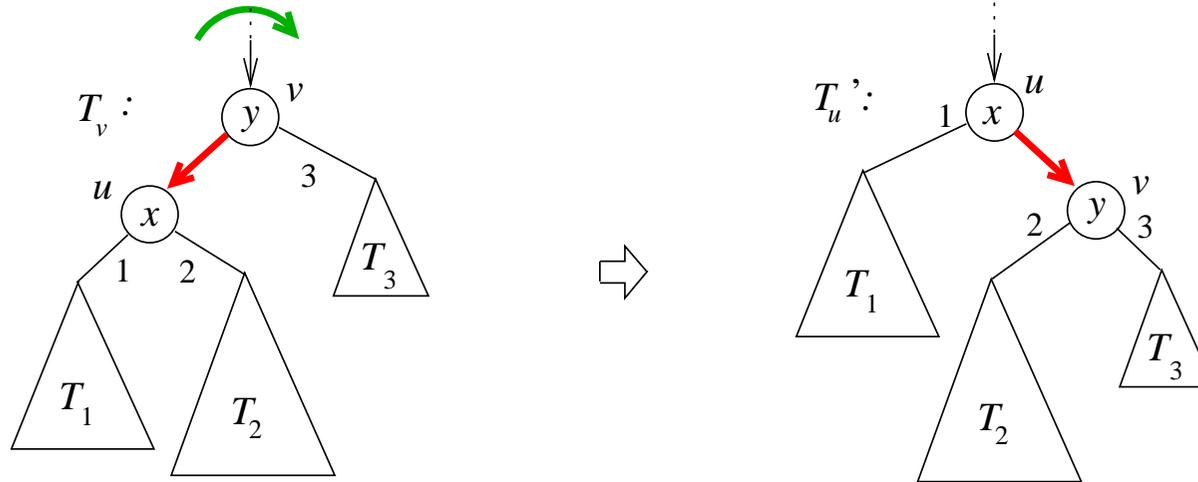


**Beobachte:** Knotenmengen von  $T_v$  und  $T'_u$  sind gleich  $\wedge T'_u$  ist binärer Suchbaum.

Denn: Knotenmengen sind klar. Weil  $T_v$  binärer Suchbaum ist, gilt für  $v_1$  in  $T_1$ ,  $v_2$  in  $T_2$ ,  $v_3$  in  $T_3$ :  $key(v_1) < x < key(v_2) < y < key(v_3)$ . Also ist  $T'_u$  auch BSB.

# Rotationen: Hilfsoperationen für Rebalancierung

**Rechtsrotation:** Mit Unterbäumen:



Beim Umbau ändert sich nur:

- rechtes Kind von  $u$
- linkes Kind von  $v$
- Zeiger auf Wurzelknoten (von  $v$  auf  $u$  umsetzen)

---

# Rotationen: Hilfsoperationen für Rebalancierung

## **Rechtsrotation als Programm:**

---

## Rotationen: Hilfsoperationen für Rebalancierung

### Rechtsrotation als Programm:

**Funktion rotateR**(v : AVL\_Tree): AVL\_Tree

- (1) // v  $\neq$   $\square$ , v.left  $\neq$   $\square$
- (2) u : AVL\_Tree ; // Hilfsvariable
- (3) u  $\leftarrow$  v.left ;
- (4) v.left  $\leftarrow$  u.right ;
- (5) u.right  $\leftarrow$  v ;
- (6) **return** u
- (7) // **!! Balancefaktoren in u und v sind falsch!**

---

# Rotationen: Hilfsoperationen für Rebalancierung

**Linksrotation:** Umkehrung von Rechtsrotation

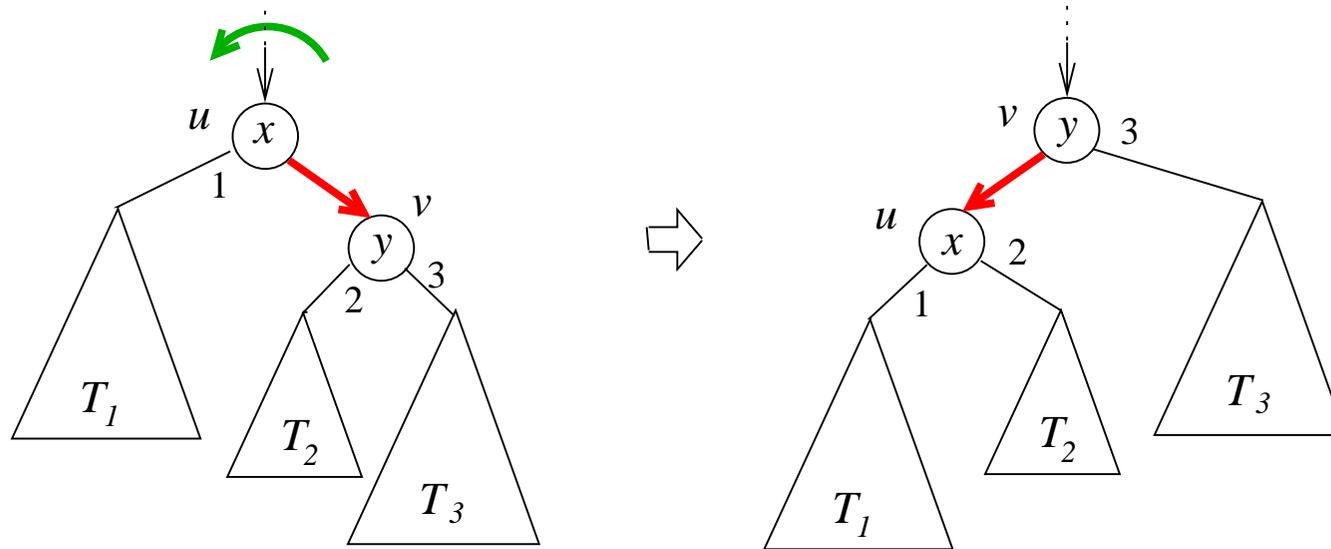
---

## Rotationen: Hilfsoperationen für Rebalancierung

**Linksrotation:** Umkehrung von Rechtsrotation, Implementierung analog.

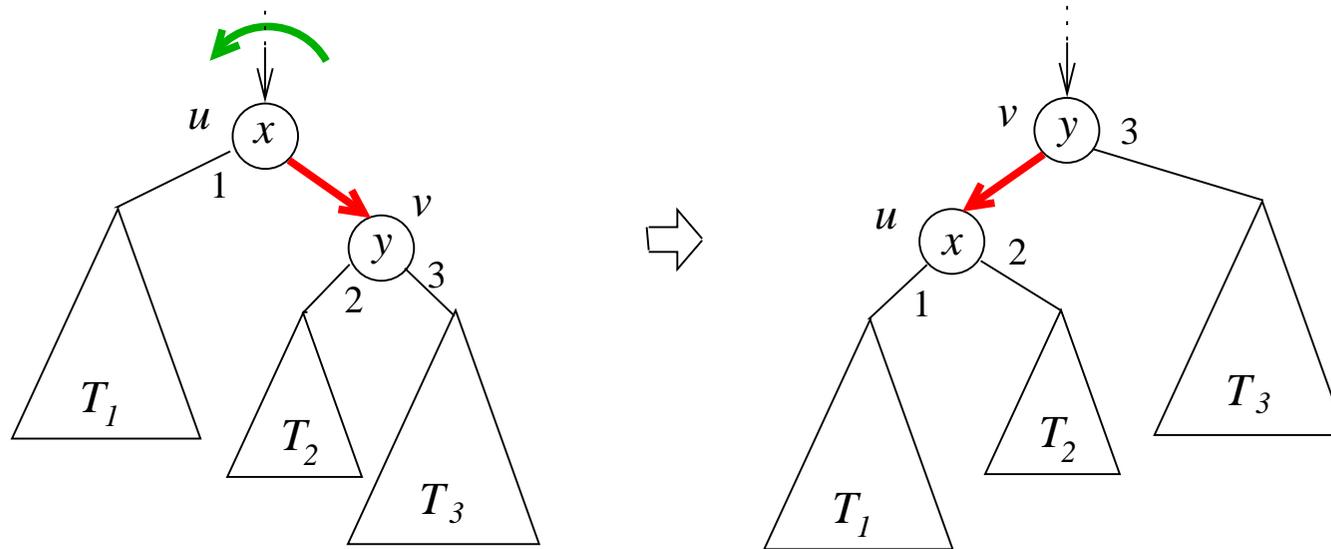
# Rotationen: Hilfsoperationen für Rebalancierung

**Linksrotation:** Umkehrung von Rechtsrotation, Implementierung analog.



# Rotationen: Hilfsoperationen für Rebalancierung

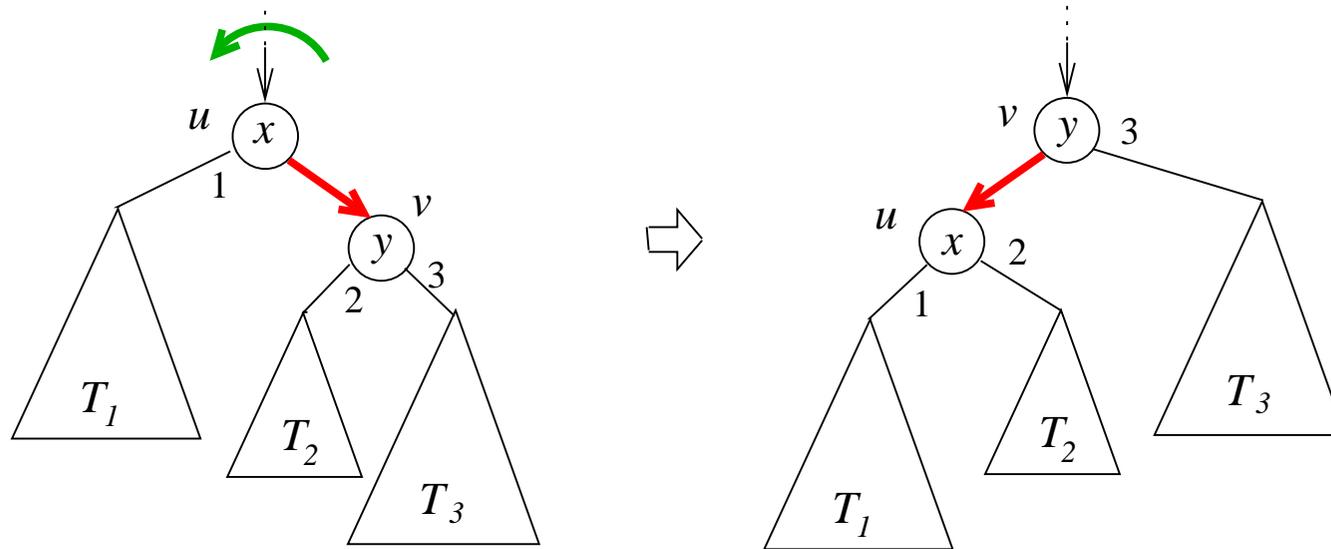
**Linksrotation:** Umkehrung von Rechtsrotation, Implementierung analog.



Hier und in den folgenden *Beispielen*:

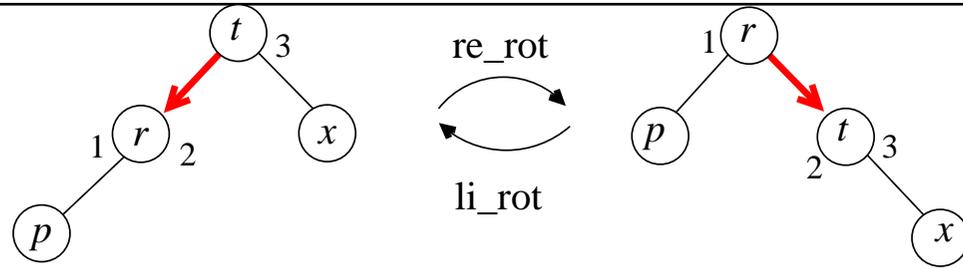
# Rotationen: Hilfsoperationen für Rebalancierung

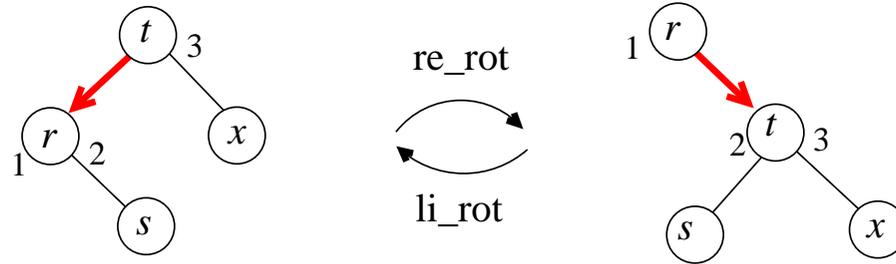
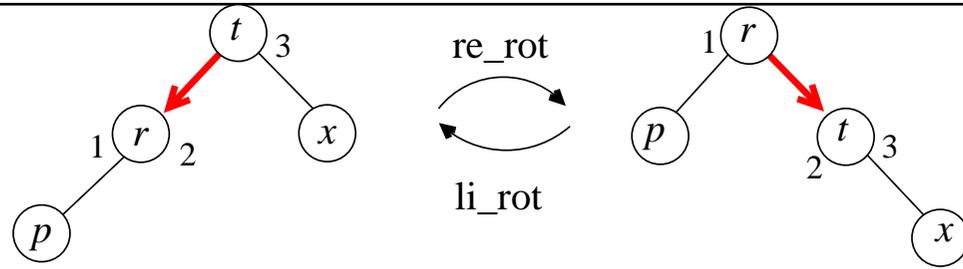
**Linksrotation:** Umkehrung von Rechtsrotation, Implementierung analog.

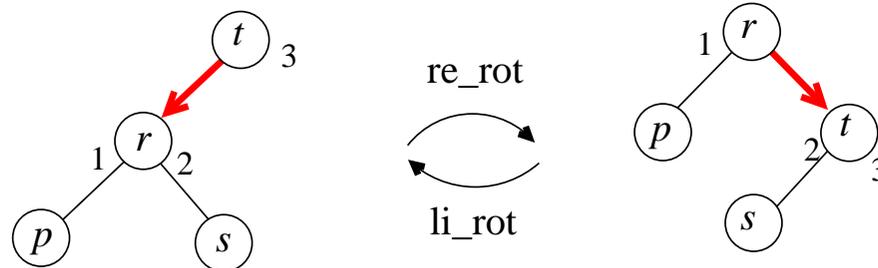
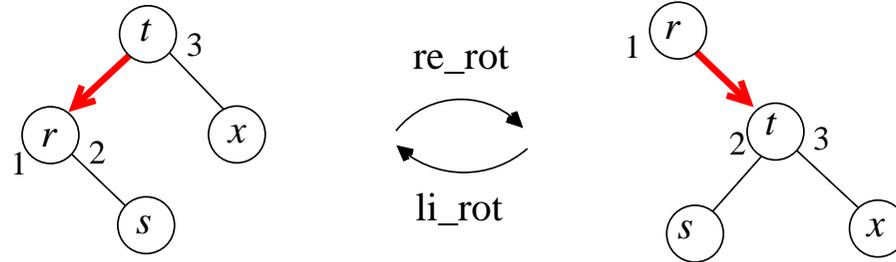
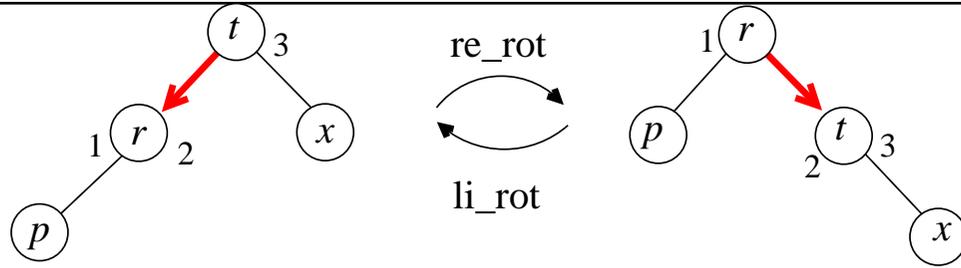


Hier und in den folgenden *Beispielen*:

Nummern bezeichnen Anschlussstellen für die Unterbäume.







---

## Doppelrotationen: Links-Rechts, Rechts-Links

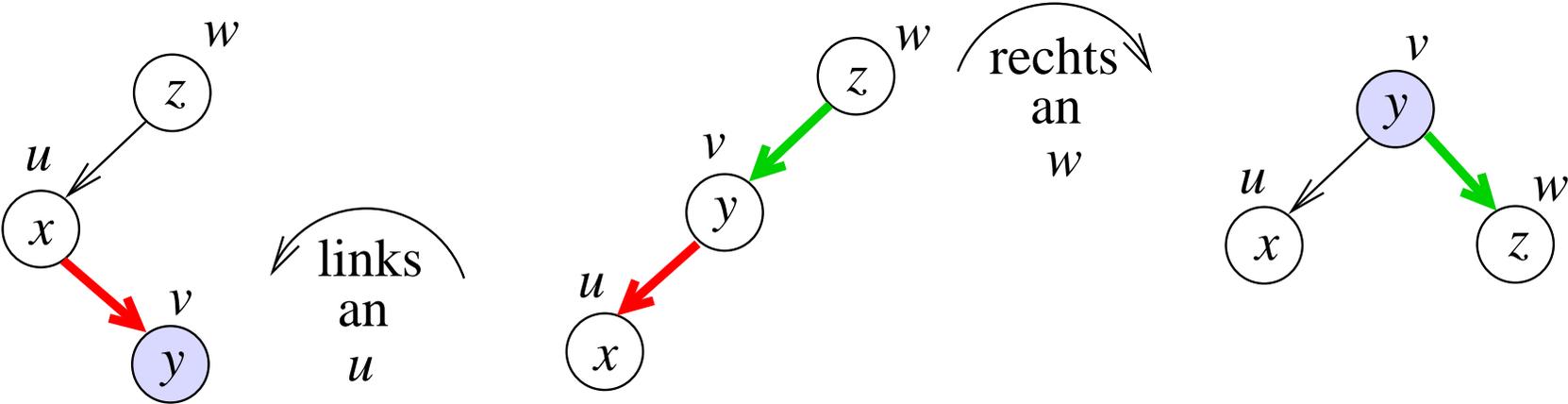
---

## Doppelrotationen: Links-Rechts, Rechts-Links

*Beispiel:* **Links-Rechts-Doppelrotation**

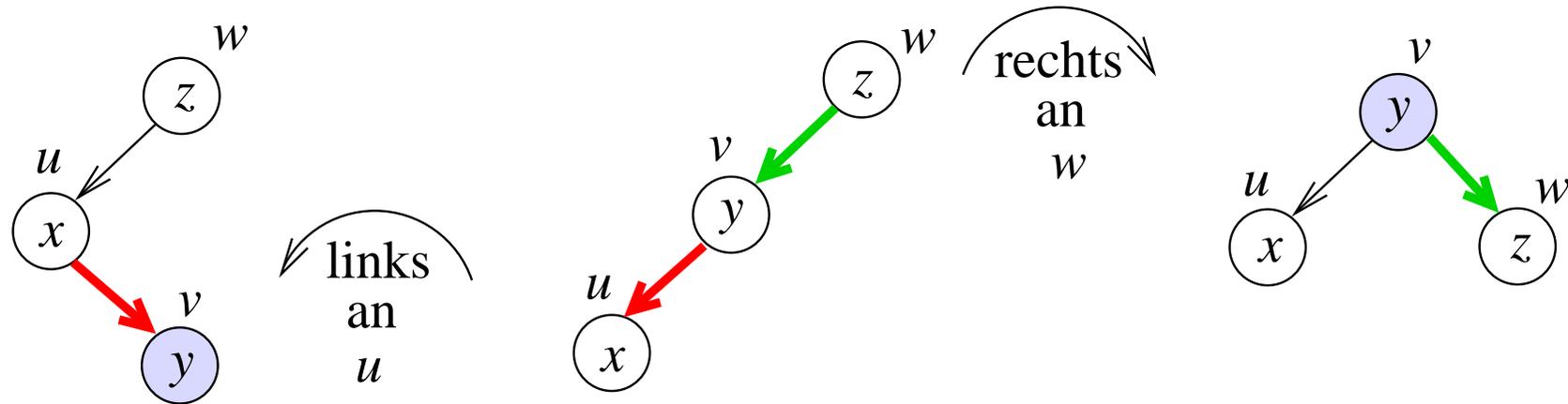
# Doppelrotationen: Links-Rechts, Rechts-Links

## Beispiel: Links-Rechts-Doppelrotation



## Doppelrotationen: Links-Rechts, Rechts-Links

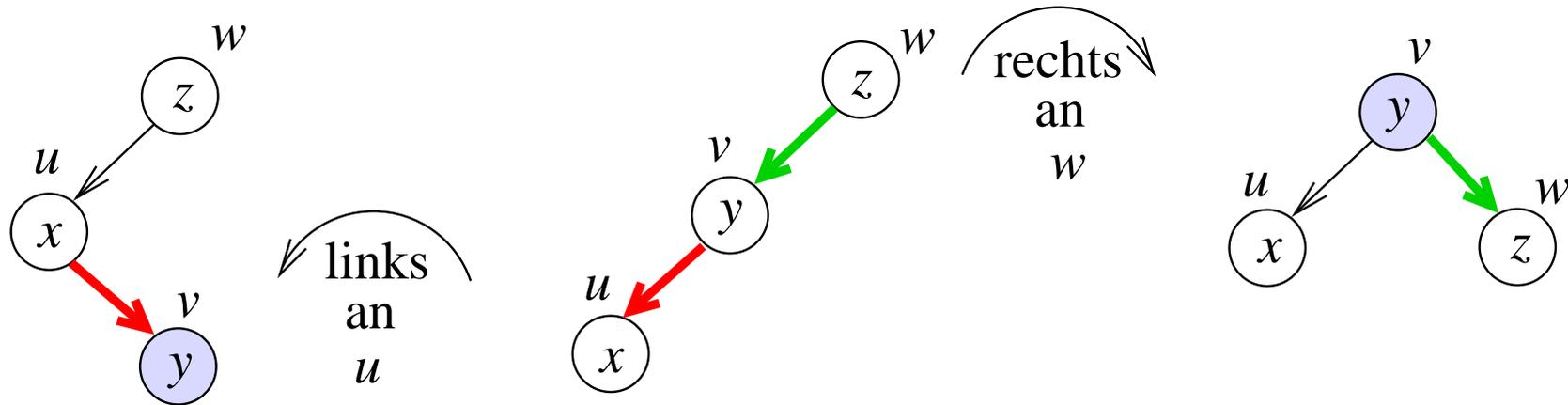
### Beispiel: Links-Rechts-Doppelrotation



Anzuwenden auf Zick-Zack-Weg aus zwei Kanten, Form „links-rechts“.

## Doppelrotationen: Links-Rechts, Rechts-Links

### Beispiel: Links-Rechts-Doppelrotation



Anzuwenden auf Zick-Zack-Weg aus zwei Kanten, Form „**links-rechts**“.

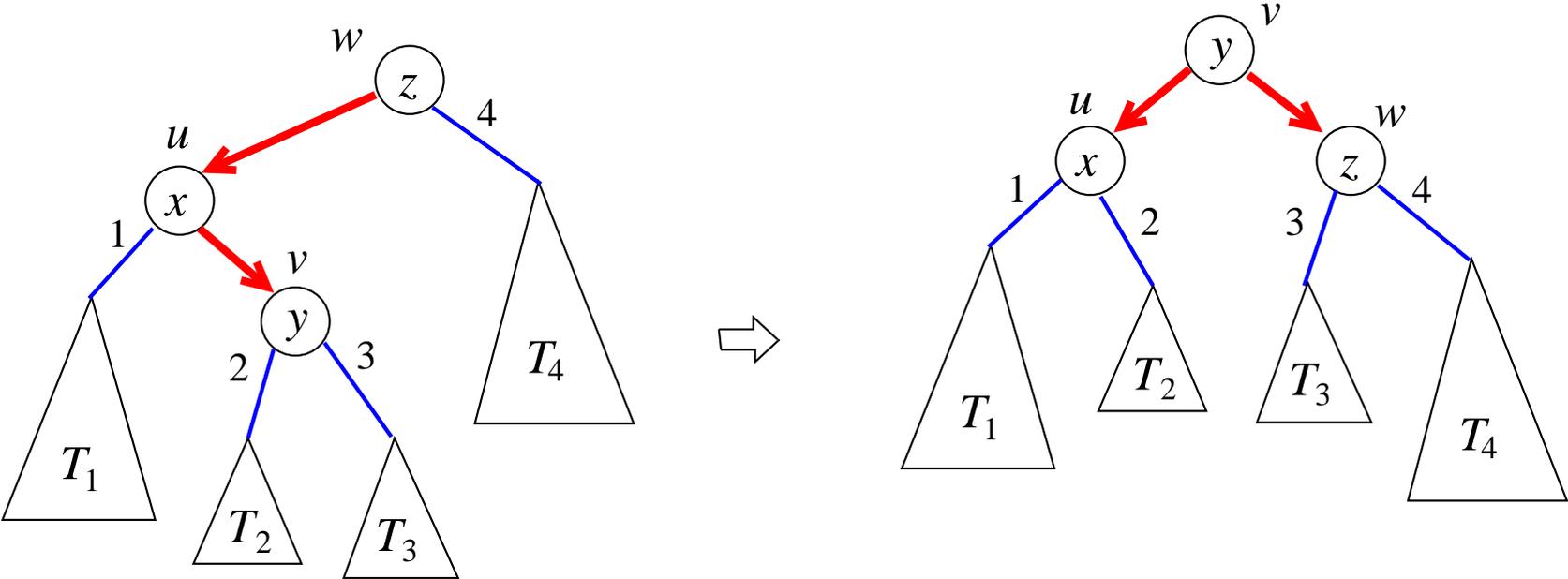
Erst eine **Links**rotation, dann eine **Rechts**rotation.

Effekt: **Tiefster** Knoten  $v$  wird Wurzel.

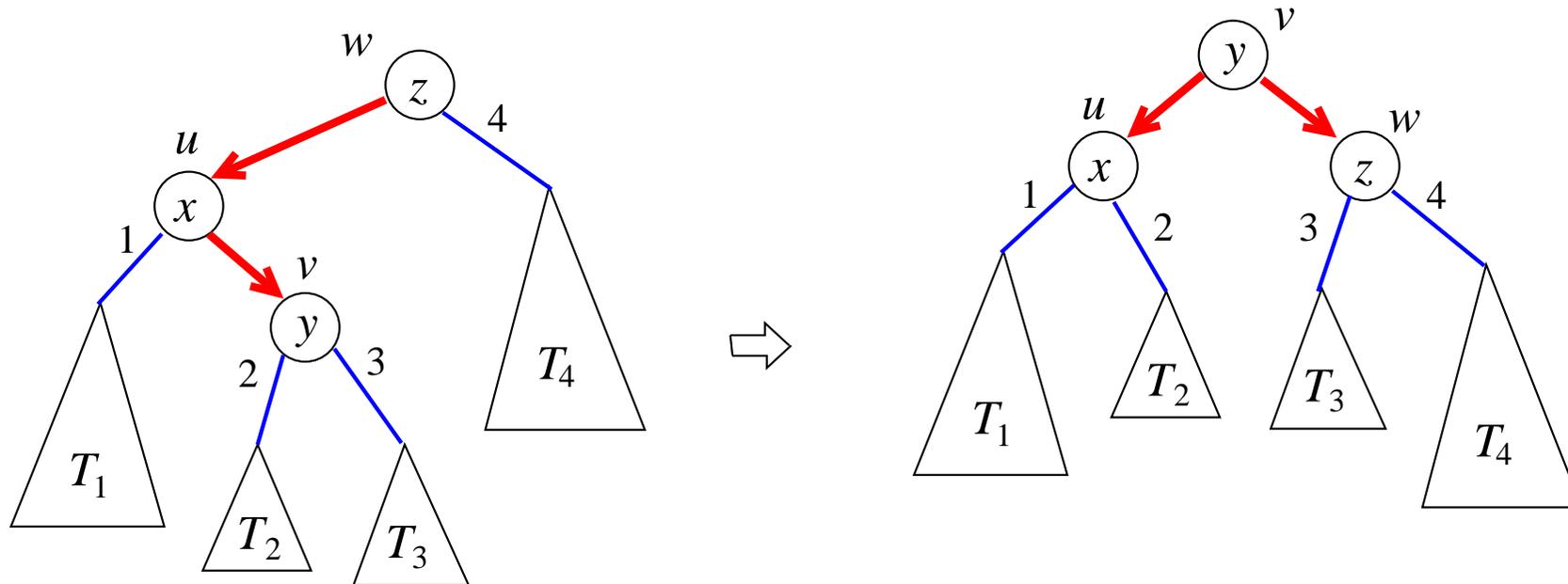
---

## Links-Rechts-Doppelrotation: Mit Unterbäumen

# Links-Rechts-Doppelrotation: Mit Unterbäumen

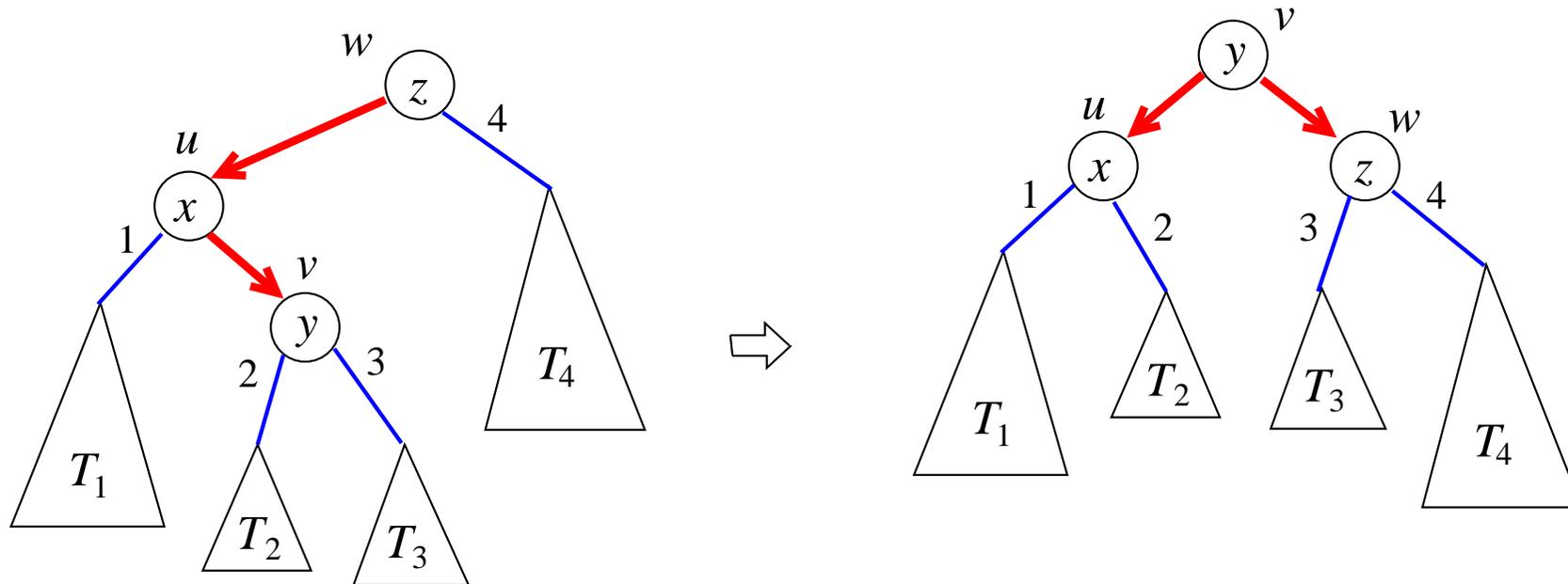


## Links-Rechts-Doppelrotation: Mit Unterbäumen



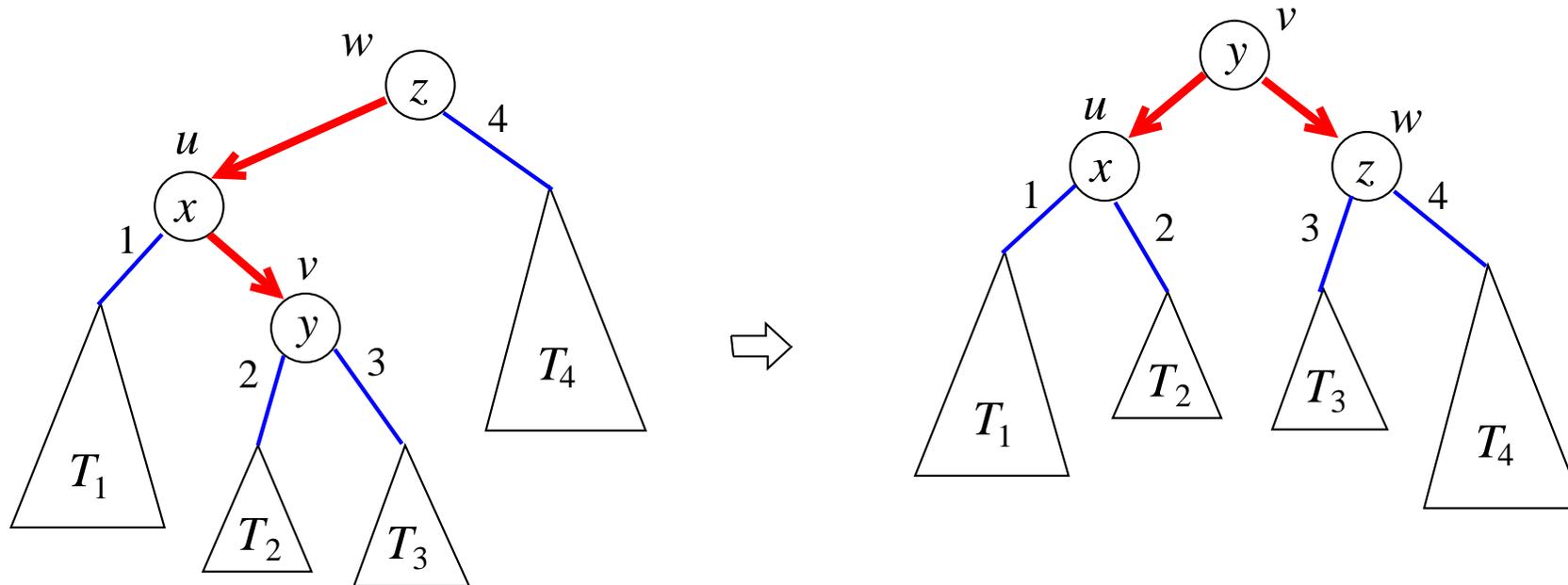
**Gesamteffekt:** Der **unterste** Knoten  $v$  des Zick-Zack-Wegs („links-rechts“) wandert nach oben, wird **Wurzel**;

## Links-Rechts-Doppelrotation: Mit Unterbäumen



**Gesamteffekt:** Der **unterste** Knoten  $v$  des Zick-Zack-Wegs („links-rechts“) wandert nach oben, wird **Wurzel**; die beiden anderen Knoten  $u$  und  $w$  werden linkes und rechtes Kind.

## Links-Rechts-Doppelrotation: Mit Unterbäumen



**Gesamteffekt:** Der **unterste** Knoten  $v$  des Zick-Zack-Wegs („links-rechts“) wandert nach oben, wird **Wurzel**; die beiden anderen Knoten  $u$  und  $w$  werden linkes und rechtes Kind. (Vier Teilbäume werden umgehängt, zwei bleiben gleich.)

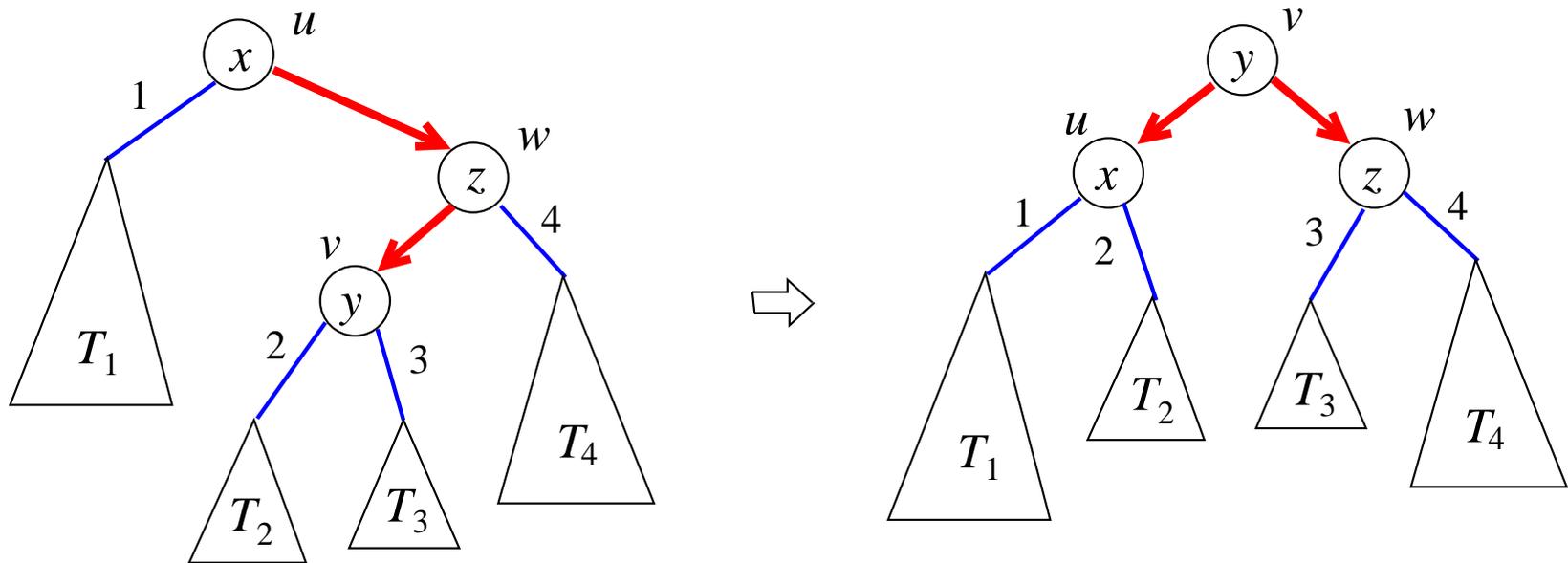
---

## LR-Doppelrotation als Programm

**Funktion rotateLR**( $w : \text{AVL\_Tree}$ ) :  $\text{AVL\_Tree}$

- (1) // Eingabe:  $w \neq \square$  mit  $w.\text{left} \neq \square$  und  $w.\text{left}.\text{right} \neq \square$
- (2)  $u, v : \text{AVL\_Tree}$  ;
- (3)  $u \leftarrow w.\text{left}$  ;
- (4)  $v \leftarrow u.\text{right}$  ;
- (5)  $w.\text{left} \leftarrow v.\text{right}$  ;
- (6)  $u.\text{right} \leftarrow v.\text{left}$  ;
- (7)  $v.\text{left} \leftarrow u$  ;
- (8)  $v.\text{right} \leftarrow w$  ;
- (9) **return**  $v$
- (10) // **!! Balancefaktoren in  $u, v, w$  sind falsch!**
- ⋮     ⋮

## Rechts-Links-Doppelrotation: Symmetrisch zu Links-Rechts-Doppelrotation



Implementierung: Analog zu Links-Rechts-Doppelrotation.

**Vorlesungsvideo:**  
**AVL\_Insert**

---

**AVL\_insert( $T, x, r$ )**

---

**AVL\_insert( $T, x, r$ )**

**Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).

---

**AVL\_insert( $T, x, r$ )**

**Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg** **von unten nach oben** ab, kontrolliere Balancebedingung.

---

**AVL\_insert( $T, x, r$ )**

**Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.

---

**AVL\_insert**( $T, x, r$ )

### **Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  $v :=$  **tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .

---

**AVL\_insert**( $T, x, r$ )

### **Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .  
An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)

---

**AVL\_insert( $T, x, r$ )**

**Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .  
An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)  
3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) zu tief ist:

---

**AVL\_insert( $T, x, r$ )**

**Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .  
An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)  
3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) zu tief ist:  
**Einfache Rotation** hebt diesen Teilbaum ein Level höher.

---

**AVL\_insert( $T, x, r$ )**

### **Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .  
An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)
  - 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) zu tief ist:  
**Einfache Rotation** hebt diesen Teilbaum ein Level höher.
  - 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) zu tief ist:

---

**AVL\_insert( $T, x, r$ )**

### **Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg** **von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.
3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .  
An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)
  - 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) zu tief ist:  
**Einfache Rotation** hebt diesen Teilbaum ein Level höher.
  - 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) zu tief ist:  
**Doppelrotation** hebt den mittleren Teilbaum ein Level höher.

---

**AVL\_insert( $T, x, r$ )**

### **Anschauliche Beschreibung:**

1. Füge ein wie bei gewöhnlichem BSB. Dies erzeugt neuen Knoten (außer Update).
2. Laufe dann **Einfügeweg** **von unten nach oben** ab, kontrolliere Balancebedingung. Wenn diese nirgendwo verletzt ist: fertig.

3. Sonst:  **$v :=$  tiefster Knoten** auf Einfügeweg mit  $bal(v) \in \{-2, 2\}$ .

An  $v$ : Einfach- oder Doppelrotation. (Dann Rebalancierung beendet.)

3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) zu tief ist:  
**Einfache Rotation** hebt diesen Teilbaum ein Level höher.

3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) zu tief ist:  
**Doppelrotation** hebt den mittleren Teilbaum ein Level höher.

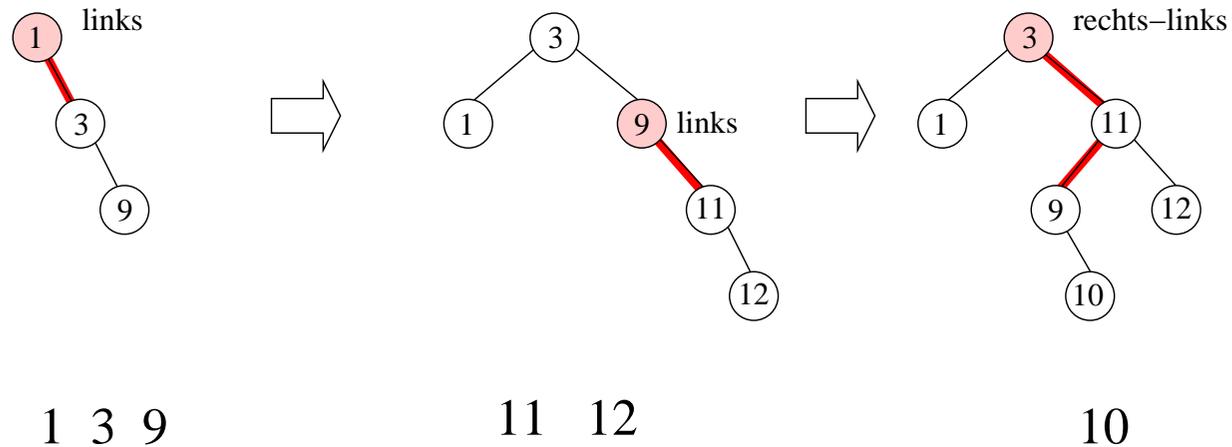
Zum Ausprobieren:

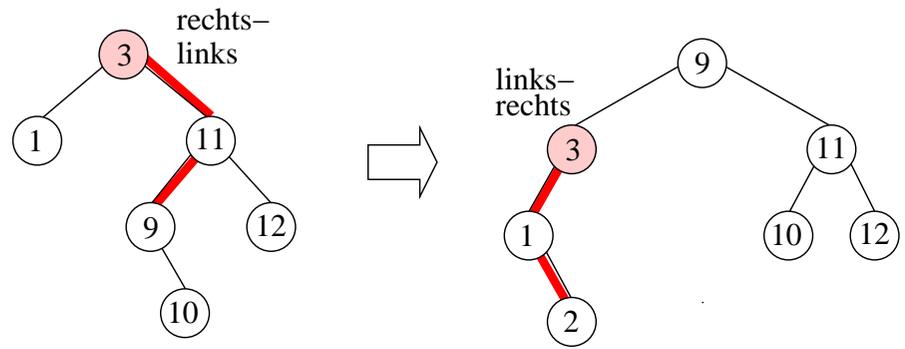
<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

---

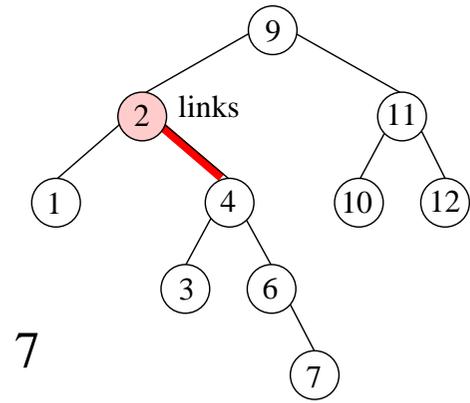
*Beispiel:* Einfügen von 1, 3, 9, 11, 12, 10, 2, 6, 4, 7, 0 in anfangs leeren AVL-Baum. Wir zeichnen nur Situationen, in denen **Rotationen stattfinden**. Der tiefste Knoten, an dem die Balancebedingung verletzt ist, ist **rot** gezeichnet.

*Beispiel:* Einfügen von 1, 3, 9, 11, 12, 10, 2, 6, 4, 7, 0 in anfangs leeren AVL-Baum. Wir zeichnen nur Situationen, in denen **Rotationen stattfinden**. Der tiefste Knoten, an dem die Balancebedingung verletzt ist, ist **rot** gezeichnet.

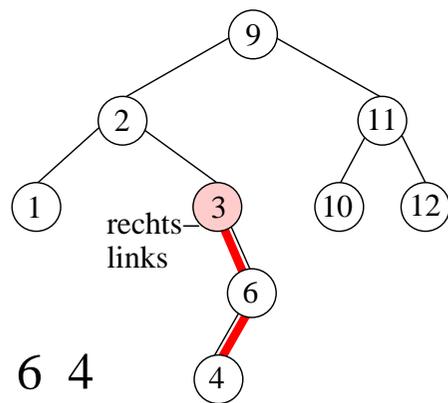




2

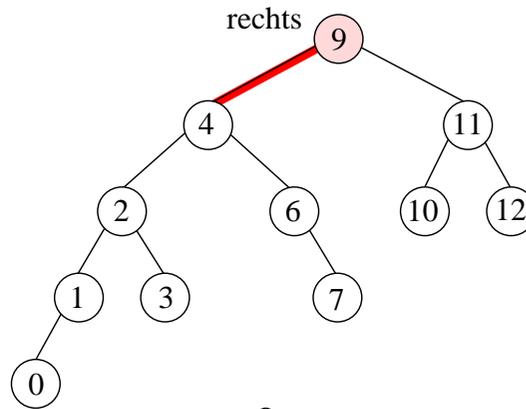
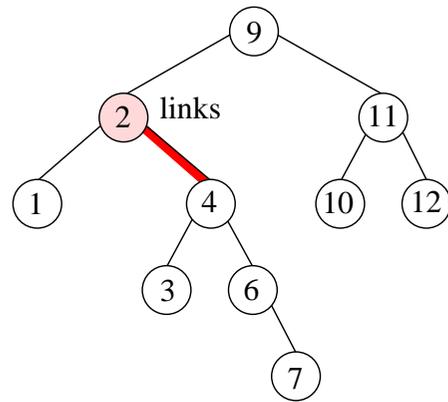


7

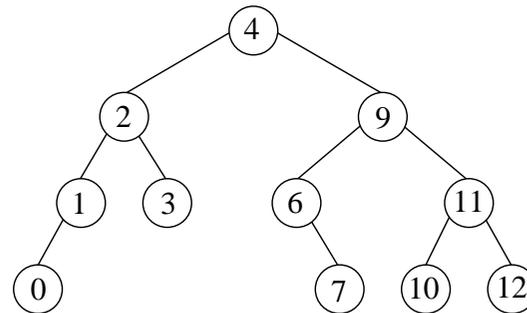


6 4





0



---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

**Benutzt werden:**

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

### **Benutzt werden:**

- Balancefaktoren  $T_{bal}$  in den Knoten,

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

### Benutzt werden:

- Balancefaktoren  $T.bal$  in den Knoten,
- Flagbit „deeper“, das als Resultat eines rekursiven Aufrufs **AVL\_insert**( $T, x, r$ ) mitteilt, ob der Baum dadurch **tiefer** geworden ist.

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

### Benutzt werden:

- Balancefaktoren  $T.bal$  in den Knoten,
- Flagbit „deeper“, das als Resultat eines rekursiven Aufrufs **AVL\_insert**( $T, x, r$ ) mitteilt, ob der Baum dadurch **tiefer** geworden ist.

Wenn **AVL\_insert**( $T', x, r$ ) für einen Unterbaum  $T'$  von  $T$  aufgerufen wurde, gibt es zur Rebalancierung an der Wurzel von  $T$  eine große Fallunterscheidung, gesteuert durch (alte) Balancefaktoren und die „deeper“-Meldung aus dem rekursivem Aufruf.

---

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Programm kann nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen, weder durch „Hinschauen“ noch rechnerisch (Tiefen von Teilbäumen werden nicht mitgeführt!).

### Benutzt werden:

- Balancefaktoren  $T.bal$  in den Knoten,
- Flagbit „deeper“, das als Resultat eines rekursiven Aufrufs **AVL\_insert**( $T, x, r$ ) mitteilt, ob der Baum dadurch **tiefer** geworden ist.

Wenn **AVL\_insert**( $T', x, r$ ) für einen Unterbaum  $T'$  von  $T$  aufgerufen wurde, gibt es zur Rebalancierung an der Wurzel von  $T$  eine große Fallunterscheidung, gesteuert durch (alte) Balancefaktoren und die „deeper“-Meldung aus dem rekursivem Aufruf.

Wir entwickeln das Programm systematisch durch Erweitern der gewöhnlichen BSB-Einfügung.

---

**Funktion AVL\_insert**(T,x,r): (AVL\_Tree,boolean)

(1) // **Eingabe:** T: AVL\_Tree, x: key, r: data

(2) // **Ausgabe:** T: AVL\_Tree, deeper: boolean

(3) **1. Fall:** T =  $\square$

(4) T: new AVL\_Tree // Erzeuge neuen AVL-Baum-Knoten

(5) T.key  $\leftarrow$  x ;

(6) T.data  $\leftarrow$  r ;

(7) T.left  $\leftarrow$   $\square$ ;

(8) T.right  $\leftarrow$   $\square$ ;

(9) T.bal  $\leftarrow$  0 ;

(10) **return** (T, true) .

(11) // Baumhöhe hat sich von  $-1$  auf 0 erhöht.

---

(12) **2. Fall:**  $T \neq \square$

- 
- (12) **2. Fall:**  $T \neq \square$  and  $T.key = x$ .
  - (13) // Update-Situation!
  - (14)  $T.data \leftarrow r$  ;
  - (15) **return**  $(T, false)$  .
  - (16) // Baumstruktur hat sich nicht verändert.

---

(12) **2. Fall:**  $T \neq \square$  and  $T.\text{key} = x$ .

(13) // **Update-Situation!**

(14)  $T.\text{data} \leftarrow r$  ;

(15) **return**  $(T, \text{false})$  .

(16) // Baumstruktur hat sich nicht verändert.

(17) **3. Fall:**  $T \neq \square$

---

(12) **2. Fall:**  $T \neq \square$  and  $T.key = x$ .

(13) // **Update-Situation!**

(14)  $T.data \leftarrow r$  ;

(15) **return**  $(T, false)$  .

(16) // Baumstruktur hat sich nicht verändert.

(17) **3. Fall:**  $T \neq \square$  and  $x < T.key$ .

(18)  $(T.left, left\_deeper) \leftarrow \mathbf{AVL\_insert}(T.left, x, r)$  ;

(19) // Rekursives Einfügen in linken Unterbaum, liefert AVL-Baum und Flagbit

- 
- (12) **2. Fall:**  $T \neq \square$  and  $T.key = x$ .
- (13) // **Update-Situation!**
- (14)  $T.data \leftarrow r$  ;
- (15) **return**  $(T, false)$  .
- (16) // Baumstruktur hat sich nicht verändert.
- 
- (17) **3. Fall:**  $T \neq \square$  and  $x < T.key$ .
- (18)  $(T.left, left\_deeper) \leftarrow \mathbf{AVL\_insert}(T.left, x, r)$  ;
- (19) // Rekursives Einfügen in linken Unterbaum, liefert AVL-Baum und Flagbit
- (20)  $(T, deeper) \leftarrow \mathbf{RebalanceInsLeft}(T, left\_deeper)$  ;
- (21) // Rebalancierung in der Wurzel von T, falls nötig, **s. unten**

---

(12) **2. Fall:**  $T \neq \square$  and  $T.key = x$ .  
(13) // **Update-Situation!**  
(14)  $T.data \leftarrow r$  ;  
(15) **return**  $(T, false)$  .  
(16) // Baumstruktur hat sich nicht verändert.

(17) **3. Fall:**  $T \neq \square$  and  $x < T.key$ .  
(18)  $(T.left, left\_deeper) \leftarrow \mathbf{AVL\_insert}(T.left, x, r)$  ;  
(19) // Rekursives Einfügen in linken Unterbaum, liefert AVL-Baum und Flagbit  
(20)  $(T, deeper) \leftarrow \mathbf{RebalanceInsLeft}(T, left\_deeper)$  ;  
(21) // Rebalancierung in der Wurzel von T, falls nötig, **s. unten**  
(22) **return**  $(T, deeper)$  .

---

```
(23) 4. Fall:  $T \neq \square$  and  $T.\text{key} < x$ .
(24)    $(T.\text{right}, \text{right\_deeper}) \leftarrow \mathbf{AVL\_insert}(T.\text{right}, x, r)$  ;
(25)   // Rekursives Einfügen in rechten Unterbaum, liefert AVL-Baum und Flagbit
(26)    $(T, \text{deeper}) \leftarrow \mathbf{RebalanceInsRight}(T, \text{right\_deeper})$  ;
(27)   // Rebalancierung in der Wurzel von T, falls nötig,
        symmetrisch zum 3. Fall.
(28)   return (T, deeper) .
```

---

Zeile (18): **AVL\_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

---

Zeile (18): **AVL\_insert** wird rekursiv auf T.left, x, r angewendet.

**Ergebnis:** Unterbaum T.left geändert. Flagbit left\_deeper bedeutet:

left\_deeper = *false*: T.left hat gleiche Höhe wie vorher;

left\_deeper = *true*: T.left ist um 1 Level tiefer geworden.

---

Zeile (18): **AVL\_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

**Ergebnis:** Unterbaum `T.left` geändert. Flagbit `left_deeper` bedeutet:

`left_deeper = false`: `T.left` hat gleiche Höhe wie vorher;

`left_deeper = true`: `T.left` ist um 1 Level tiefer geworden.

Wir beweisen die Korrektheit durch Induktion über rekursive Aufrufe. Daher können wir als **I.V.** verwenden: `T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

---

Zeile (18): **AVL\_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

**Ergebnis:** Unterbaum `T.left` geändert. Flagbit `left_deeper` bedeutet:

`left_deeper = false`: `T.left` hat gleiche Höhe wie vorher;

`left_deeper = true`: `T.left` ist um 1 Level tiefer geworden.

Wir beweisen die Korrektheit durch Induktion über rekursive Aufrufe. Daher können wir als **I.V.** verwenden: `T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

Verbleibende **Probleme**:

- Ist die Balancebedingung in der Wurzel von `T` erfüllt?

---

Zeile (18): **AVL\_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

**Ergebnis:** Unterbaum `T.left` geändert. Flagbit `left_deeper` bedeutet:

`left_deeper = false`: `T.left` hat gleiche Höhe wie vorher;

`left_deeper = true`: `T.left` ist um 1 Level tiefer geworden.

Wir beweisen die Korrektheit durch Induktion über rekursive Aufrufe. Daher können wir als **I.V.** verwenden: `T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

Verbleibende **Probleme**:

- Ist die Balancebedingung in der Wurzel von `T` erfüllt?
- Der Balancefaktor `T.ba1` in der Wurzel von `T` könnte falsch sein.

---

Zeile (18): **AVL\_insert** wird rekursiv auf `T.left`, `x`, `r` angewendet.

**Ergebnis:** Unterbaum `T.left` geändert. Flagbit `left_deeper` bedeutet:

`left_deeper = false`: `T.left` hat gleiche Höhe wie vorher;

`left_deeper = true`: `T.left` ist um 1 Level tiefer geworden.

Wir beweisen die Korrektheit durch Induktion über rekursive Aufrufe. Daher können wir als **I.V.** verwenden: `T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

Verbleibende **Probleme**:

- Ist die Balancebedingung in der Wurzel von `T` erfüllt?
- Der Balancefaktor `T.ba1` in der Wurzel von `T` könnte falsch sein.

Zeile (20): **RebalanceInsLeft** prüft, ändert Struktur, korrigiert Balancefaktoren;

**Ergebnis:** Neuer AVL-Baum `T` und Flagbit `deeper`.

---

## RebalanceLeft(T, left\_deeper)

---

## RebalanceInsLeft(T, left\_deeper)

### Fall ReblL-1

**Aktion:**

left\_deeper = *false*

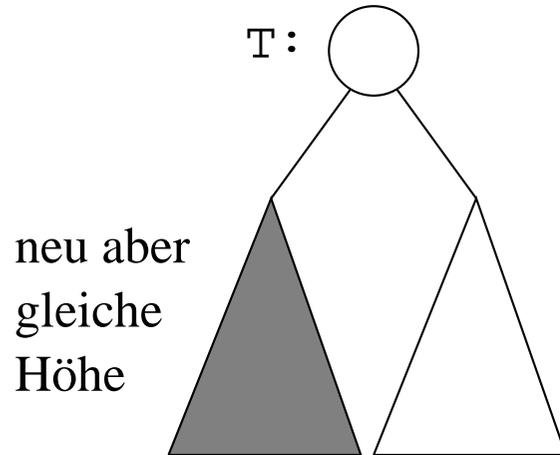
---

## RebalanceLeft(T, left\_deeper)

### Fall ReblL-1

Aktion:

left\_deeper = *false*



---

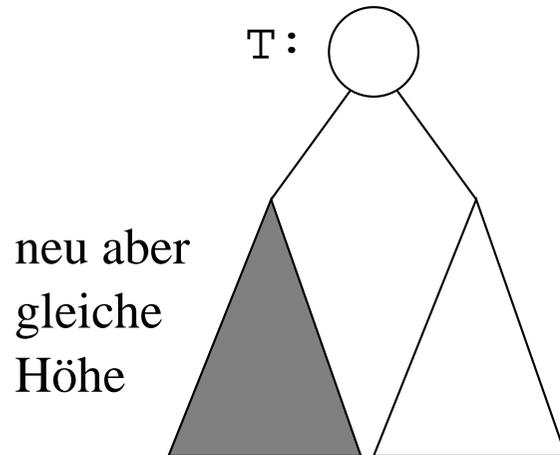
## RebalanceLeft(T, left\_deeper)

### Fall ReblL-1

left\_deeper = *false*

Aktion:

deeper ← *false* ;  
// T.bal stimmt noch

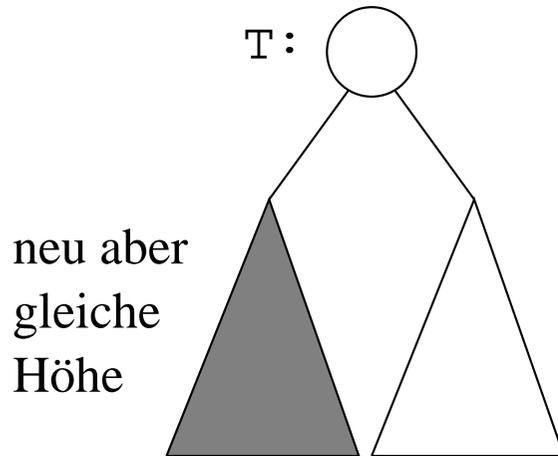


## RebalanceLeft(T, left\_deeper)

### Fall Rebl-1

left\_deeper = *false*

Aktion:



deeper ← *false* ;  
// T.bal stimmt noch

// Wenn dieser Fall einmal eingetreten ist,  
// setzt er sich nach oben immer weiter fort.  
// D. h.: Rebalancierung ist abgeschlossen.

---

## RebalanceInsLeft(T, left\_deeper)

### Fall Rebil-2

**Aktion:**

left\_deeper = *true*  $\wedge$   
T.bal = 0 // alter Wert!

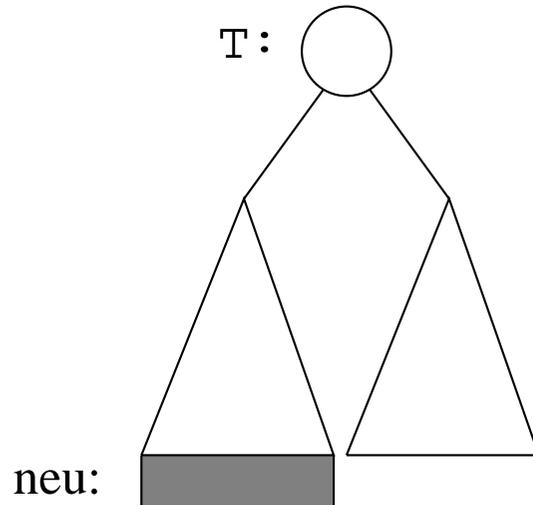
---

## RebalanceLeft(T, left\_deeper)

### Fall ReIL-2

Aktion:

left\_deeper = *true*  $\wedge$   
T.bal = 0 // alter Wert!



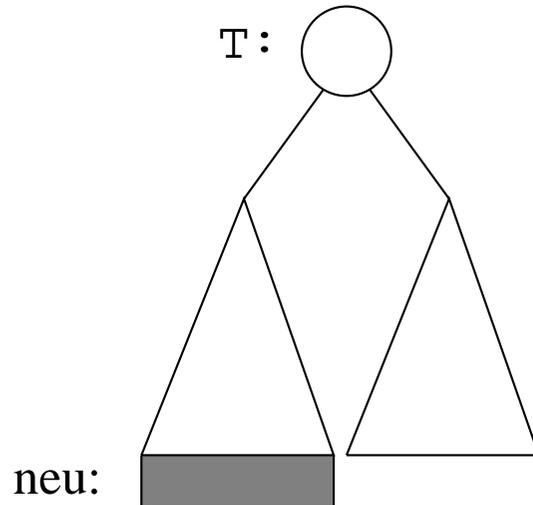
## RebalanceLeft(T, left\_deeper)

### Fall ReIL-2

left\_deeper = **true**  $\wedge$   
T.bal = **0** // alter Wert!

Aktion:

deeper  $\leftarrow$  **true** ;  
T.bal  $\leftarrow$  **-1** ;



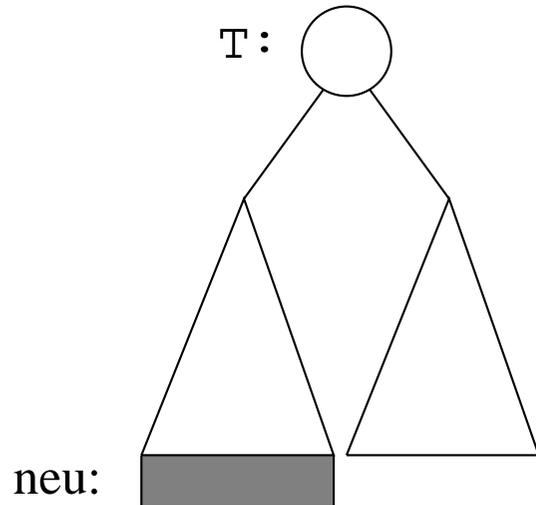
## RebalanceLeft(T, left\_deeper)

### Fall ReIL-2

left\_deeper = **true**  $\wedge$   
T.bal = **0** // alter Wert!

Aktion:

deeper  $\leftarrow$  **true** ;  
T.bal  $\leftarrow$  **-1** ;



// Die Rebalancierungsaufgabe wird nach oben weitergereicht. Balancefaktor in Wurzel ist  $\neq 0$ .

---

RebalanceInsLeft(T, left\_deeper)

**Fall Rebil-3**

**Aktion:**

left\_deeper = **true**  $\wedge$   
T.bal = **1** // alter Wert!

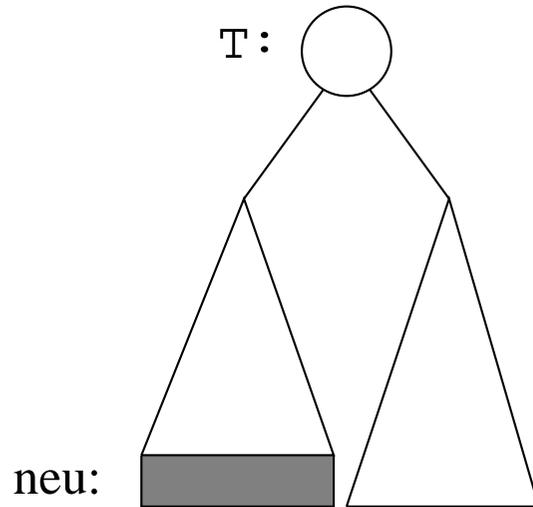
---

## RebalanceLeft(T, left\_deeper)

### Fall Rebl-3

Aktion:

left\_deeper = *true*  $\wedge$   
T.bal = 1 // alter Wert!



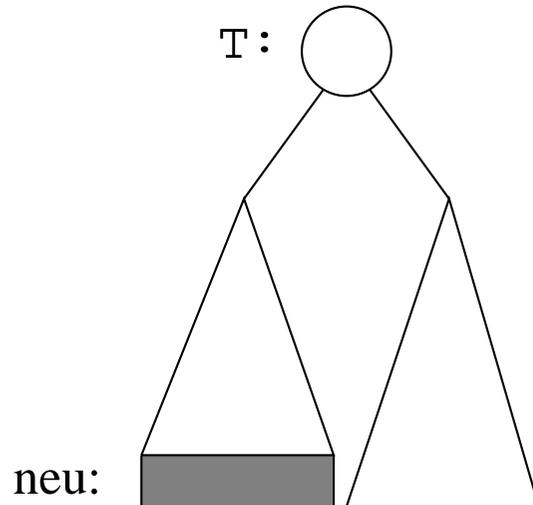
## RebalanceLeft(T, left\_deeper)

### Fall Rebl-3

left\_deeper = **true**  $\wedge$   
T.bal = **1** // alter Wert!

Aktion:

deeper  $\leftarrow$  **false** ;  
T.bal  $\leftarrow$  0 ;



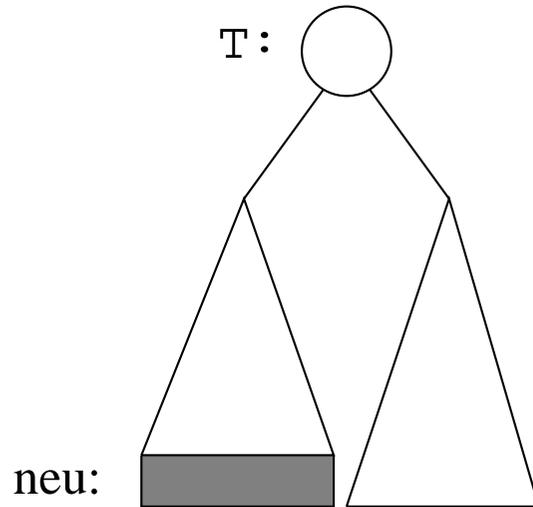
## RebalanceLeft(T, left\_deeper)

### Fall Rebl-3

left\_deeper = **true**  $\wedge$   
T.bal = **1** // alter Wert!

Aktion:

deeper  $\leftarrow$  **false** ;  
T.bal  $\leftarrow$  0 ;



// Rebalancierung ist abgeschlossen, ohne Strukturänderung.

---

## RebalanceInsLeft(T, left\_deeper)

---

## RebalanceInsLeft(T, left\_deeper)

### Fall ReIL-4

**Aktion:**

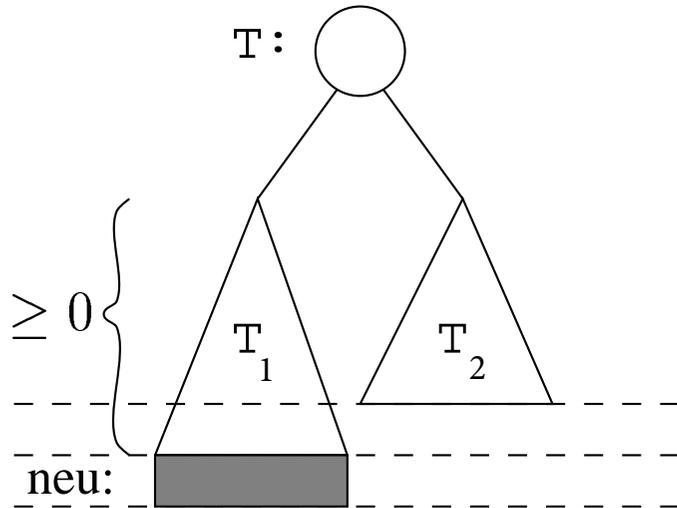
```
left_deeper = true ^  
T.bal = -1 // alter Wert!
```

## RebalanceLeft(T, left\_deeper)

### Fall Rebl-4

Aktion:

left\_deeper = **true**  $\wedge$   
T.bal = **-1** // alter Wert!

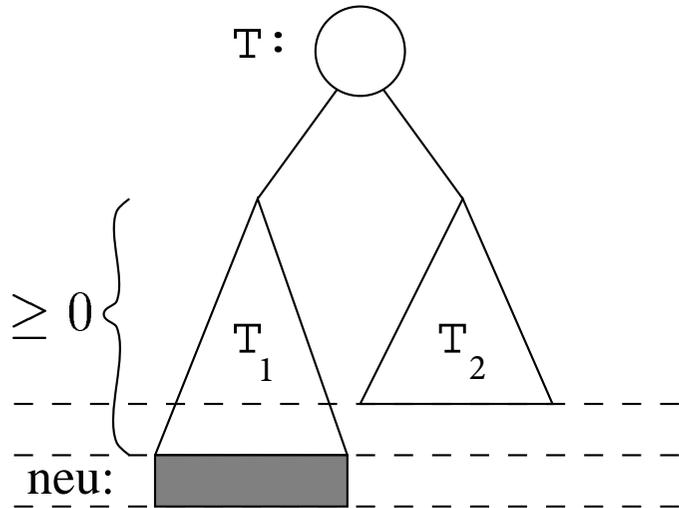


## RebalanceLeft(T, left\_deeper)

### Fall Rebl-4

Aktion:

left\_deeper = **true**  $\wedge$   
T.bal = **-1** // alter Wert!



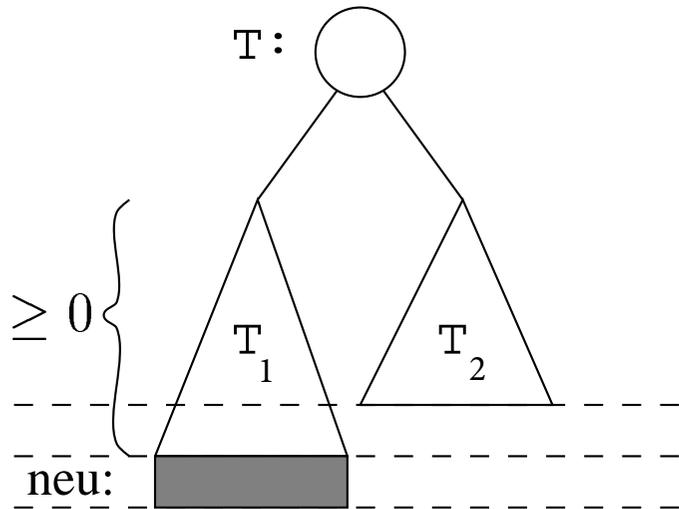
???

## RebalanceLeft(T, left\_deeper)

### Fall ReIL-4

Aktion:

left\_deeper = **true**  $\wedge$   
T.bal = **-1** // alter Wert!



// Strukturänderung nötig.

---

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL\_insert**(`T.left`, `x`, `r`) muss der linke Unterbaum  $T_1 = T.\text{left}$  von  $T$  nicht leer gewesen sein.

---

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL\_insert**(*T.left*, *x*, *r*) muss der linke Unterbaum  $T_1 = T.left$  von  $T$  nicht leer gewesen sein.

Schon gesehen: **RebIL-1**, **RebIL-3** (analog: **RebIR-1**, **RebIR-3**) liefern `deeper = false`.

---

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL\_insert**( $T.\text{left}, x, r$ ) muss der linke Unterbaum  $T_1 = T.\text{left}$  von  $T$  nicht leer gewesen sein.

Schon gesehen: **RebIL-1**, **RebIL-3** (analog: **RebIR-1**, **RebIR-3**) liefern `deeper = false`.

Werden sehen: **RebIL-4** und **RebIR-4** liefern ebenfalls `deeper = false`.

---

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL\_insert**(*T.left*, *x*, *r*) muss der linke Unterbaum  $T_1 = T.left$  von  $T$  nicht leer gewesen sein.

Schon gesehen: **RebIL-1**, **RebIL-3** (analog: **RebIR-1**, **RebIR-3**) liefern `deeper = false`.

Werden sehen: **RebIL-4** und **RebIR-4** liefern ebenfalls `deeper = false`.

⇒ beim rekursiven Aufruf der Rebalancierung für *T.left* ist Fall **RebIL-2** oder **RebIR-2** eingetreten (Baum ist tiefer, durch neues Blatt).

---

Beobachte: Schon **vor** dem rekursiven Aufruf **AVL\_insert**(*T.left*, *x*, *r*) muss der linke Unterbaum  $T_1 = T.left$  von  $T$  nicht leer gewesen sein.

Schon gesehen: **RebIL-1**, **RebIL-3** (analog: **RebIR-1**, **RebIR-3**) liefern `deeper = false`.

Werden sehen: **RebIL-4** und **RebIR-4** liefern ebenfalls `deeper = false`.

⇒ beim rekursiven Aufruf der Rebalancierung für *T.left* ist Fall **RebIL-2** oder **RebIR-2** eingetreten (Baum ist tiefer, durch neues Blatt).

⇒ Balancefaktor *T.left.balance* in der Wurzel von  $T_1$  ist  $-1$  oder  $1$ .

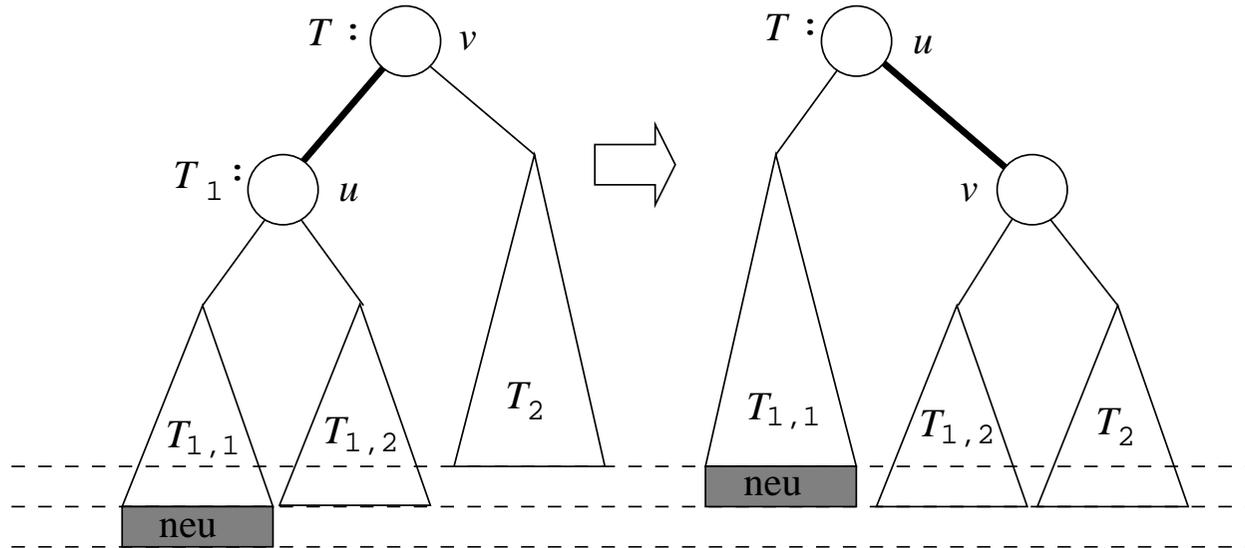
---

## Unterfall ReblL-4.1

`T.left.bal = -1`

## Unterfall Rebl-4.1

`T.left.bal = -1`

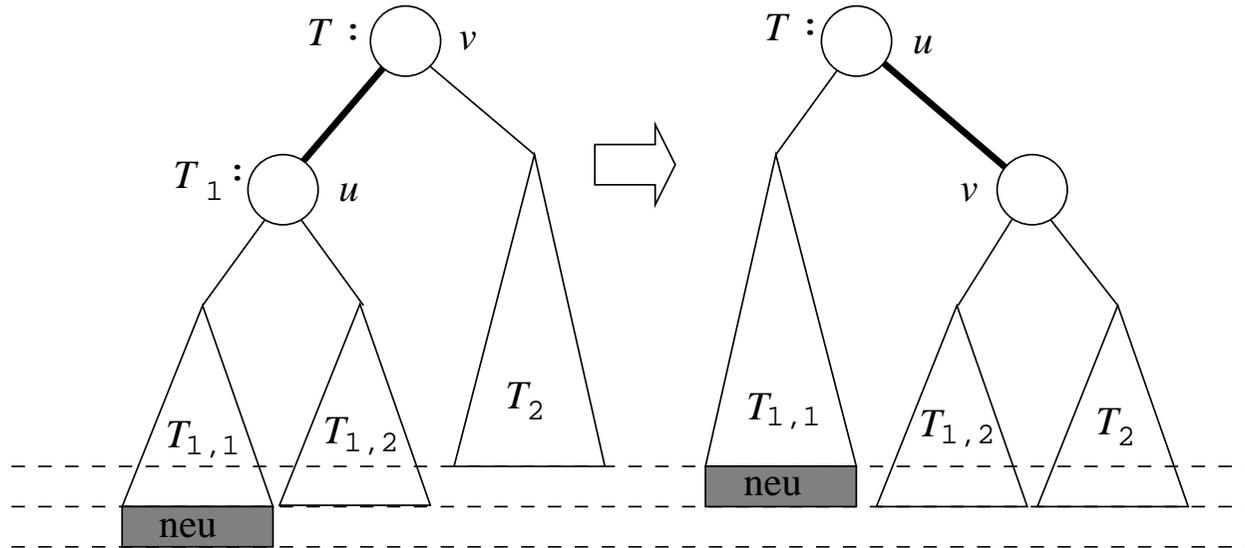


## Unterfall ReblL-4.1

T.left.bal = **-1**

Aktion:

**Rechtsrotation**

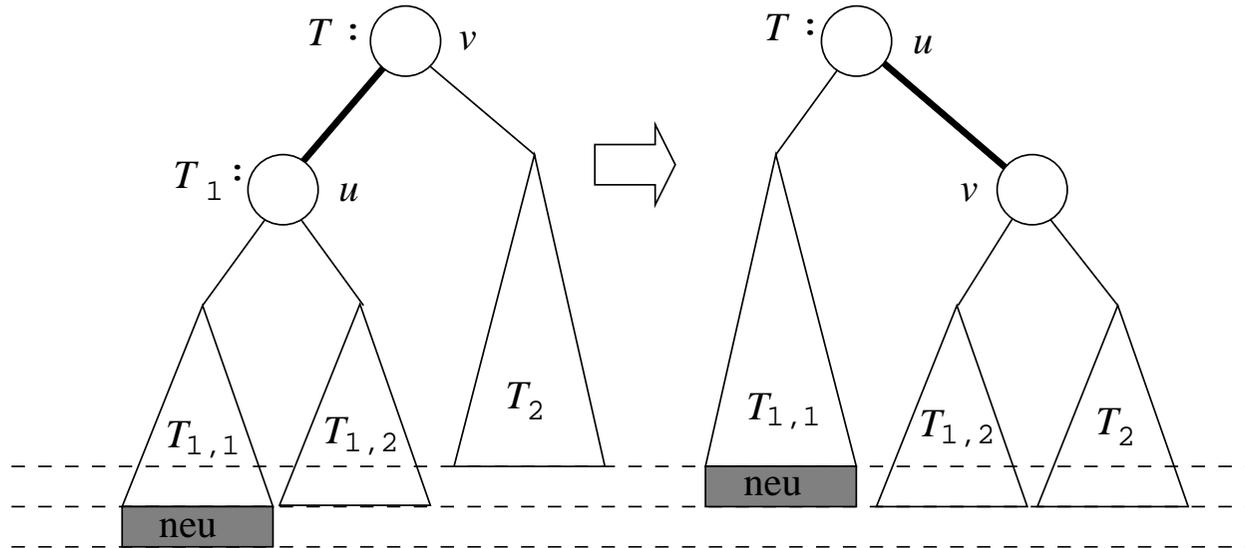


## Unterfall ReblL-4.1

$T.\text{left.bal} = -1$

Aktion:

Rechtsrotation



```
T ← rotateR(T) ;  
deeper ← false ;  
T.bal ← 0 ; // Knoten u  
T.right.bal ← 0 ; // Knoten v
```

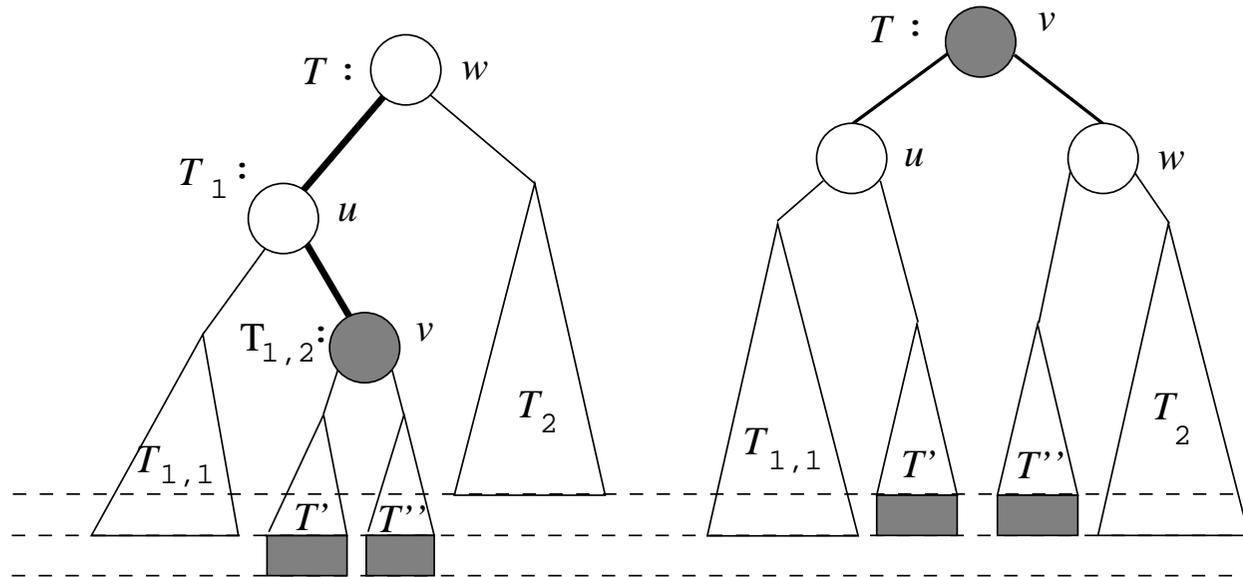
---

## Unterfall ReblL-4.2

T.left.bal = 1

## Unterfall ReblL-4.2

$T.\text{left.bal} = 1$

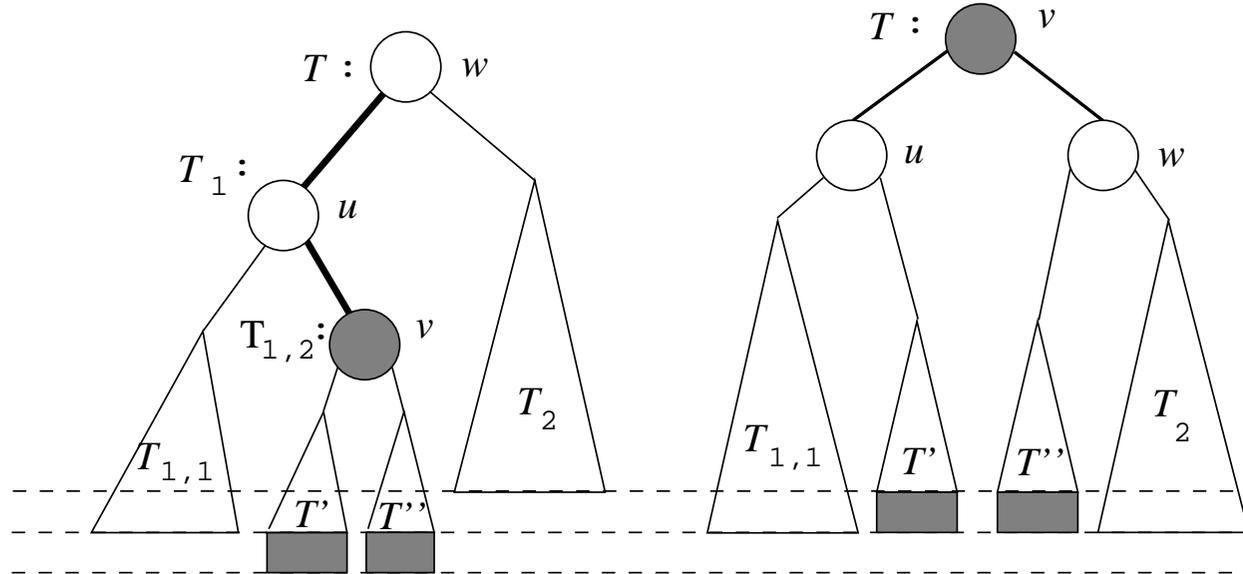


## Unterfall ReblL-4.2

$T.\text{left.bal} = 1$

Aktion:

Links-Rechts-Doppelrotation

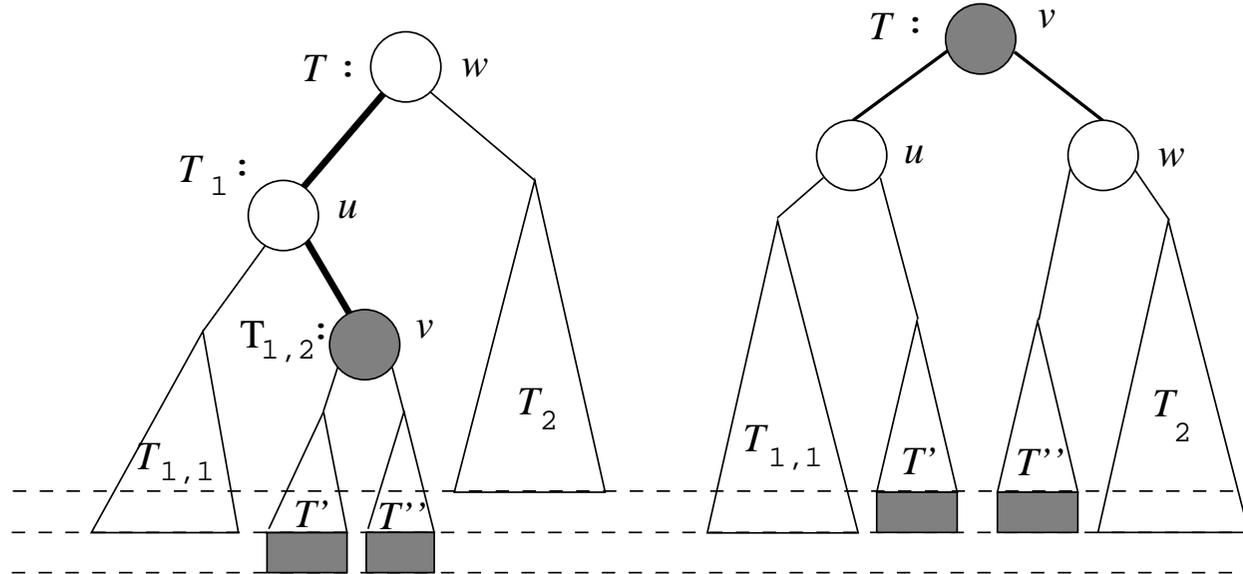


## Unterfall ReblL-4.2

$T.\text{left.bal} = 1$

Aktion:

Links-Rechts-Doppelrotation



$T \leftarrow \text{rotateLR}(T) ;$

$\text{deeper} \leftarrow \text{false} ;$

Neue bal-Werte: s. Tabelle.

---

Neue ba1-Werte im Fall ReIL-4.2:

---

Neue bal-Werte im Fall Rebil-4.2:

(alt)	(neu)		
<i>v.bal</i>	<i>u.bal</i>	<i>w.bal</i>	<i>v.bal</i>
-1	0	1	0
0	0	0	0
1	-1	0	0

---

Neue bal-Werte im Fall ReblL-4.2:

(alt)	(neu)		
$v.bal$	$u.bal$	$w.bal$	$v.bal$
-1	0	1	0
0	0	0	0
1	-1	0	0

$u$  ist Wurzel von  $T.left$ ,  $w$  ist Wurzel von  $T.right$ ,  $v$  ist die Wurzel von  $T$ .

Die Zahl  $v.bal$  (alt) stand schon **vor der LR-Rotation** in  $v.bal$ , gibt also den Balancefaktor des alten Unterbaums  $T_{1,2} = T.left.right$  an.

---

Neue bal-Werte im Fall ReblL-4.2:

(alt)	(neu)		
$v.bal$	$u.bal$	$w.bal$	$v.bal$
-1	0	1	0
0	0	0	0
1	-1	0	0

$u$  ist Wurzel von  $T.left$ ,  $w$  ist Wurzel von  $T.right$ ,  $v$  ist die Wurzel von  $T$ .

Die Zahl  $v.bal$  (alt) stand schon **vor der LR-Rotation** in  $v.bal$ , gibt also den Balancefaktor des alten Unterbaums  $T_{1,2} = T.left.right$  an.

**Überlege:** Kann es überhaupt passieren, dass der alte Baum  $T_{1,2}$  Balancefaktor 0 hatte?

---

Neue bal-Werte im Fall ReblL-4.2:

(alt)	(neu)		
$v.bal$	$u.bal$	$w.bal$	$v.bal$
-1	0	1	0
0	0	0	0
1	-1	0	0

$u$  ist Wurzel von  $T.left$ ,  $w$  ist Wurzel von  $T.right$ ,  $v$  ist die Wurzel von  $T$ .

Die Zahl  $v.bal$  (alt) stand schon **vor der LR-Rotation** in  $v.bal$ , gibt also den Balancefaktor des alten Unterbaums  $T_{1,2} = T.left.right$  an.

**Überlege:** Kann es überhaupt passieren, dass der alte Baum  $T_{1,2}$  Balancefaktor 0 hatte?

**Antwort: Ja!** Im rekursiven Aufruf **AVL\_insert**( $T.left, x, r$ ) bestand  $T.left$  nur aus einem Knoten,  $T_{1,2}$  ist der neu eingefügte Knoten;  $T'$ ,  $T''$ ,  $T_{1,1}$  und  $T_2$  sind leer, haben also alle Tiefe  $-1$ .

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert** $(T, x, r)$  führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für  $insert(f_T, x, r)$ .

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert** $(T, x, r)$  führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für  $insert(f_T, x, r)$ .

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für  $insert(f_T, x, r)$ .

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL\_insert** die intuitive Beschreibung von Folie 60 umsetzt.

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für  $insert(f_T, x, r)$ .

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL\_insert** die intuitive Beschreibung von Folie 60 umsetzt.

### 4.3.5 Folgerung (aus Algorithmus **AVL\_insert**):

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für *insert*( $f_T, x, r$ ).

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL\_insert** die intuitive Beschreibung von Folie 60 umsetzt.

### 4.3.5 Folgerung (aus Algorithmus **AVL\_insert**):

Für jedes  $n \geq 0$  gibt es einen höhenbalancierten Baum mit  $n$  Knoten.

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für *insert*( $f_T, x, r$ ).

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL\_insert** die intuitive Beschreibung von Folie 60 umsetzt.

### 4.3.5 Folgerung (aus Algorithmus **AVL\_insert**):

Für jedes  $n \geq 0$  gibt es einen höhenbalancierten Baum mit  $n$  Knoten.

*Beweis:* Man fügt  $1, \dots, n$  (in beliebiger Reihenfolge) mit **AVL\_insert** in einen anfangs leeren AVL-Baum ein. **Gute Übung:** Führe dies für  $n = 16$  (z. B.) von Hand aus.

---

### Proposition 4.3.4

Die rekursive Prozedur **AVL\_insert**( $T, x, r$ ) führt die Wörterbuchoperation *insert* korrekt durch, d. h.: Aus  $T$  entsteht ein AVL-Baum für *insert*( $f_T, x, r$ ).

Die Prozedur hat Laufzeit  $O(\log n)$  und führt höchstens eine Einfach- oder Doppelrotation durch.

Man überlege (an Beispielen), wie die beschriebene rekursive Realisierung von **AVL\_insert** die intuitive Beschreibung von Folie 60 umsetzt.

### 4.3.5 Folgerung (aus Algorithmus **AVL\_insert**):

Für jedes  $n \geq 0$  gibt es einen höhenbalancierten Baum mit  $n$  Knoten.

*Beweis:* Man fügt  $1, \dots, n$  (in beliebiger Reihenfolge) mit **AVL\_insert** in einen anfangs leeren AVL-Baum ein. **Gute Übung:** Führe dies für  $n = 16$  (z. B.) von Hand aus.

Bemerkung: Wenn  $T$  ein beliebiger höhenbalancierter Baum mit  $n$  Knoten ist, dann gibt es eine Einfügereihenfolge für Schlüssel  $1, \dots, n$ , die genau diesen Baum erzeugt – sogar ohne jede Rotation (**Übung**).

**Vorlesungsvideo:**  
**AVL\_Delete**

---

**AVL\_delete( $T, x$ )**

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus.

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.  
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)

---

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
  2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
  3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.  
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)
- 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) der (strikt) tiefste ist:

---

**AVL\_delete( $T, x$ )**

**Anschauliche Beschreibung:**

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d.h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.  
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)
- 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) der (strikt) tiefste ist:  
Die entsprechende **Doppelrotation** hebt diesen mittleren Teilbaum ein Level höher.

## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
3. Wenn an  $v$  die Balancebedingung verletzt ist, d.h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.  
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)
- 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) der (strikt) tiefste ist:  
Die entsprechende **Doppelrotation** hebt diesen mittleren Teilbaum ein Level höher.

Zum Ausprobieren: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

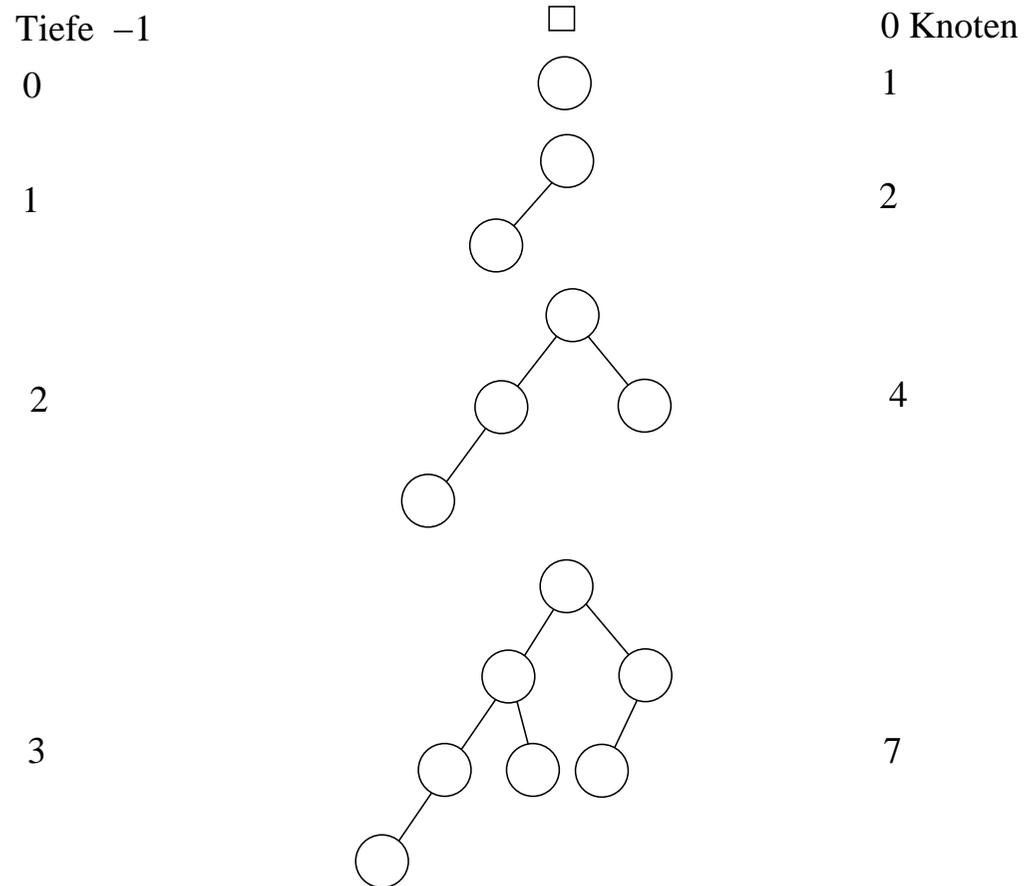
## AVL\_delete( $T, x$ )

### Anschauliche Beschreibung:

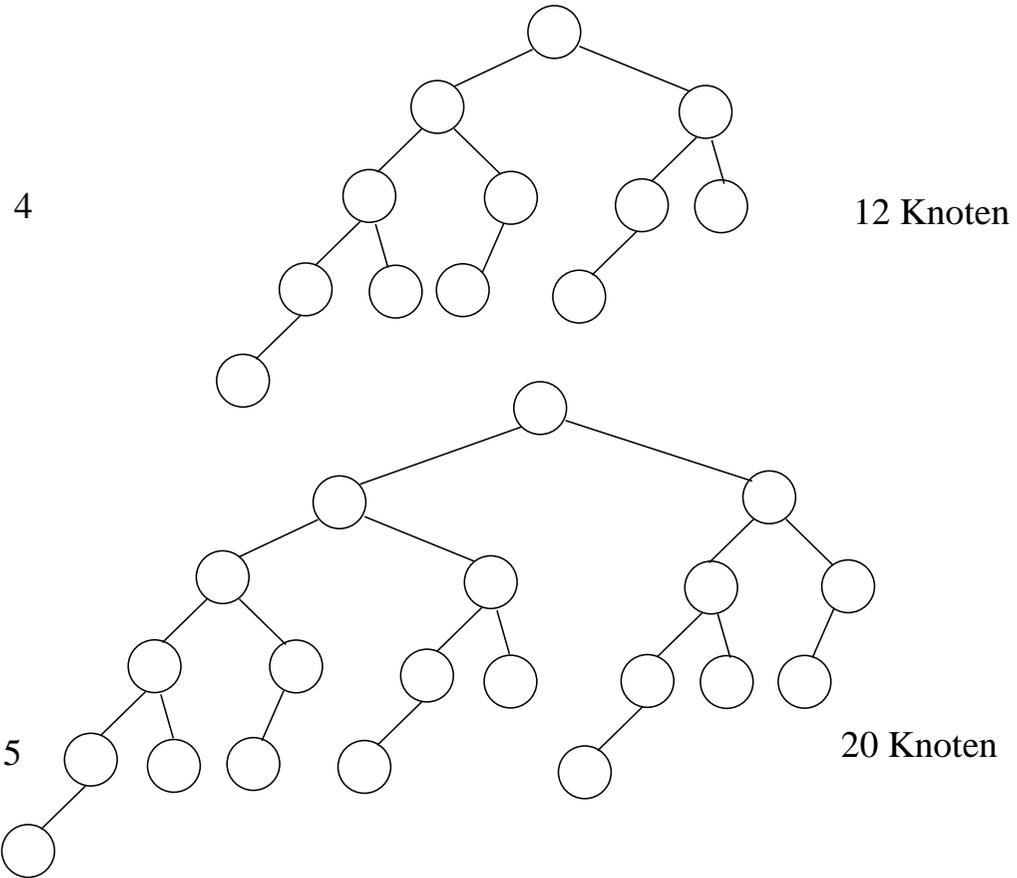
1. Suche Knoten mit  $x$  und lösche ihn wie bei gewöhnlichem BSB. Dadurch wird ein Knoten  $u$  mit  $T_u.left = \square$  oder  $T_u.right = \square$  entfernt. Sei  $w$  der Vorgängerknoten von  $u$ . Einer der Unterbäume von  $w$  ist nun „flacher“ als vorher.
  2. Laufe Weg von  $w$  zur Wurzel **von unten nach oben**, kontrolliere an jedem Knoten  $v$  dieses Weges die Balancebedingung.
  3. Wenn an  $v$  die Balancebedingung verletzt ist, d. h.  $bal(v) \in \{-2, 2\}$ , führe Einfach- oder Doppelrotation aus. **Aber welche?**
- 3a) Wenn ein **äußerer** Teilbaum unter  $v$  (links-links oder rechts-rechts) der tiefste ist:  
**Einfache Rotation**, um diesen Teilbaum ein Level höher zu heben.  
(Möglich ist: Der Geschwister-Teilbaum ist genauso tief. Dann ist die Rebalancierung beendet.)
- 3b) Wenn ein **mittlerer** Teilbaum unter  $v$  (links-rechts oder rechts-links) der (strikt) tiefste ist:  
Die entsprechende **Doppelrotation** hebt diesen mittleren Teilbaum ein Level höher.

Zum Ausprobieren: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

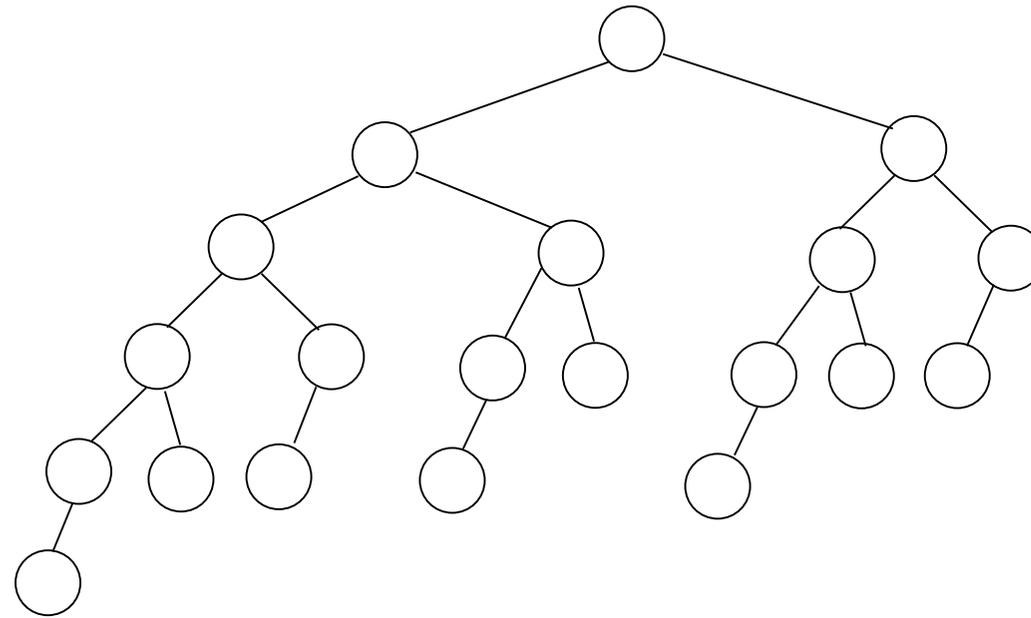
*Beispiel:* Fibonacci-Bäume . . .

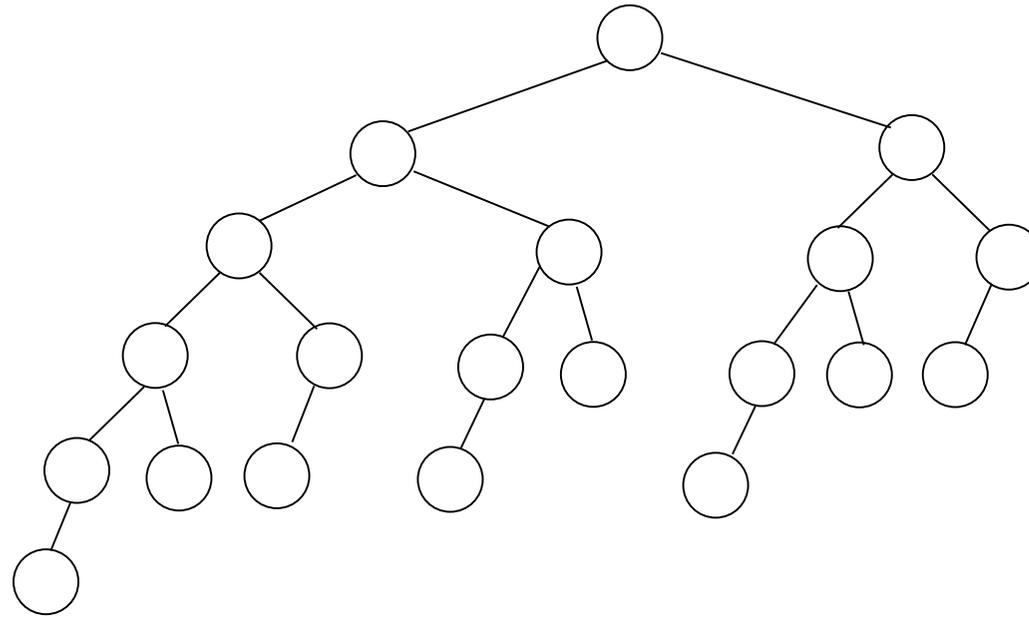


Fibonacci-Bäume der Tiefen  $-1, 0, 1, 2, 3$



Fibonacci-Bäume der Tiefen 4 und 5

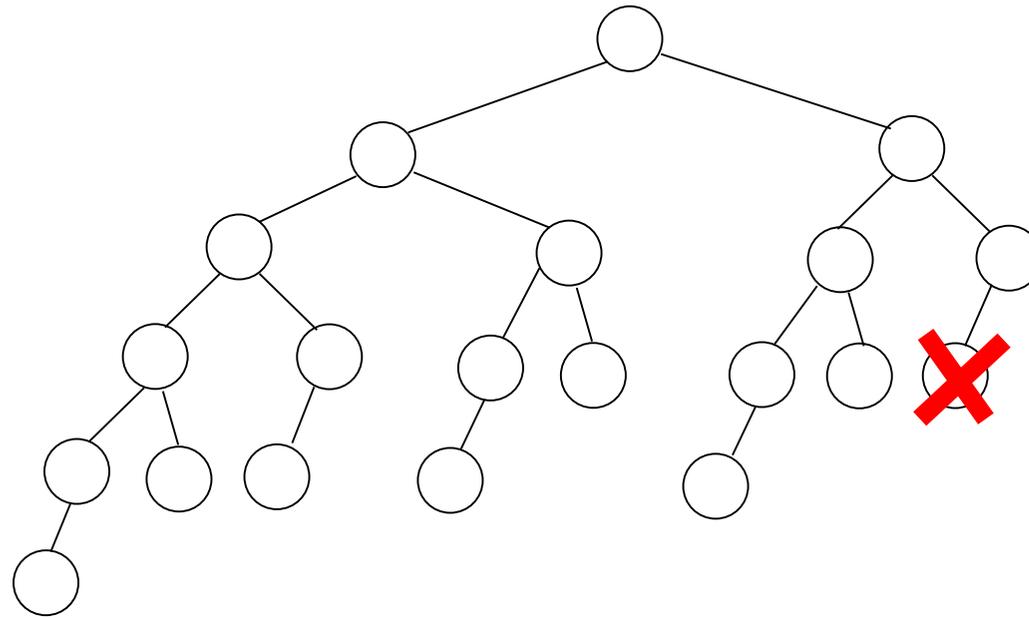




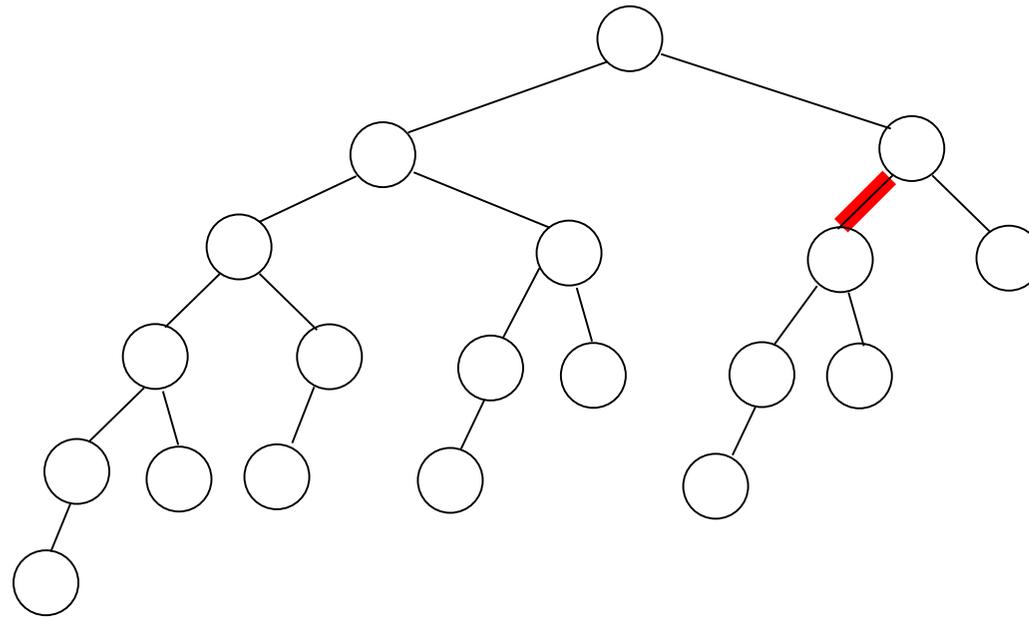
Blatt  $v$ :

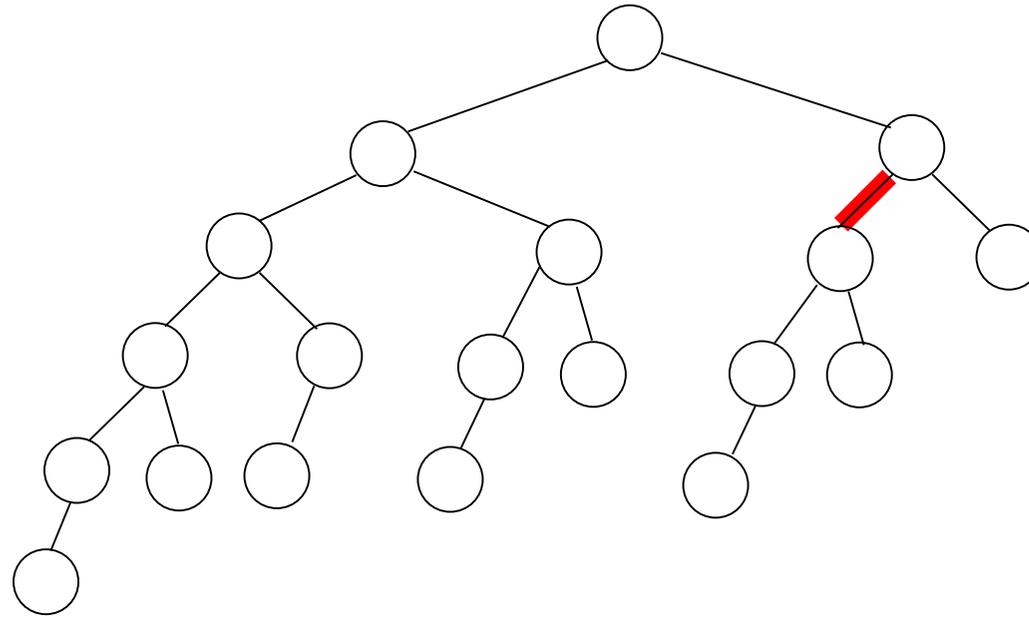
Starte bei Wurzel, gehe rekursiv in flacheren Unterbaum.

Entfernen von  $v$  löst Rotations**kaskade** aus.

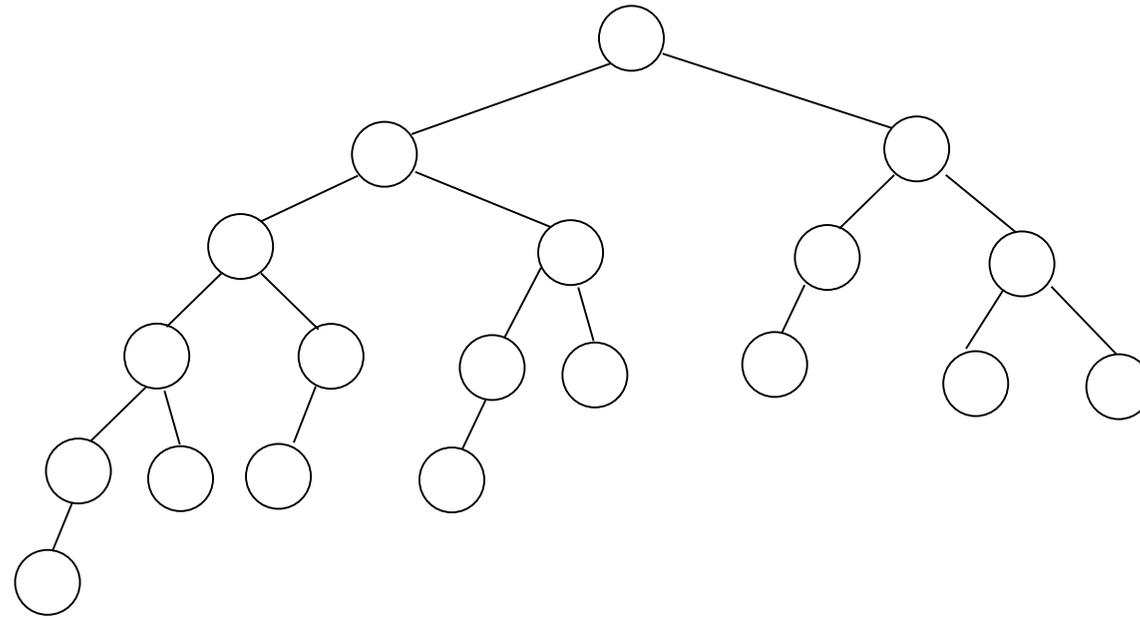


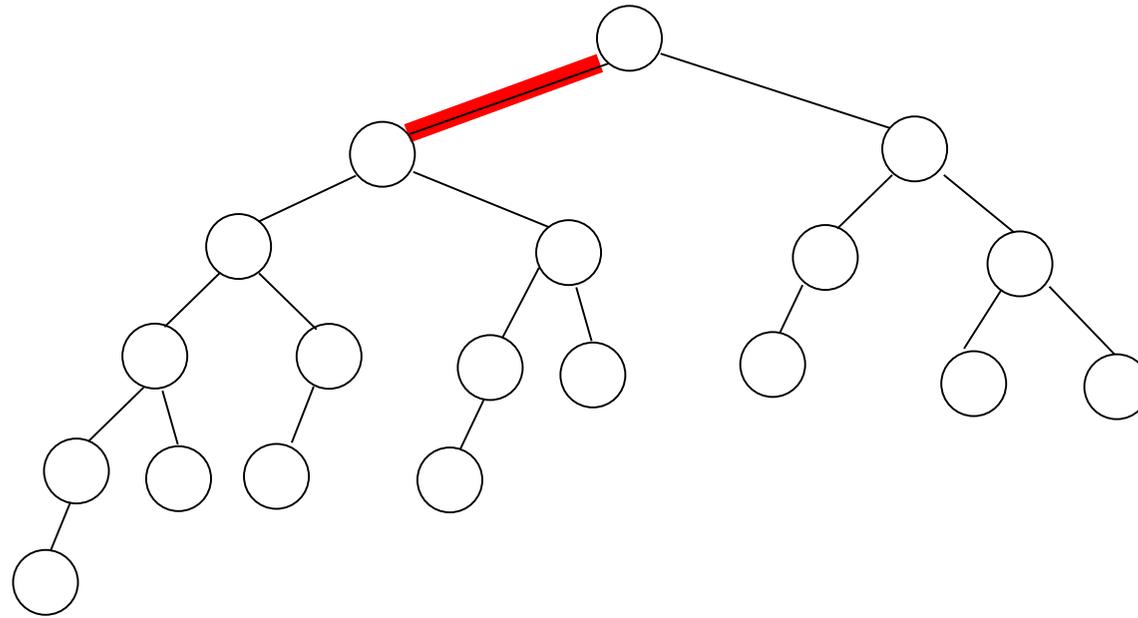


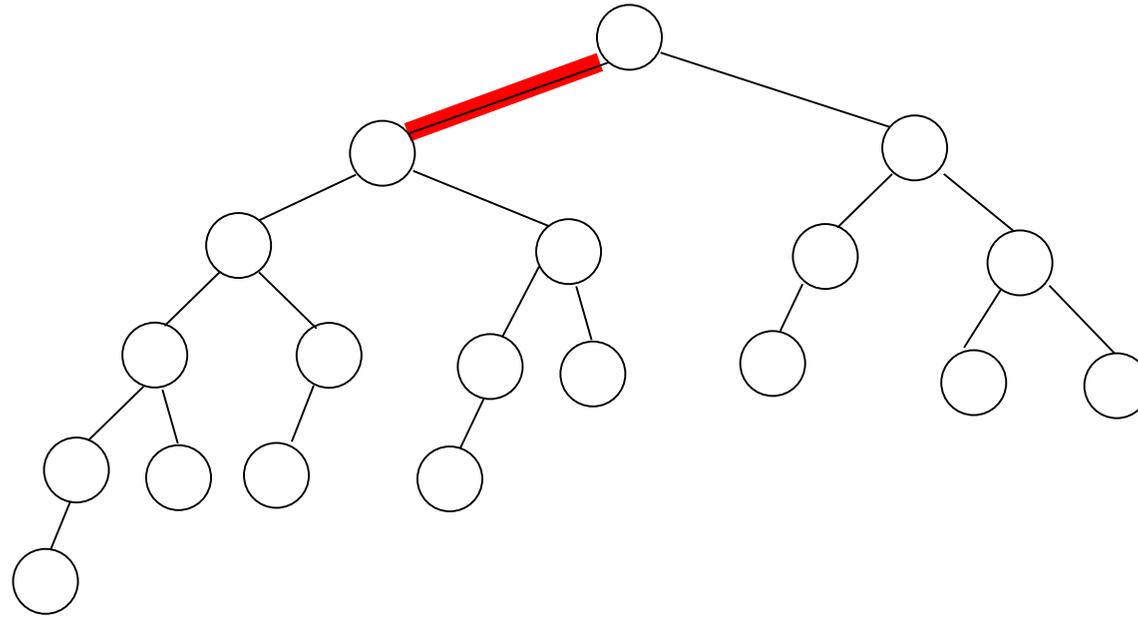




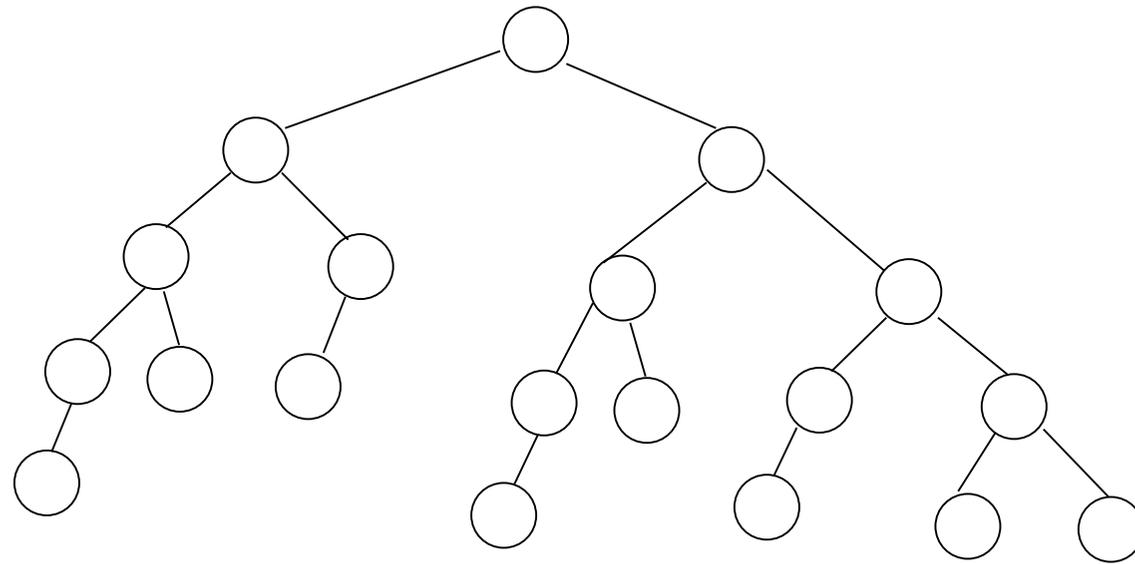
**Rechtsrotation!**

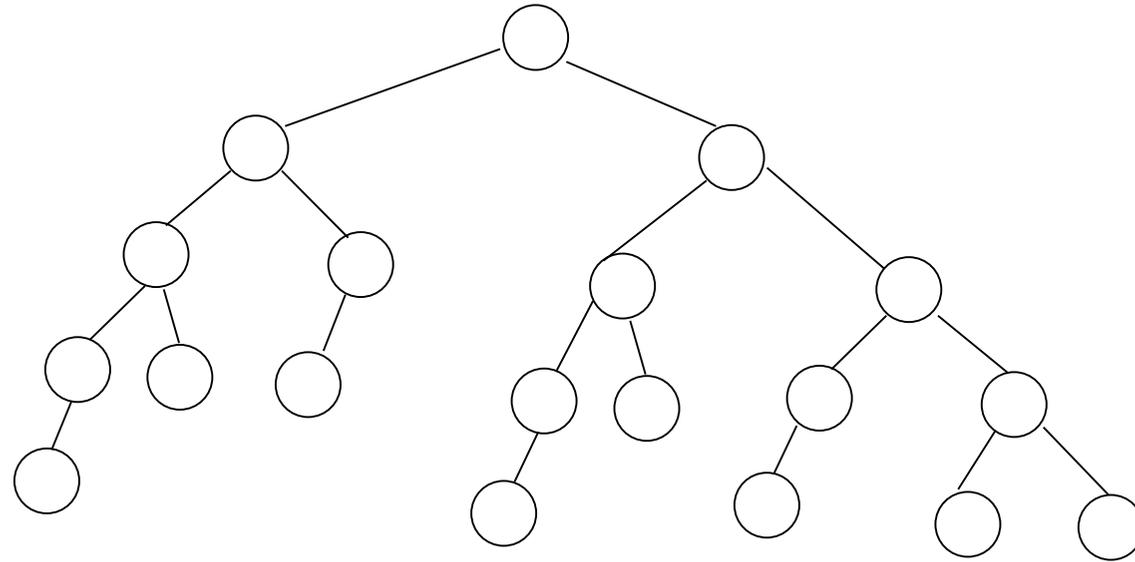




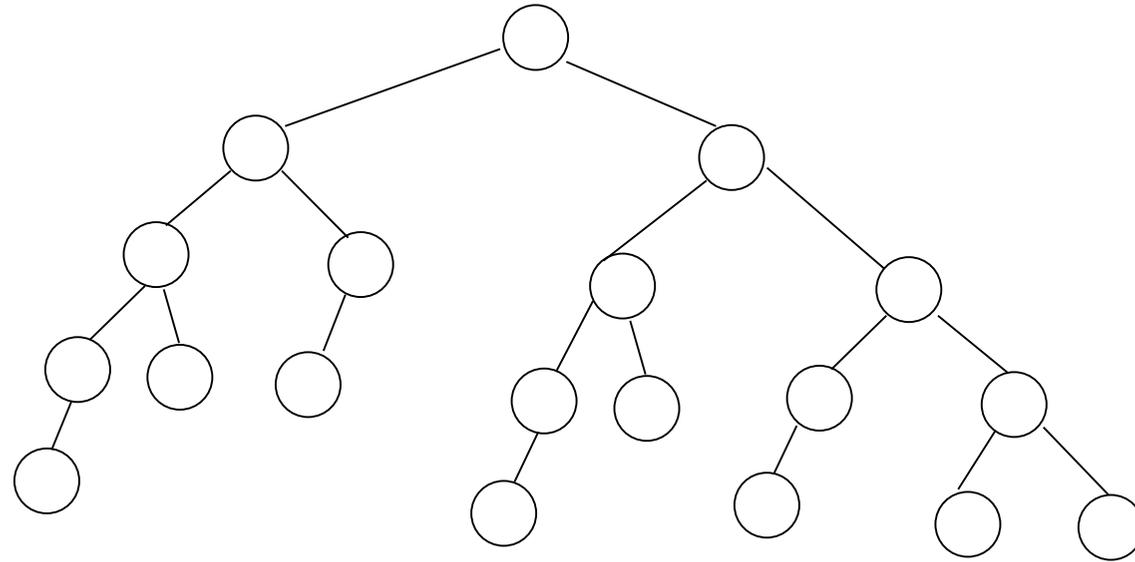


**Rechtsrotation!**



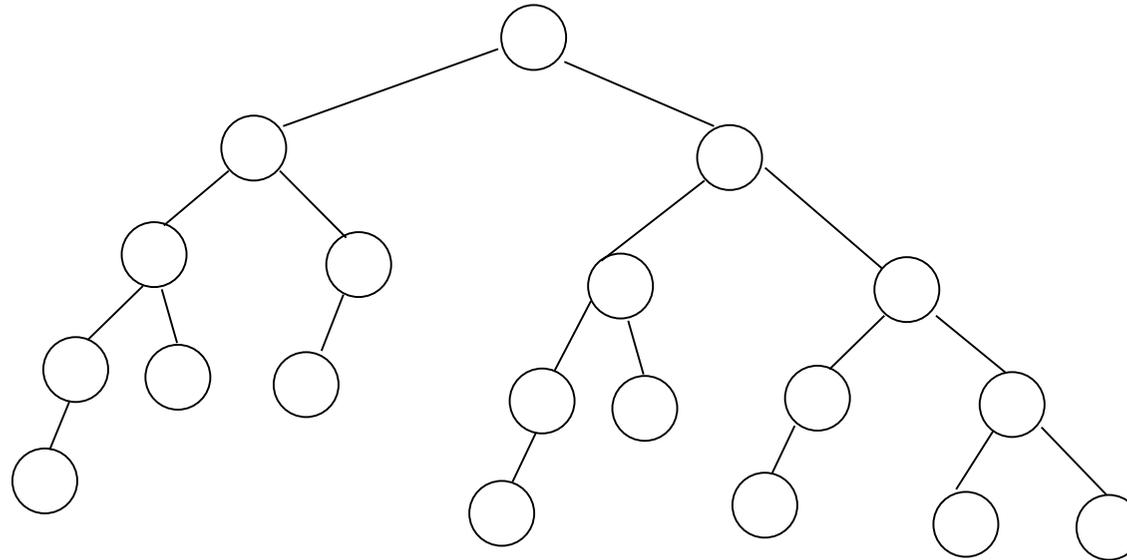


**Fertig!**



**Fertig!**

Programmierung von **AVL\_delete**: Nicht prüfungsrelevant!



**Fertig!**

Programmierung von **AVL\_delete**: Nicht prüfungsrelevant!

Können muss man aber: Ausführen der Löschung und Rebalancierung, analog zur Einfügung.

---

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

---

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

**Funktion `AVL_delete(T, x)`: (AVL\_Tree, boolean)**

---

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

**Funktion `AVL_delete`**(`T, x`): (`AVL_Tree, boolean`)

(1) // **Eingabe:** `T: AVL_Tree, x: key`

(2) // **Ausgabe:** `T: AVL_Tree, shallower: boolean`

---

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

**Funktion `AVL_delete(T,x)`: (AVL\_Tree,boolean)**

- (1) // **Eingabe:** T: AVL\_Tree, x: key
- (2) // **Ausgabe:** T: AVL\_Tree, shallower: boolean
  
- (3) **1. Fall:** T =  $\square$  // x nicht da
- (4) **return** (T, *false*).
- (5) // **Keine Rebalancierung nötig!**

---

In Prozeduren: `shallower` ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

**Funktion `AVL_delete(T,x)`: (AVL\_Tree,boolean)**

- (1) // **Eingabe:** T: AVL\_Tree, x: key
- (2) // **Ausgabe:** T: AVL\_Tree, shallower: boolean
  
- (3) **1. Fall:** T =  $\square$  // x nicht da
- (4)     **return** (T, *false*).
- (5)         // **Keine Rebalancierung nötig!**
  
- (6) **2. Fall:** T  $\neq$   $\square$

---

In Prozeduren: shallower ist ein Flagbit, das anzeigt, ob der soeben bearbeitete Teilbaum durch die Löschung flacher geworden ist.

**Funktion AVL\_delete**(T,x): (AVL\_Tree,boolean)

- (1) // **Eingabe:** T: AVL\_Tree, x: key
- (2) // **Ausgabe:** T: AVL\_Tree, shallower: boolean
  
- (3) **1. Fall:**  $T = \square$  // x nicht da
- (4) **return** (T, *false*).
- (5) // **Keine Rebalancierung nötig!**
  
- (6) **2. Fall:**  $T \neq \square$  **and**  $x < T.key$ .
- (7) (T.left, **left\_shallower**)  $\leftarrow$  **AVL\_delete**(T.left, x);
- (8) // Rekursives Löschen im linken Unterbaum
- (9) (T, shallower)  $\leftarrow$  **RebalanceDelLeft**(T, **left\_shallower**);
- (10) // Rebalancierung in der Wurzel von T, **s. unten**
- (11) **return** (T, shallower).

---

```
(12) 3. Fall:  $T \neq \square$  and  $T.\text{key} < x$ .
(13)    $(T.\text{right}, \text{right\_shallower}) \leftarrow \text{AVL\_delete}(T.\text{right}, x)$  ;
(14)   // Rekursives Löschen im rechten Unterbaum
(15)    $(T, \text{shallower}) \leftarrow \text{RebalanceDelRight}(T, \text{right\_shallower})$  ;
(16)   // Rebalancierung in der Wurzel von T, symmetrisch zu 2. Fall
(17)   return  $(T, \text{shallower})$  .
```

---

(18) **4. Fall:**  $T \neq \square$  **and**  $T.\text{key} = x$ .

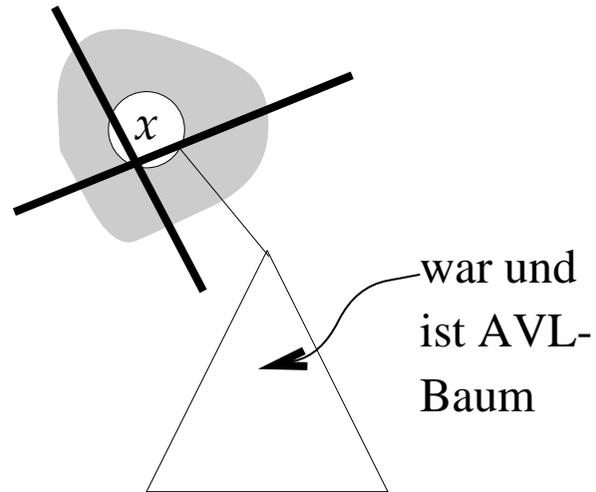
(19) // Entferne Wurzelknoten!

---

(18) **4. Fall:**  $T \neq \square$  and  $T.\text{key} = x$ .

(19) // Entferne Wurzelknoten!

(20) **Fall 4a:**  $T.\text{left} = \square$



(21) **return** ( $T.\text{right}$ , **true**) .

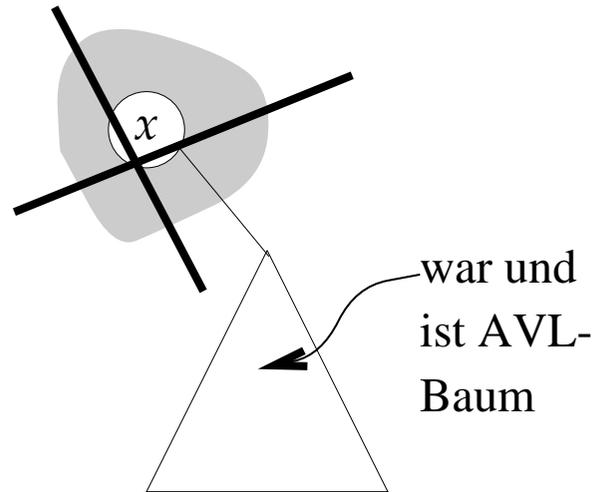
(22) // Baum ist **flacher** als vorher.

---

(18) **4. Fall:**  $T \neq \square$  and  $T.\text{key} = x$ .

(19) // Entferne Wurzelknoten!

(20) **Fall 4a:**  $T.\text{left} = \square$



(21) **return** ( $T.\text{right}$ , *true*) .

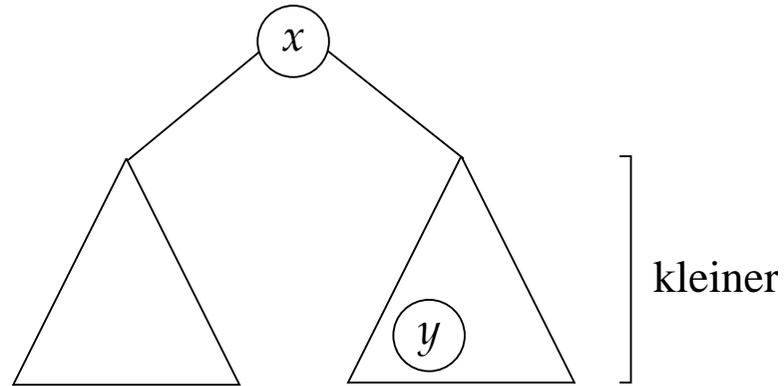
(22) // Baum ist **flacher** als vorher.

(23) **Fall 4b:**  $T.\text{right} = \square$

(24) **return** ( $T.\text{left}$ , *true*) .

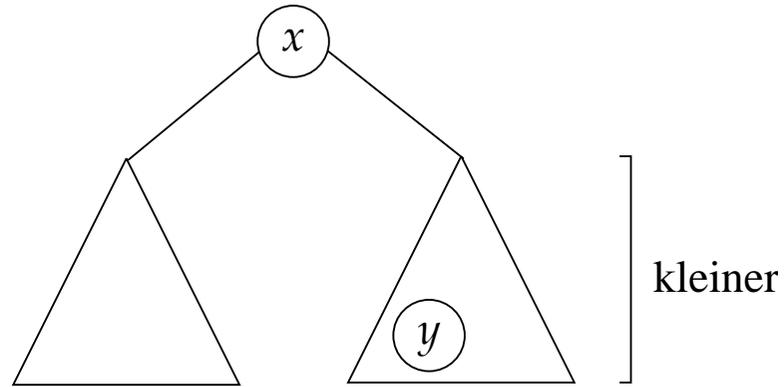
---

(25) **Fall 4c:**  $T \neq \square$  and beide Unterbäume nicht leer



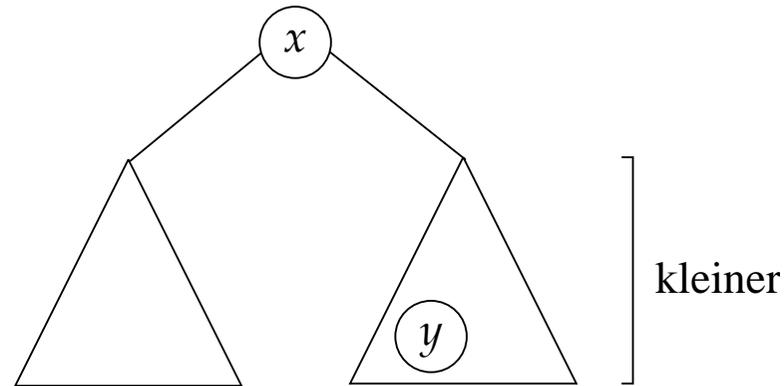
---

(25) **Fall 4c:**  $T \neq \square$  **and** beide Unterbäume nicht leer



(26)  $(T.right, v, \text{right\_shallower}) \leftarrow \text{AVL\_extractMin}(T.right);$

(25) **Fall 4c:  $T \neq \square$  and beide Unterbäume nicht leer**



```
(26) (T.right, v, right_shallower) ← AVL_extractMin(T.right);  
(27) v.left ← T.left; v.right ← T.right;  
(28) v.bal ← T.bal;  
(29) T ← v; // v ersetzt die Wurzel von T  
(30) (T, shallower) ← RebalanceDelRight(T, right_shallower);  
(31) return (T, shallower) .
```

---

**Funktion AVL\_extractMin(T): (AVL\_Tree, boolean)**

(1) // **Eingabe:** T: AVL\_Tree mit  $T \neq \square$

(2) // **Ausgabe:** T, v: AVL\_Tree; shallower: boolean

(3) // Knoten v mit minimalem Eintrag aus T ausgeklint

(4) // shallower = *true*, falls T flacher geworden ist

...

Details: Druckfolien.

---

**RebalanceDelLeft**(T, left\_shallower)

// prüft Balancebedingung in der Wurzel von T, korrigiert

// (**RebalanceDelRight**: symmetrisch)

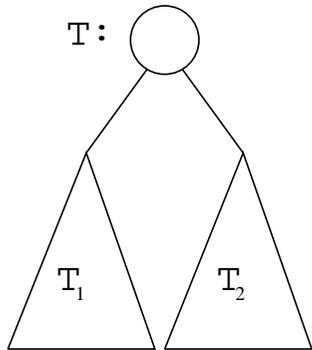
---

## RebalanceDelLeft( $T$ , left\_shallower)

// prüft Balancebedingung in der Wurzel von  $T$ , korrigiert

// (**RebalanceDelRight**: symmetrisch)

Situation:



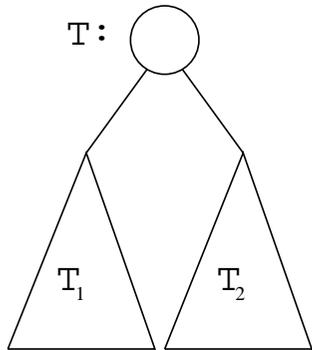
---

## RebalanceDelLeft( $T$ , left\_shallower)

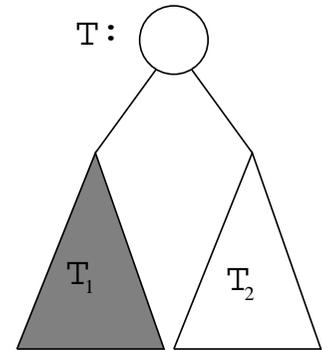
// prüft Balancebedingung in der Wurzel von  $T$ , korrigiert

// (**RebalanceDelRight**: symmetrisch)

Situation:



durch rekursives  
Löschen in  $T_1$   
umgebaut zu



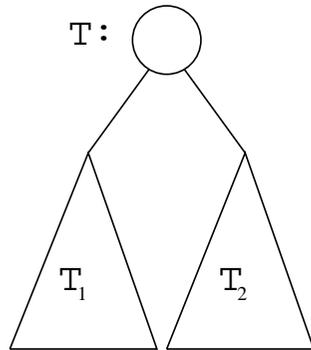
---

## RebalanceDelLeft( $T$ , left\_shallower)

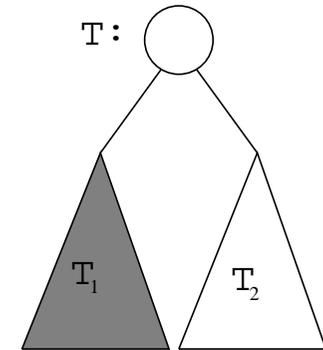
// prüft Balancebedingung in der Wurzel von  $T$ , korrigiert

// (**RebalanceDelRight**: symmetrisch)

Situation:



durch rekursives  
Löschen in  $T_1$   
umgebaut zu



$T_1 = T.left$  verändert; „left\_shallower“ teilt mit, ob  $T_1$  flacher geworden ist.  
 $T.bal$  ist bislang unverändert.

---

Fall

Aktion

**Fall RebDL-1:**

`left_shallower = false`

---

Fall

Aktion

**Fall RebDL-1:**

`left_shallower = false`

`shallower ← false;`

---

Fall

Aktion

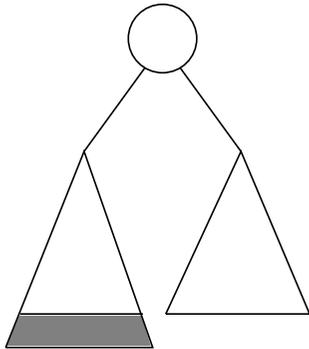
**Fall RebDL-1:**

`left_shallower = false`

`shallower ← false;`

**Fall RebDL-2:**

`left_shallower = true`  $\wedge$  `T.bal = -1`



---

Fall

Aktion

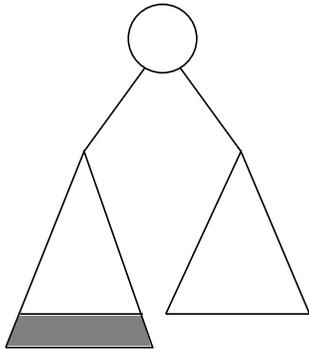
**Fall RebDL-1:**

`left_shallower = false`

`shallower ← false;`

**Fall RebDL-2:**

`left_shallower = true ∧ T.bal = -1`



geschrumpft

`shallower ← true;`

`T.bal ← 0;`

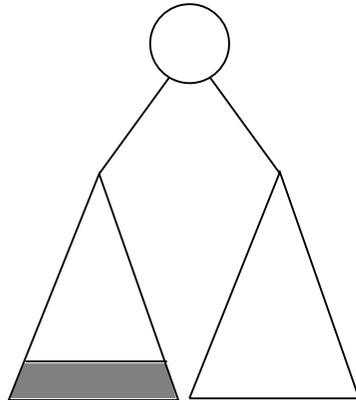
---

Fall

Aktion

**Fall RebDL-3:**

`left_shallower = true`  $\wedge$  `T.bal = 0`



geschrumpft

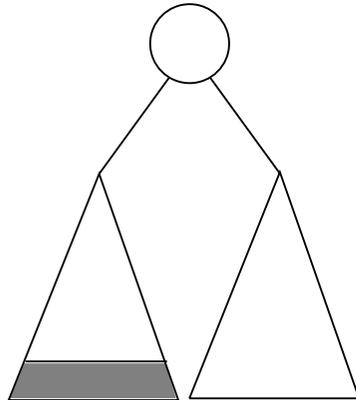
---

Fall

Aktion

**Fall RebDL-3:**

`left_shallower = true`  $\wedge$  `T.bal = 0`



`shallower`  $\leftarrow$  `false`;

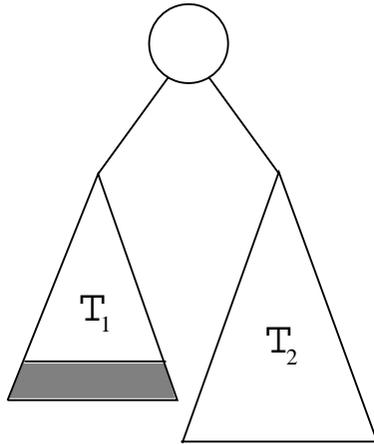
`T.bal`  $\leftarrow$  `1`;

geschrumpft

---

## Fall RebDL-4:

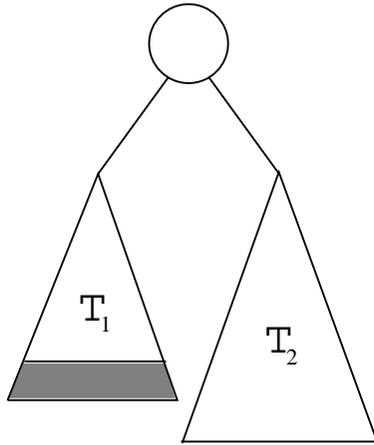
`left_shallower = true`  $\wedge$  `T.bal = 1`



---

## Fall RebDL-4:

`left_shallower = true`  $\wedge$  `T.bal = 1`



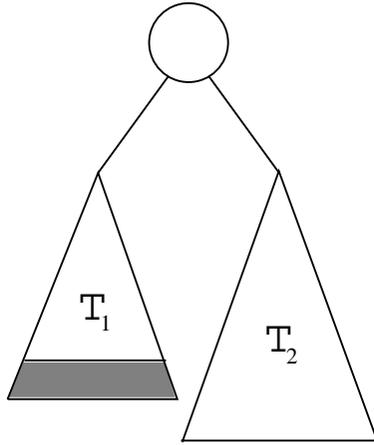
geschrumpft

**Unterfälle** je nach Aussehen von  $T_2$ .

---

## Fall RebDL-4:

`left_shallower = true`  $\wedge$  `T.bal = 1`



geschrumpft

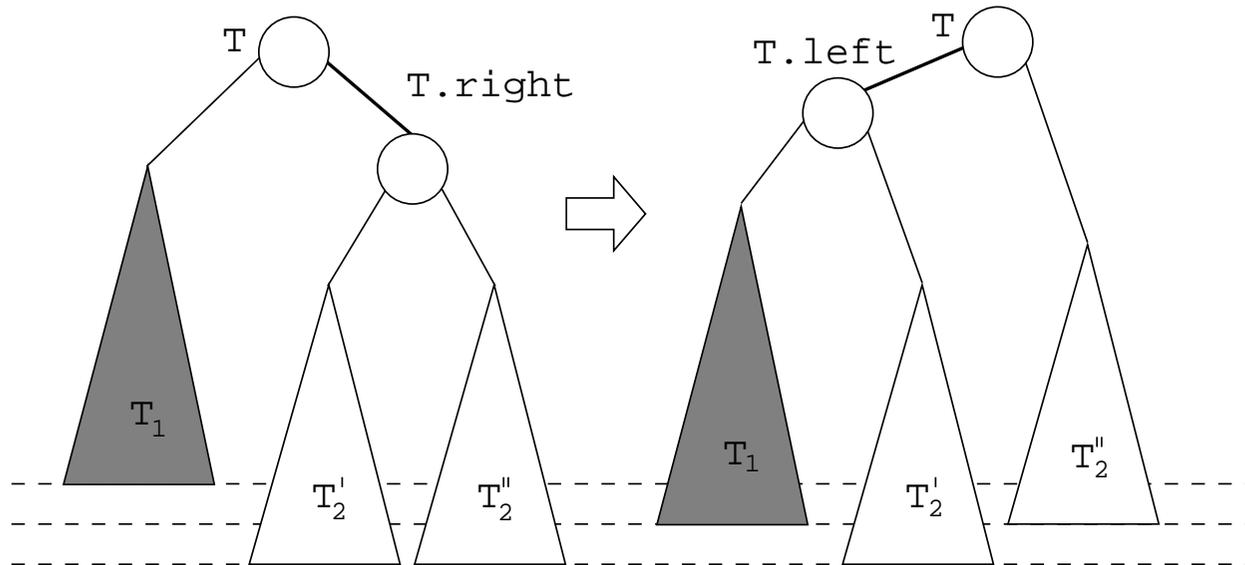
**Unterfälle** je nach Aussehen von  $T_2$ .

Wir wissen:  $T_2$  hat Tiefe mindestens 1.

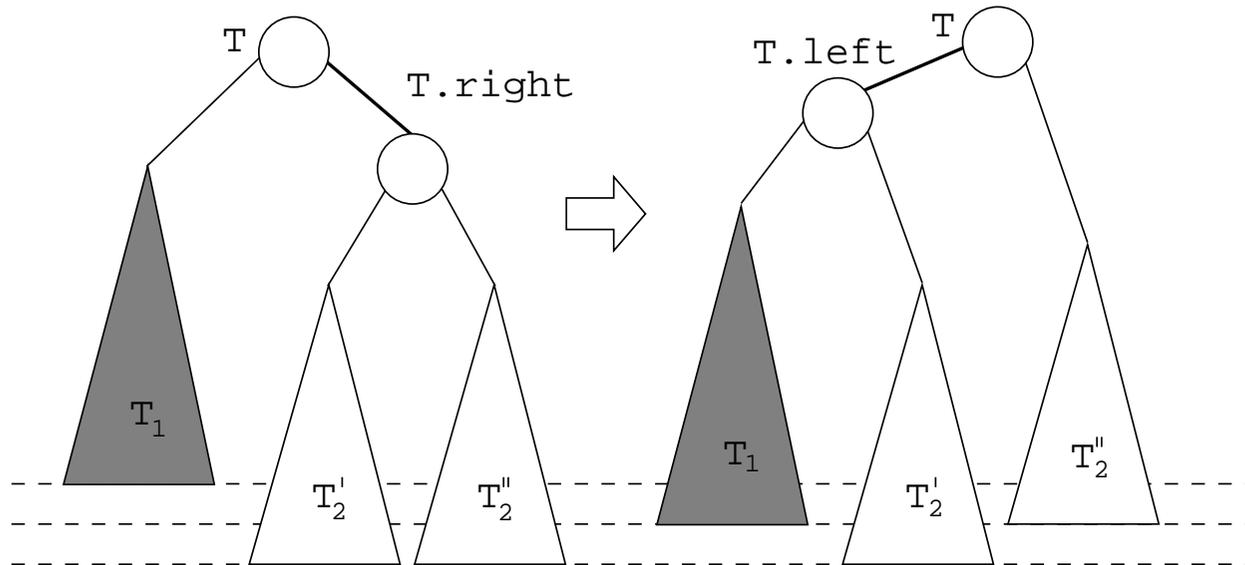
( $T_1$  kann vor der Löschung nicht leer gewesen sein, hatte also Tiefe mindestens 0.)

---

Fall RebDL-4.1:  $T.right.bal = 0$ ; // Zwei gleich tiefe Unterbäume in  $T.right$



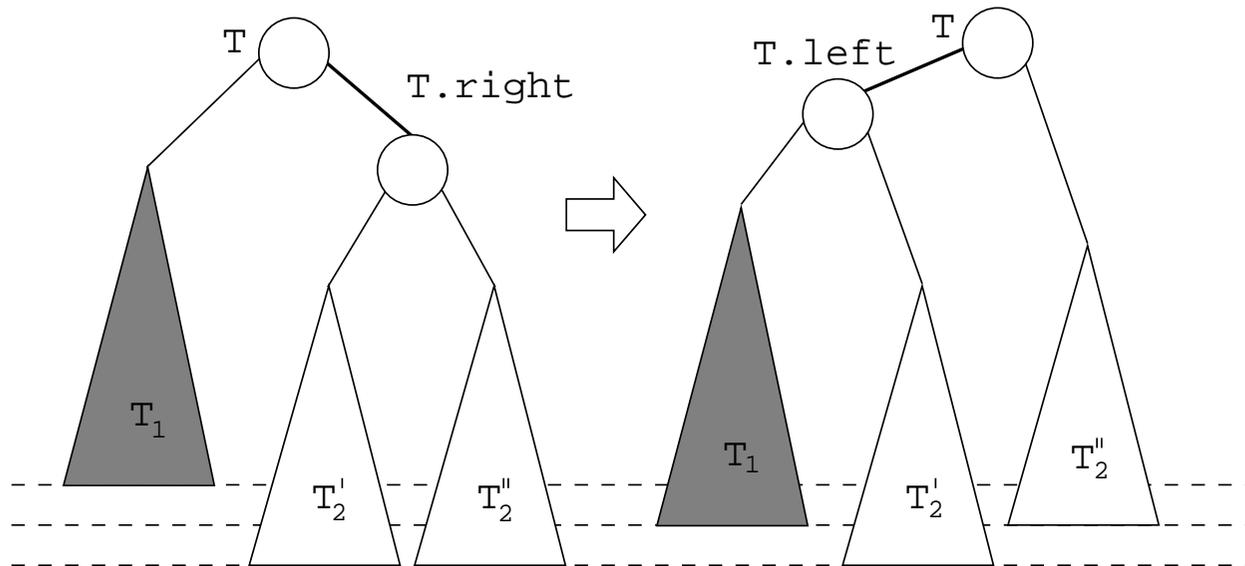
Fall RebDL-4.1: `T.right.bal = 0`; // Zwei gleich tiefe Unterbäume in `T.right`



**Linksrotation!**

```
T ← rotateL(T);  
T.left.bal ← 1;  
T.bal ← -1;  
shallower ← false;
```

Fall RebDL-4.1: `T.right.bal = 0`; // Zwei gleich tiefe Unterbäume in T.right

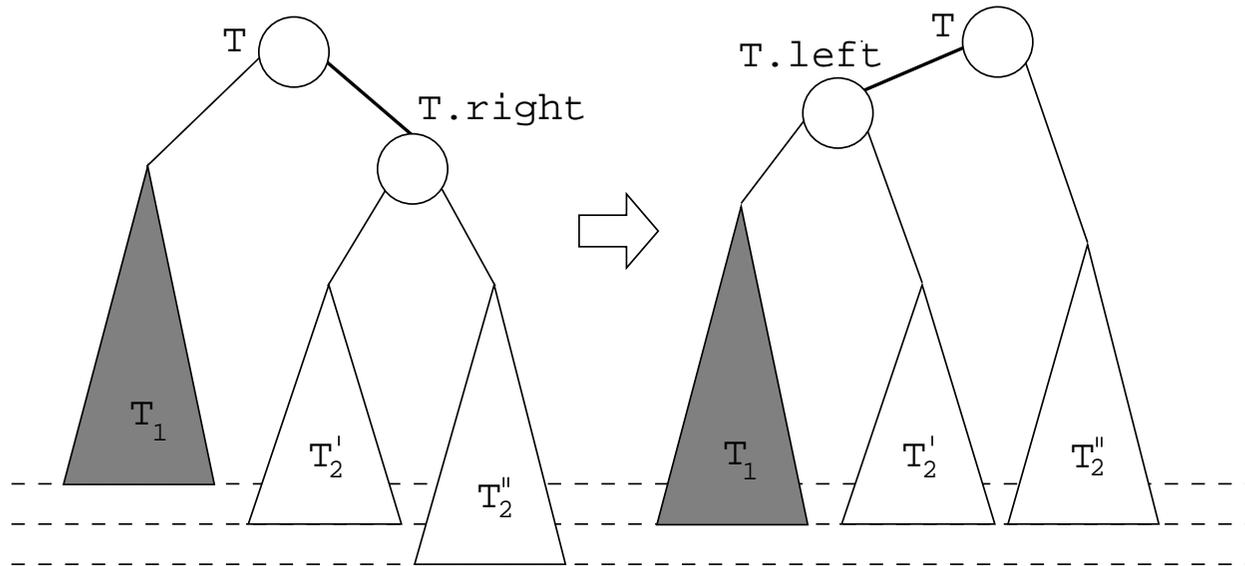


**Linksrotation!**

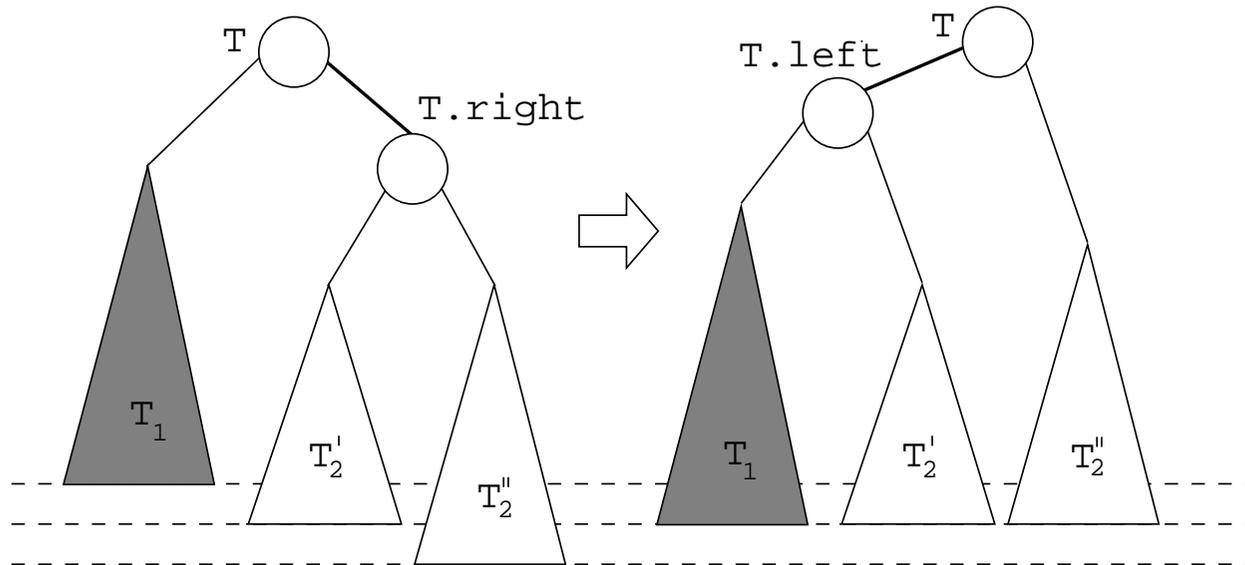
```
T ← rotateL(T);  
T.left.bal ← 1;  
T.bal ← -1;  
shallower ← false;
```

// Rebalancierung beendet.

Fall RebDL-4.2:  $T.right.bal = 1$ ; // der äußere Unterbaum in  $T.right$  ist tiefer



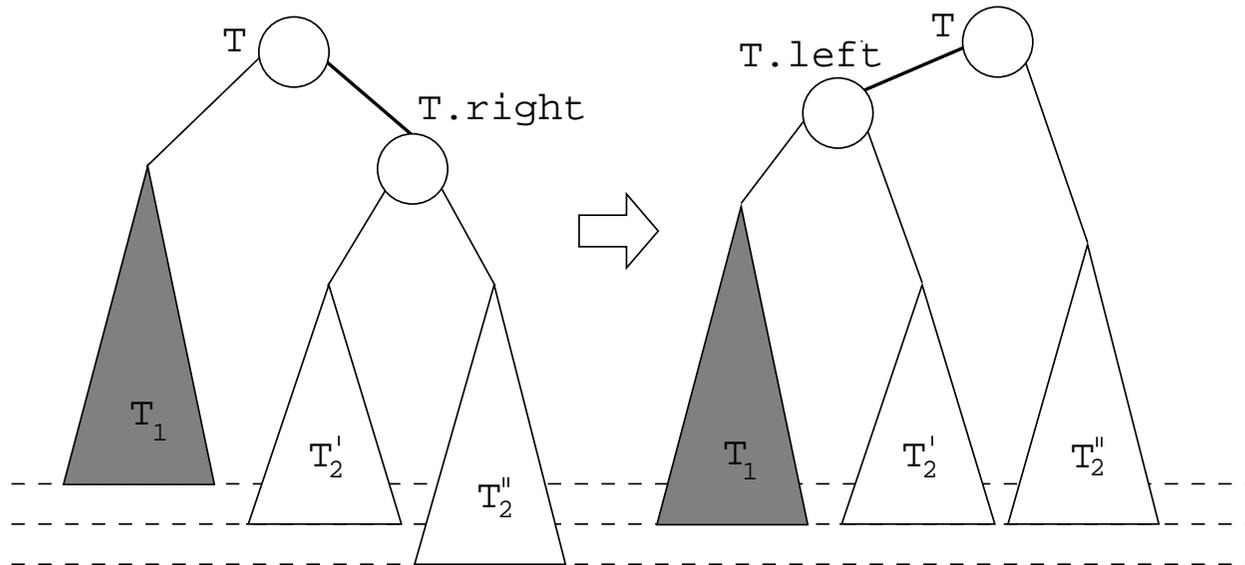
Fall RebDL-4.2: `T.right.bal = 1`; // der **äußere** Unterbaum in `T.right` ist tiefer



**Linksrotation!**

```
T ← rotateL(T);  
T.left.bal ← 0;  
T.bal ← 0;  
shallower ← true;
```

Fall RebDL-4.2: `T.right.bal = 1`; // der **äußere** Unterbaum in `T.right` ist tiefer

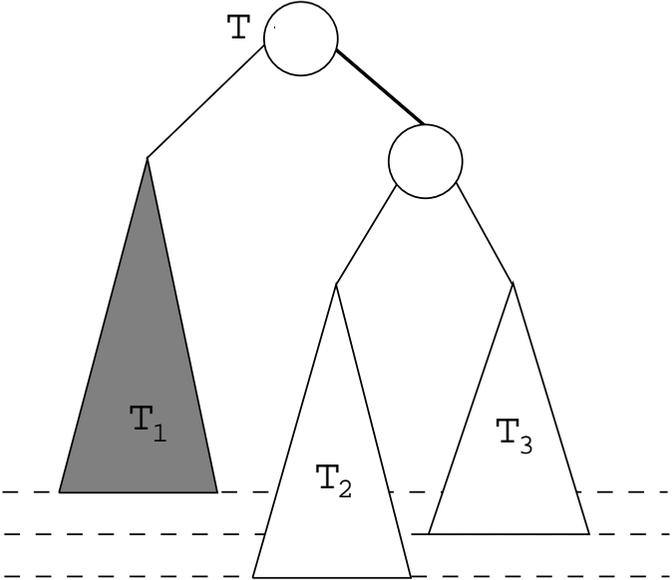


**Linksrotation!**

```
T ← rotateL(T);  
T.left.bal ← 0;  
T.bal ← 0;  
shallower ← true;
```

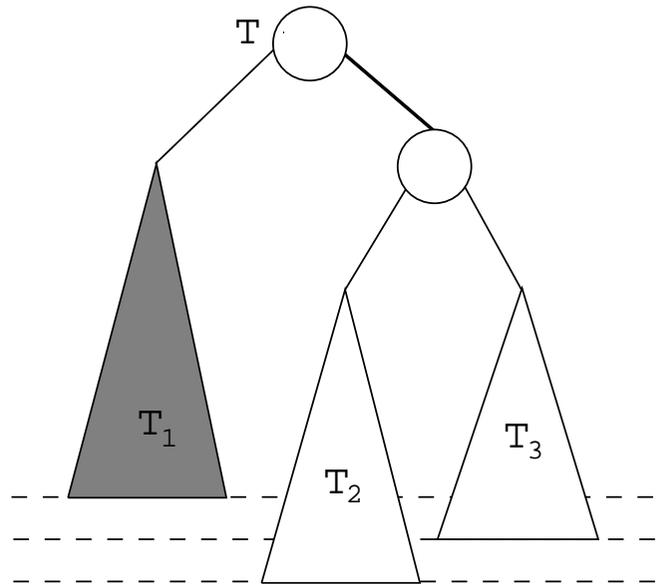
// Rebalancierung muss weitergehen!

Fall RebDL-4.3:  $T.right.bal = -1$ ; // der **innere** Unterbaum in  $T.right$  ist tiefer



---

Fall RebDL-4.3:  $T.\text{right}.bal = -1$ ; // der **innere** Unterbaum in  $T.\text{right}$  ist tiefer

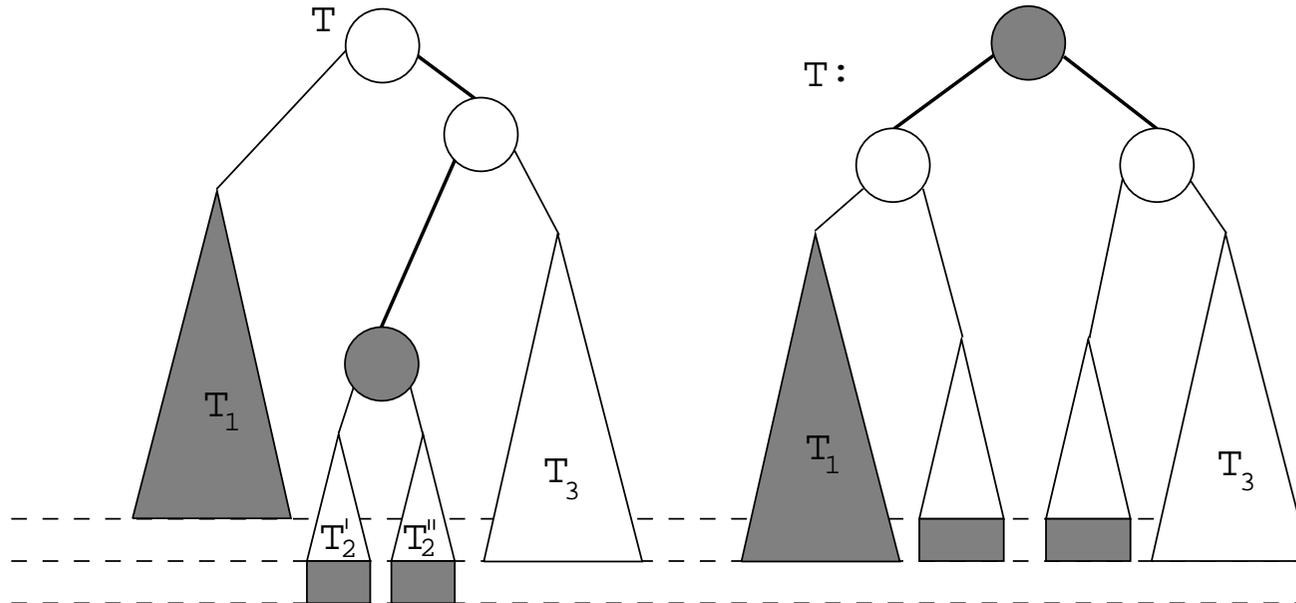


// Beobachte: Der zu tief hängende Teilbaum ist auf dem Weg „rechts-links“ zu erreichen.

// → **Doppelrotation!**

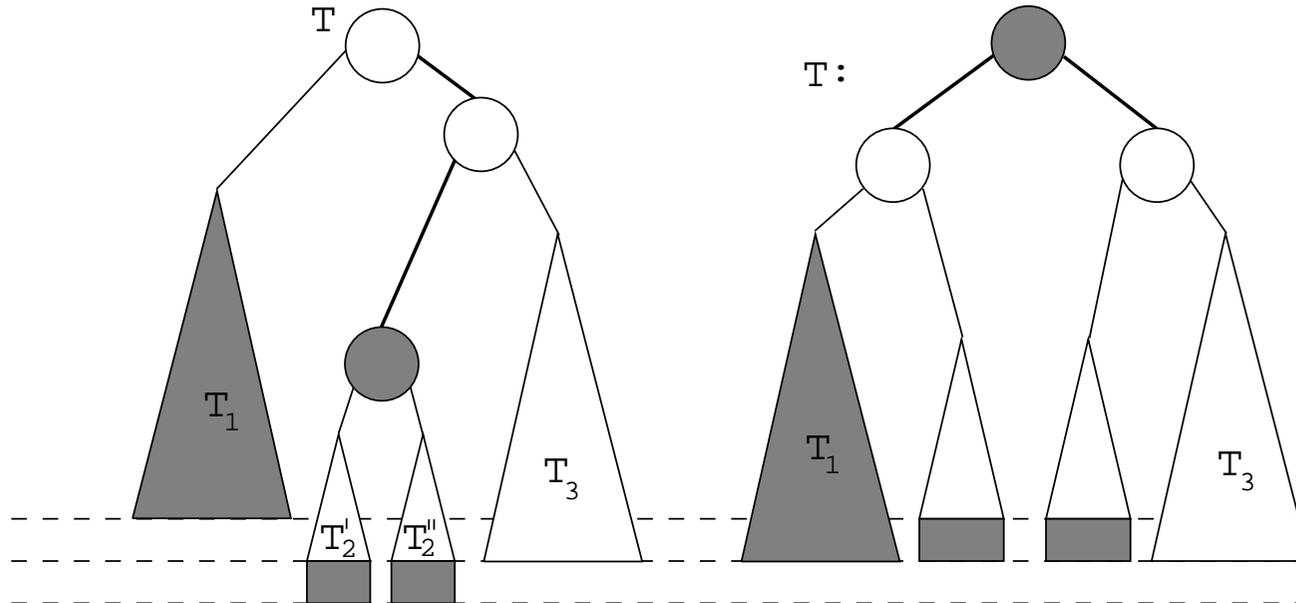
## Fall RebDL-4.3: (Forts.)

Betrachte Teilbäume von  $T_2 = T.\text{right}.\text{left}$ .



## Fall RebDL-4.3: (Forts.)

Betrachte Teilbäume von  $T_2 = T.\text{right}.\text{left}$ .

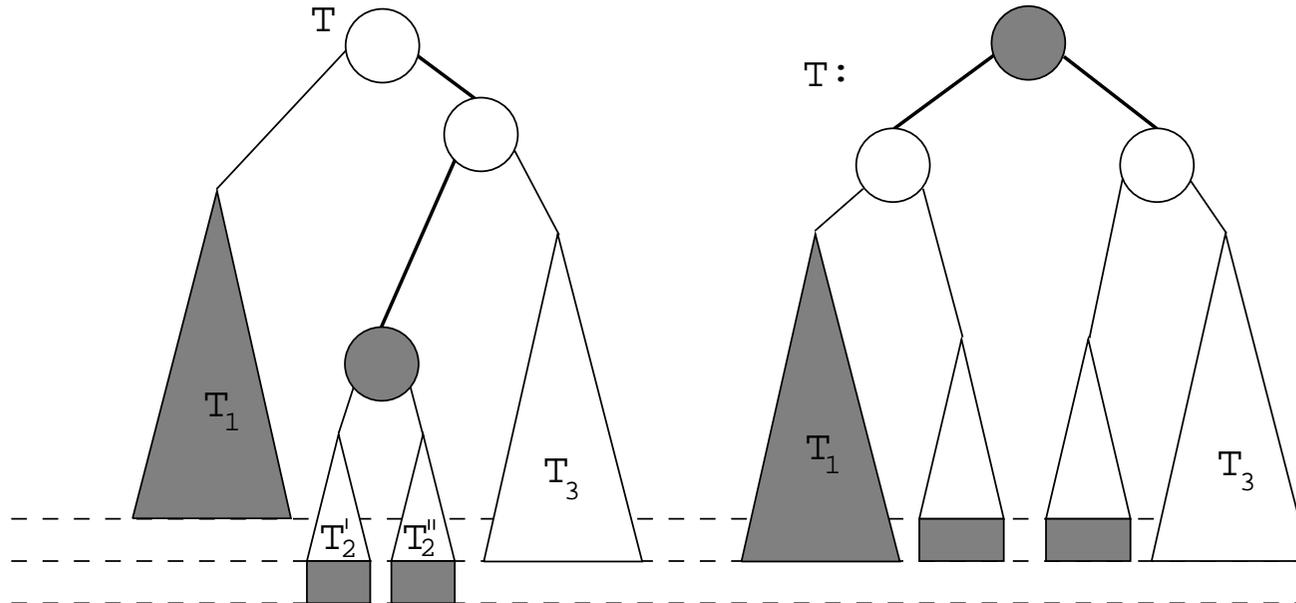


**Doppelrotation!**

```
T ← rotateRL(T);  
shallower ← true;
```

## Fall RebDL-4.3: (Forts.)

Betrachte Teilbäume von  $T_2 = T.\text{right}.\text{left}$ .



**Doppelrotation!**

```
T ← rotateRL(T);  
shallower ← true;
```

Neue Werte der Balancefaktoren:

---

Neue Werte der Balancefaktoren:

---

Neue Werte der Balancefaktoren:

altes	neues		
T.bal	T.left.bal	T.right.bal	T.bal
-1	0	1	0
0	0	0	0
1	-1	0	0

---

Neue Werte der Balancefaktoren:

altes	neues		
T.bal	T.left.bal	T.right.bal	T.bal
-1	0	1	0
0	0	0	0
1	-1	0	0

(Das alte T.bal enthält (nach der Doppelrotation) den Balancefaktor, der ursprünglich in der Wurzel des „mittleren“ Unterbaums  $T_2$  stand.)

---

Neue Werte der Balancefaktoren:

altes	neues		
T.bal	T.left.bal	T.right.bal	T.bal
-1	0	1	0
0	0	0	0
1	-1	0	0

(Das alte T.bal enthält (nach der Doppelrotation) den Balancefaktor, der ursprünglich in der Wurzel des „mittleren“ Unterbaums  $T_2$  stand.)

**Ende des nicht prüfungsrelevanten Lösch-Programms.**

---

Neue Werte der Balancefaktoren:

altes T.bal	neues		
	T.left.bal	T.right.bal	T.bal
-1	0	1	0
0	0	0	0
1	-1	0	0

(Das alte T.bal enthält (nach der Doppelrotation) den Balancefaktor, der ursprünglich in der Wurzel des „mittleren“ Unterbaums  $T_2$  stand.)

**Ende des nicht prüfungsrelevanten Lösch-Programms.**

**Achtung:** Man muss Löschungen an Beispielen durchführen können!

---

### Proposition 4.3.6

Die rekursive Prozedur  $AVL\_delete(T, x)$  führt die Wörterbuchoperation *delete* korrekt durch.

---

### Proposition 4.3.6

Die rekursive Prozedur  $AVL\_delete(T, x)$  führt die Wörterbuchoperation *delete* korrekt durch. D. h.: es entsteht ein AVL-Baum für das Wörterbuch  $delete(f_T, x)$ .

---

### Proposition 4.3.6

Die rekursive Prozedur  $AVL\_delete(T, x)$  führt die Wörterbuchoperation *delete* korrekt durch. D. h.: es entsteht ein AVL-Baum für das Wörterbuch  $delete(f_T, x)$ .

Die Prozedur hat Laufzeit  $O(\log n)$

---

### Proposition 4.3.6

Die rekursive Prozedur  $AVL\_delete(T, x)$  führt die Wörterbuchoperation *delete* korrekt durch. D. h.: es entsteht ein AVL-Baum für das Wörterbuch  $delete(f_T, x)$ .

Die Prozedur hat Laufzeit  $O(\log n)$  und führt **an jedem Knoten auf dem Weg** vom gelöschten Knoten zur Wurzel höchstens eine Einfach- oder Doppelrotation durch.

---

### Proposition 4.3.6

Die rekursive Prozedur  $AVL\_delete(T, x)$  führt die Wörterbuchoperation *delete* korrekt durch. D. h.: es entsteht ein AVL-Baum für das Wörterbuch  $delete(f_T, x)$ .

Die Prozedur hat Laufzeit  $O(\log n)$  und führt **an jedem Knoten auf dem Weg** vom gelöschten Knoten zur Wurzel höchstens eine Einfach- oder Doppelrotation durch.

### Satz 4.3.7

In **AVL-Bäumen** kostet jede **Wörterbuchoperation** Zeit

$$O(\log n),$$

wenn  $n$  die aktuelle Anzahl der Einträge ist.

**Vorlesungsvideo:**

**2-3-Bäume**

---

## 4.4 Mehrweg-Suchbäume

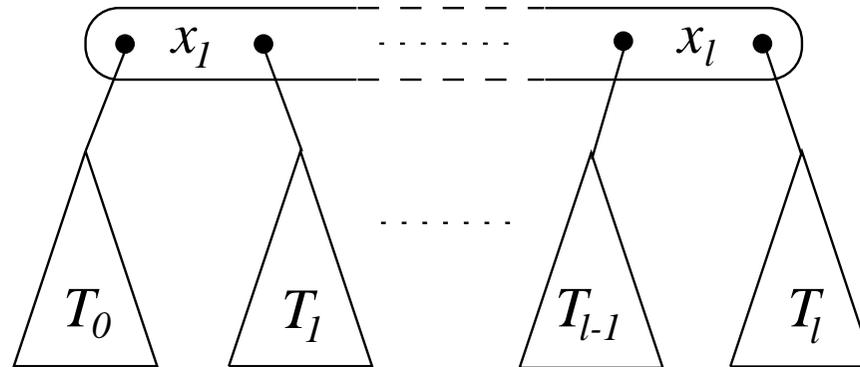
Bäume aus Knoten mit variablem (Ausgangs-)Grad.

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüssel und  $l + 1 \geq 2$  Unterbäumen:

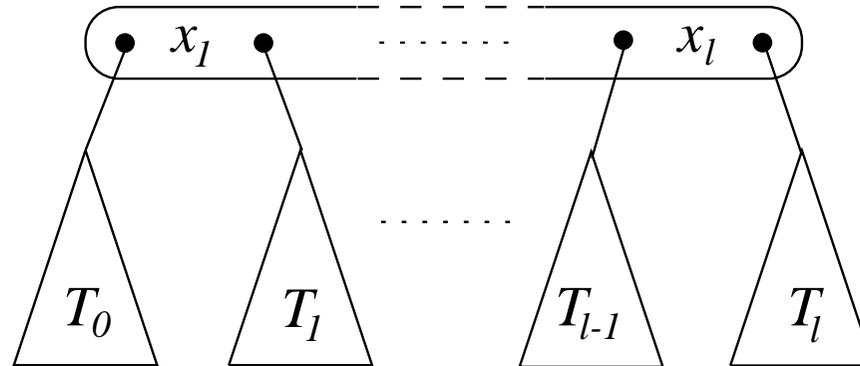


---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



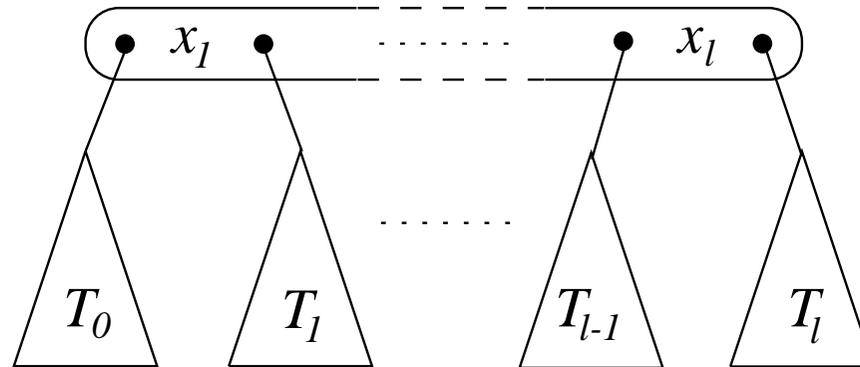
$l$  Schlüssel:  $x_1 < \dots < x_l$ .

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



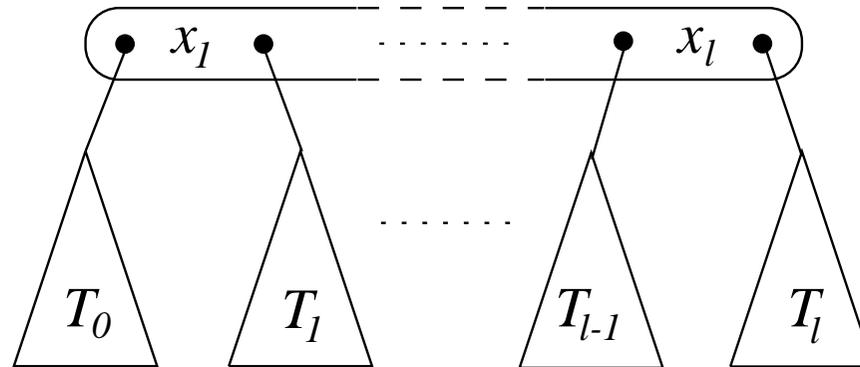
$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

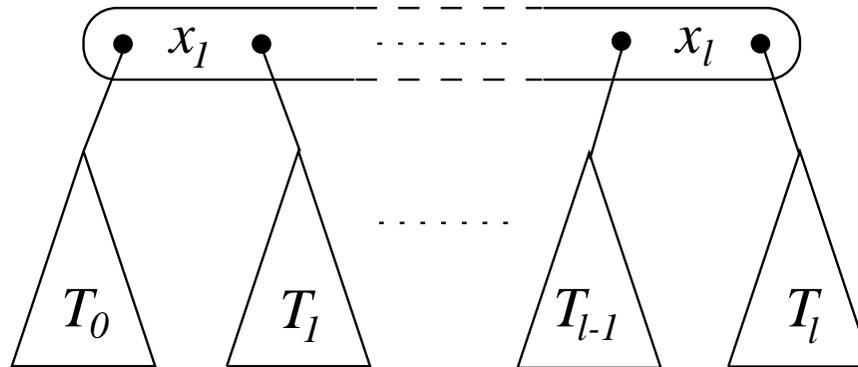
Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

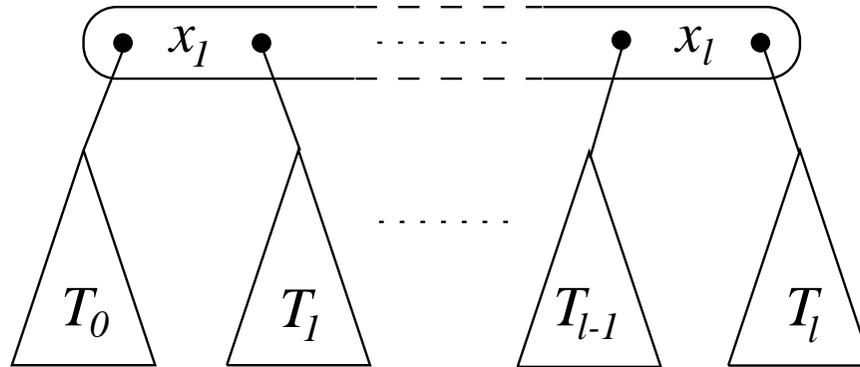
Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:  $y_0$

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

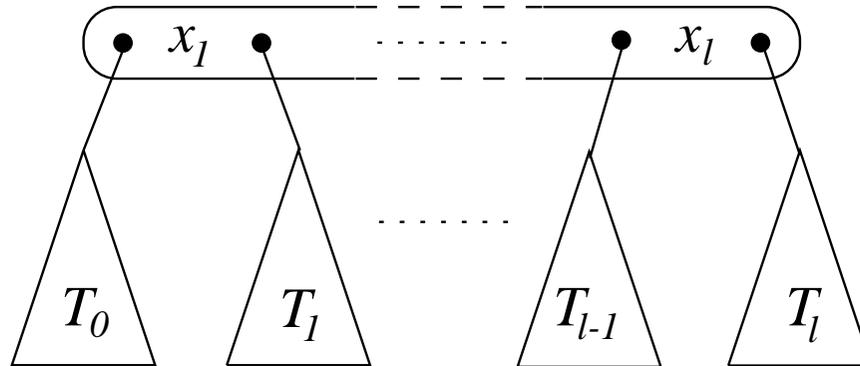
Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:  $y_0 < x_1$

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

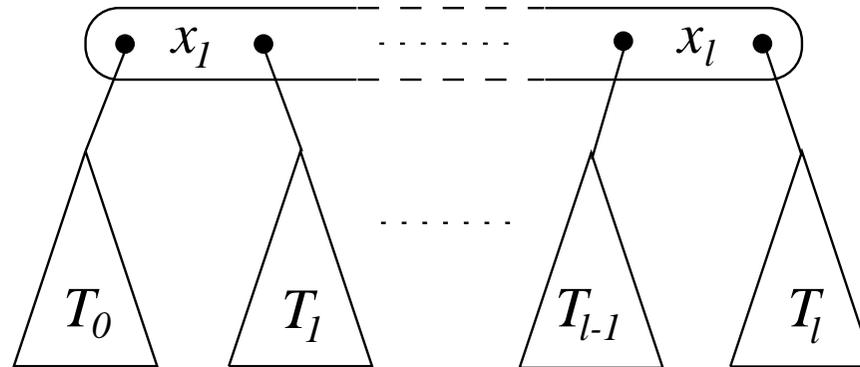
Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:  $y_0 < x_1 < y_1$

---

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



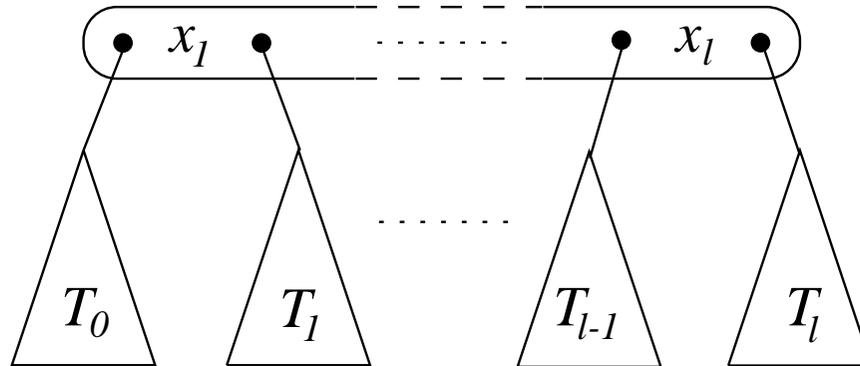
$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:  $y_0 < x_1 < y_1 < x_2$

## 4.4 Mehrweg-Suchbäume

Bäume aus Knoten mit variablem (Ausgangs-)Grad.

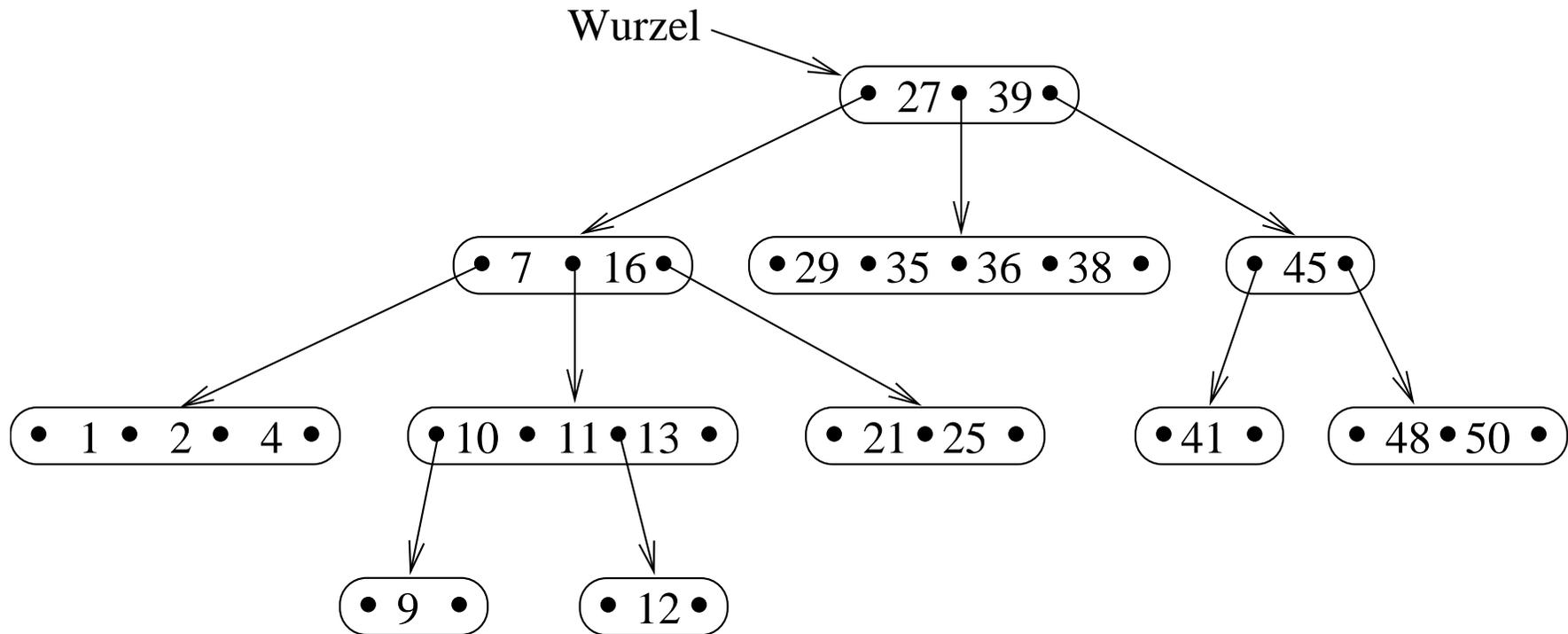
Ein Knoten mit  $l \geq 1$  Schlüsseln und  $l + 1 \geq 2$  Unterbäumen:



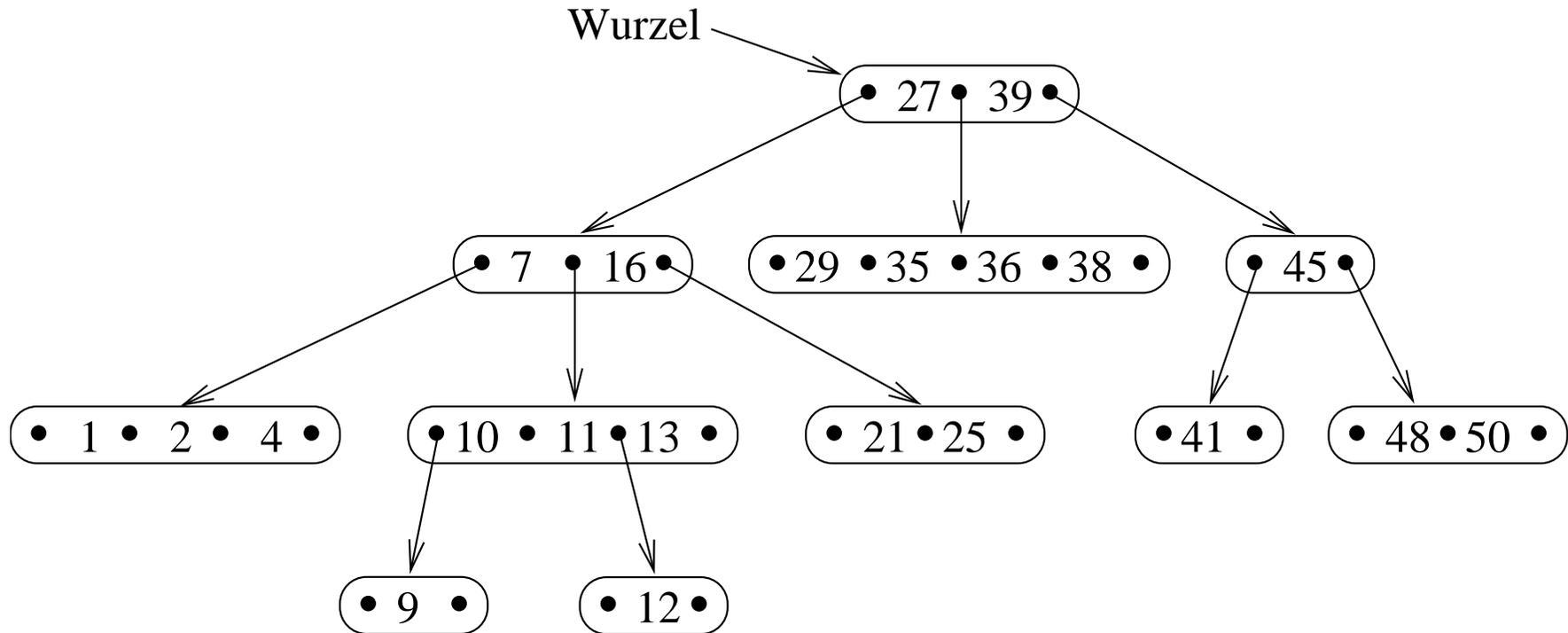
$l$  Schlüssel:  $x_1 < \dots < x_l$ .  $l + 1$  Unterbäume:  $T_0, \dots, T_l$ .

Für  $y_i$  in  $T_i$ ,  $0 \leq i \leq l$  gilt:  $y_0 < x_1 < y_1 < x_2 < \dots < y_{l-1} < x_l < y_l$

# Beispiel eines $\mathbb{N}$ -MwSB:

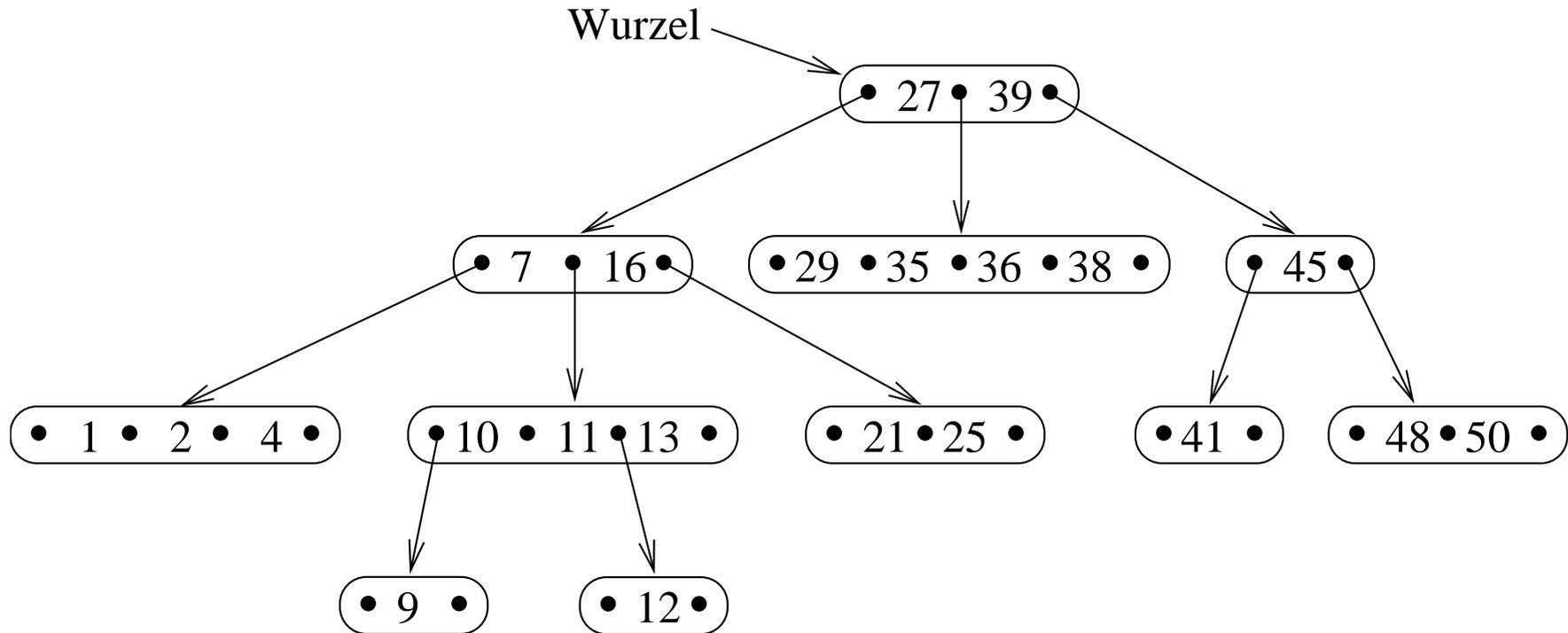


Beispiel eines  $\mathbb{N}$ -MwSB:



„•“ ohne Pfeil steht für einen Zeiger auf einen leeren Baum  $\square$ .

Beispiel eines  $\mathbb{N}$ -MwSB:



„•“ ohne Pfeil steht für einen Zeiger auf einen leeren Baum  $\square$ .

Man überprüfe die Anordnungsbedingung in diesem Beispiel!

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

(0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

- (0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.
- (1) Ist  $l \geq 1$

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

- (0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.
- (1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

- (0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.
- (1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$  und sind  $T_0, T_1, \dots, T_l$   $U$ -MwSBe mit:

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

(0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.

(1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$  und sind  $T_0, T_1, \dots, T_l$   $U$ -MwSBe mit:

$$y_0 < x_1 < y_1 < x_2 < \dots < y_{l-1} < x_l < y_l, \text{ für alle } y_i \text{ in } T_i, 0 \leq i \leq l,$$

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

(0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.

(1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$  und sind  $T_0, T_1, \dots, T_l$   $U$ -MwSBe mit:

$$y_0 < x_1 < y_1 < x_2 < \dots < y_{l-1} < x_l < y_l, \text{ für alle } y_i \text{ in } T_i, 0 \leq i \leq l,$$

dann ist auch  $(T_0, x_1, T_1, x_2, \dots, x_{l-1}, T_{l-1}, x_l, T_l)$  ein  $U$ -Mehrweg-Suchbaum.

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

(0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.

(1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$  und sind  $T_0, T_1, \dots, T_l$   $U$ -MwSBe mit:

$$y_0 < x_1 < y_1 < x_2 < \dots < y_{l-1} < x_l < y_l, \text{ für alle } y_i \text{ in } T_i, 0 \leq i \leq l,$$

dann ist auch  $(T_0, x_1, T_1, x_2, \dots, x_{l-1}, T_{l-1}, x_l, T_l)$  ein  $U$ -Mehrweg-Suchbaum.

Analog:  $(U, R)$ -Mehrweg-Suchbäume, mit Menge  $R$  von Datensätzen.

---

## Definition 4.4.1

**Mehrweg-Suchbäume (MwSBe)** über dem Universum  $(U, <)$  sind wie folgt **induktiv definiert**:

(0) Der leere Baum  $\square$  ist ein  $U$ -MwSB.

(1) Ist  $l \geq 1$  und sind  $x_1 < x_2 < \dots < x_l$  Schlüssel in  $U$  und sind  $T_0, T_1, \dots, T_l$   $U$ -MwSBe mit:

$$y_0 < x_1 < y_1 < x_2 < \dots < y_{l-1} < x_l < y_l, \text{ für alle } y_i \text{ in } T_i, 0 \leq i \leq l,$$

dann ist auch  $(T_0, x_1, T_1, x_2, \dots, x_{l-1}, T_{l-1}, x_l, T_l)$  ein  $U$ -Mehrweg-Suchbaum.

Analog:  $(U, R)$ -Mehrweg-Suchbäume, mit Menge  $R$  von Datensätzen.

Knotenformat ist dann:  $(T_0, (x_1, r_1), T_1, (x_2, r_2), \dots, (x_{l-1}, r_{l-1}), T_{l-1}, (x_l, r_l), T_l)$ .

---

Rekursive **Suche** in einem Mehrwegsuchbaum  $T$  nach Schlüssel  $x$  aus  $U$ .

---

Rekursive **Suche** in einem Mehrwegsuchbaum  $T$  nach Schlüssel  $x$  aus  $U$ .

(Abstrakt, aber auch die Implementierungen arbeiten nach diesem Muster.)

---

Rekursive **Suche** in einem Mehrwegsuchbaum  $T$  nach Schlüssel  $x$  aus  $U$ .

(Abstrakt, aber auch die Implementierungen arbeiten nach diesem Muster.)

Wenn  $T = \square$ , dann gibt es keinen Eintrag mit Schlüssel  $x$ .

---

Rekursive **Suche** in einem Mehrwegsuchbaum  $T$  nach Schlüssel  $x$  aus  $U$ .

(Abstrakt, aber auch die Implementierungen arbeiten nach diesem Muster.)

Wenn  $T = \square$ , dann gibt es keinen Eintrag mit Schlüssel  $x$ .

Wenn  $T = (T_0, x_1, T_1, x_2, \dots, x_{l-1}, T_{l-1}, x_l, T_l)$ , dann:

- Falls  $x = x_i$  für ein  $i$ , dann sitzt Eintrag mit Schlüssel  $x$  im Wurzelknoten von  $T$ .

---

Rekursive **Suche** in einem Mehrwegsuchbaum  $T$  nach Schlüssel  $x$  aus  $U$ .

(Abstrakt, aber auch die Implementierungen arbeiten nach diesem Muster.)

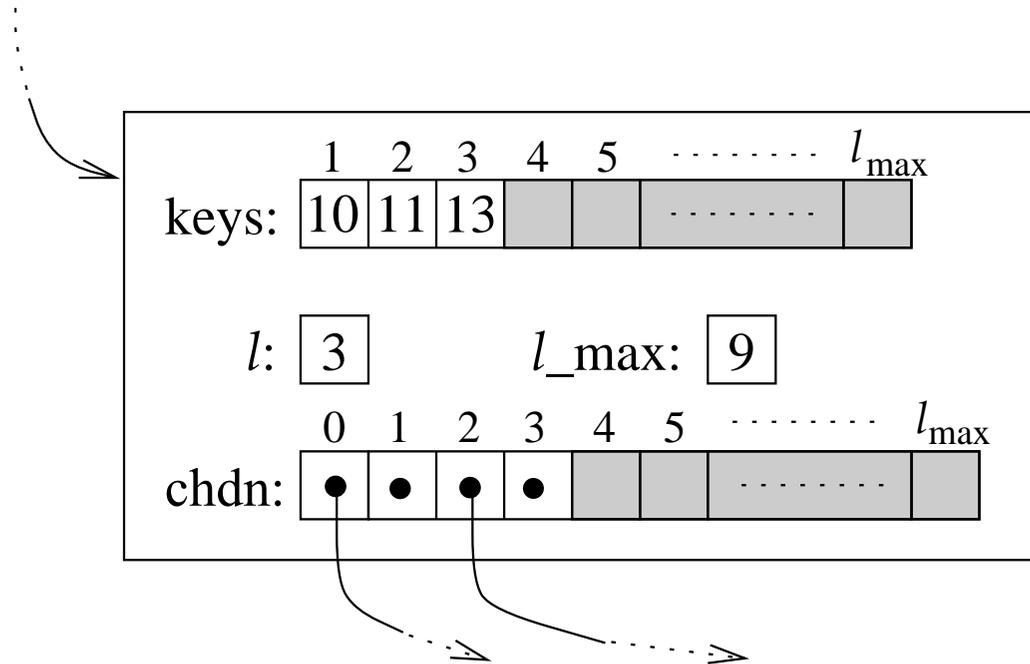
Wenn  $T = \square$ , dann gibt es keinen Eintrag mit Schlüssel  $x$ .

Wenn  $T = (T_0, x_1, T_1, x_2, \dots, x_{l-1}, T_{l-1}, x_l, T_l)$ , dann:

- Falls  $x = x_i$  für ein  $i$ , dann sitzt Eintrag mit Schlüssel  $x$  im Wurzelknoten von  $T$ .
- Sonst: Bestimme das eindeutige  $i$  mit  $x_i < x < x_{i+1}$  und suche rekursiv in  $T_i$ .  
(Dabei sollen  $x_0$  und  $x_{n+1}$  die „virtuellen Schlüssel“  $-\infty$  bzw.  $+\infty$  sein.)

Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 1:** Jeder Knoten enthält zwei Arrays (oder „vectors“), *keys* und *chdn* (für „children“), bzw. noch ein Array *data* für die Daten zu den Schlüsseln.



---

Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 2:** Schlüssel und Kinder als Listen *keys* und *chdn*.

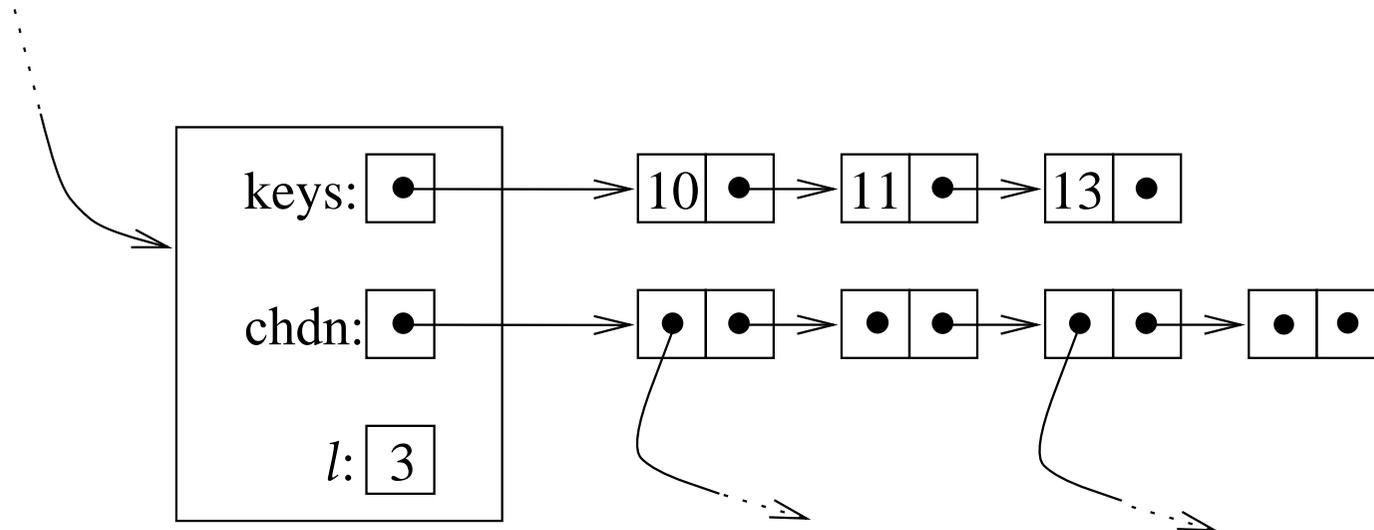
(Daten sind gegebenenfalls in die Schlüsselliste zu integrieren.)

---

Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 2:** Schlüssel und Kinder als Listen *keys* und *chdn*.

(Daten sind gegebenenfalls in die Schlüsselliste zu integrieren.)

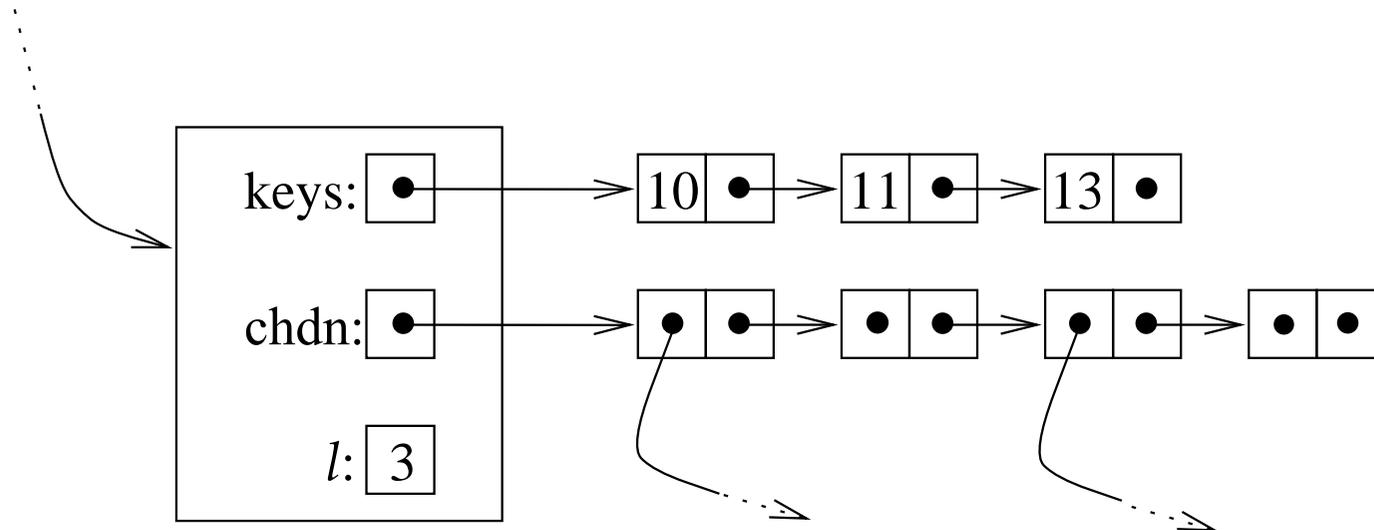


---

Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 2:** Schlüssel und Kinder als Listen *keys* und *chdn*.

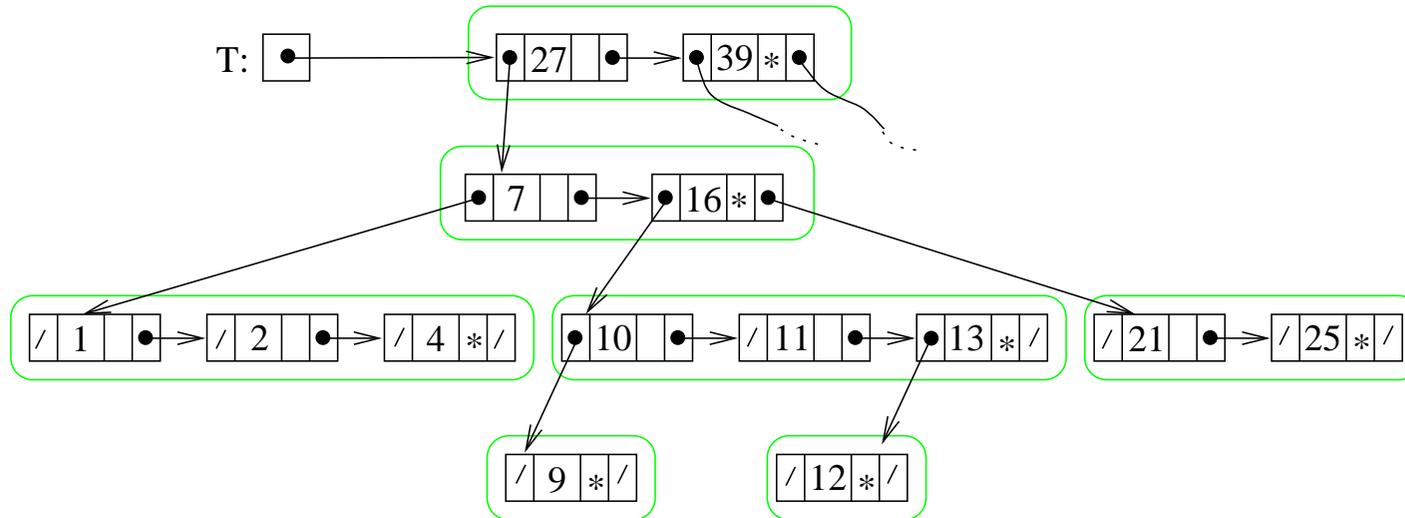
(Daten sind gegebenenfalls in die Schlüsselliste zu integrieren.)



Vorteil: Kein Platzverbrauch für nicht existierende Schlüssel/Unterbäume.

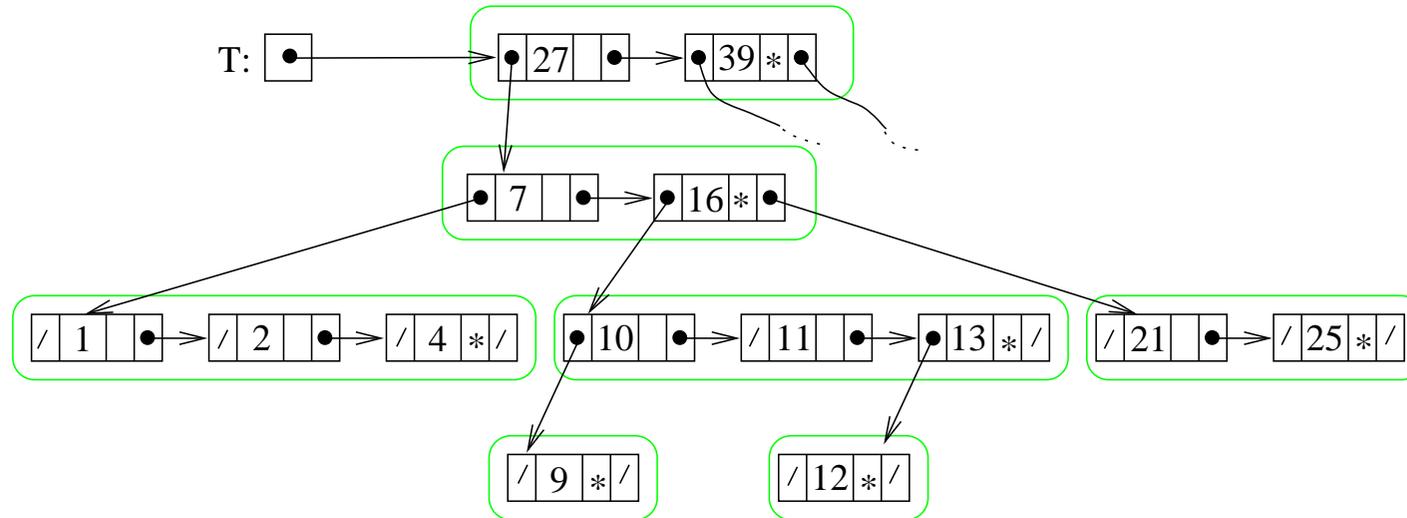
Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 3:** Ein MwSB-Knoten wird durch eine aufsteigend sortierte lineare Liste von **Binärbaumknoten** dargestellt.



Implementierung der Knoten in einem Mehrwegsuchbaum:

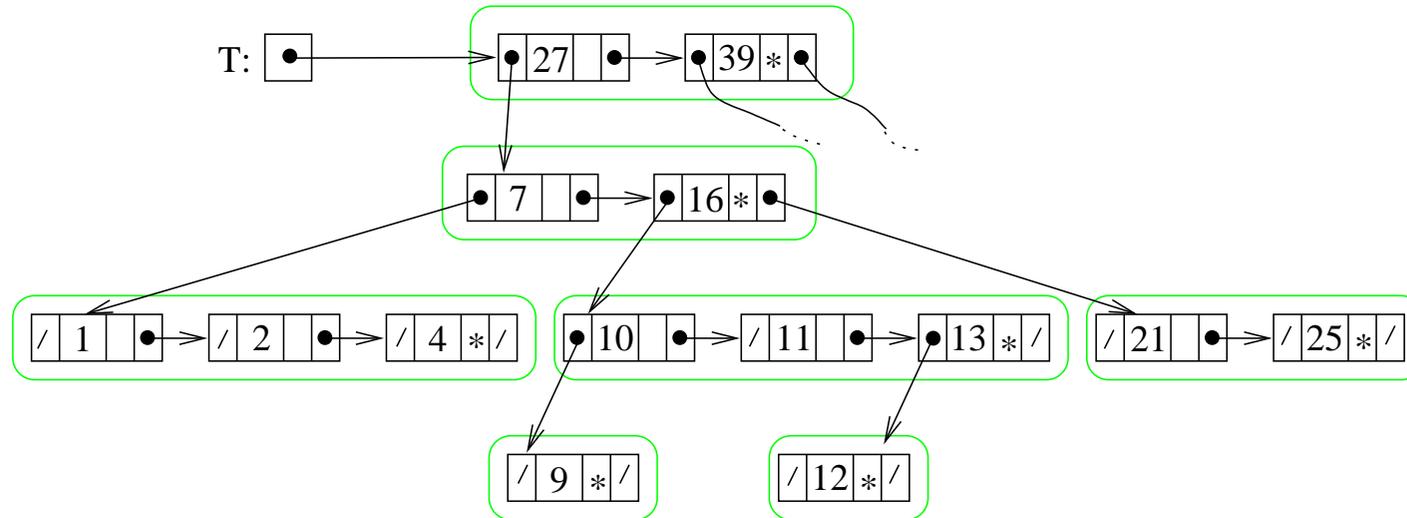
**Variante 3:** Ein MwSB-Knoten wird durch eine aufsteigend sortierte lineare Liste von **Binärbaumknoten** dargestellt.



Jeder Knoten hat Platz für Schlüssel, (Daten,) Zeiger auf Unterbaum, Zeiger auf nächsten in Liste. Im Knoten muss als Boolescher Wert vermerkt sein, ob es sich um den Knoten am Listenende handelt. (Im Bild: „\*“.)

Implementierung der Knoten in einem Mehrwegsuchbaum:

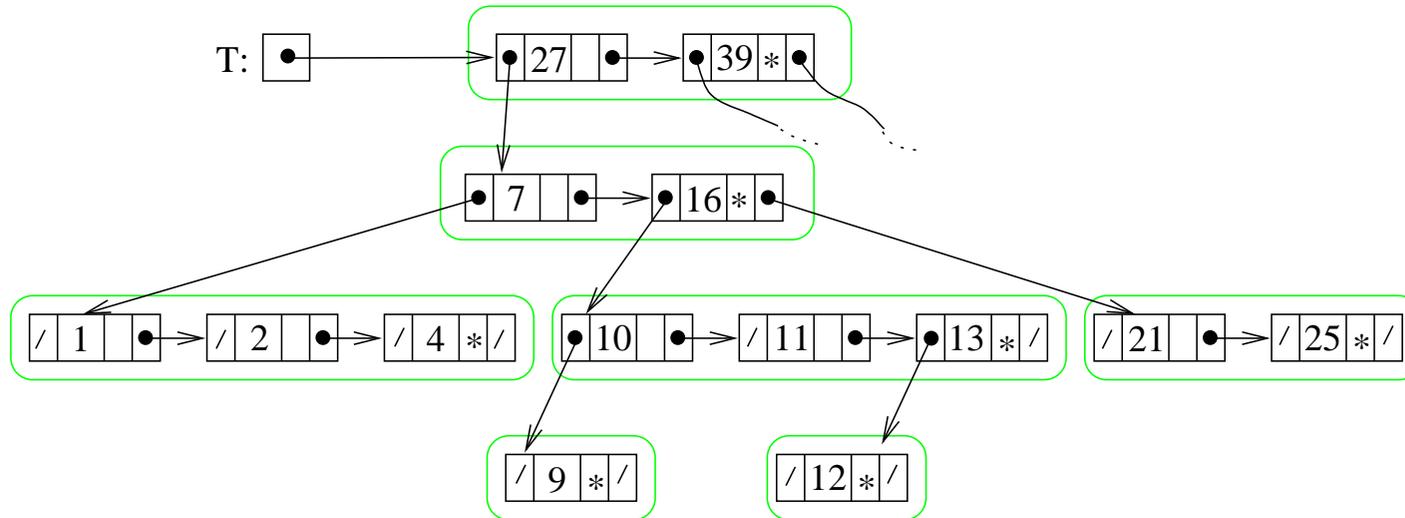
**Variante 3:** Ein MwSB-Knoten wird durch eine aufsteigend sortierte lineare Liste von **Binärbaumknoten** dargestellt.



Jeder Knoten hat Platz für Schlüssel, (Daten,) Zeiger auf Unterbaum, Zeiger auf nächsten in Liste. Im Knoten muss als Boolescher Wert vermerkt sein, ob es sich um den Knoten am Listenende handelt. (Im Bild: „\*“.) Ein solcher Knoten hat dann zwei Zeiger auf Unterbäume.

Implementierung der Knoten in einem Mehrwegsuchbaum:

**Variante 3:** Ein MwSB-Knoten wird durch eine aufsteigend sortierte lineare Liste von **Binärbaumknoten** dargestellt.



Jeder Knoten hat Platz für Schlüssel, (Daten,) Zeiger auf Unterbaum, Zeiger auf nächsten in Liste. Im Knoten muss als Boolescher Wert vermerkt sein, ob es sich um den Knoten am Listenende handelt. (Im Bild: „\*“.) Ein solcher Knoten hat dann zwei Zeiger auf Unterbäume.

**Elegant:** Für die Suche kann man die gewöhnliche Suchprozedur für binäre Suchbäume benutzen.

---

## Spezialfall: 2-3-Bäume

### Definition 4.4.2

Ein Mehrweg-Suchbaum  $T$  heißt ein **2-3-Baum**, wenn gilt:

---

## Spezialfall: 2-3-Bäume

### Definition 4.4.2

Ein Mehrweg-Suchbaum  $T$  heißt ein **2-3-Baum**, wenn gilt:

- (a) Jeder Knoten enthält 1 oder 2 Schlüssel  
(also hat jeder Knoten **2** oder **3** Unterbäume, was der Struktur den Namen gibt);

---

## Spezialfall: 2-3-Bäume

### Definition 4.4.2

Ein Mehrweg-Suchbaum  $T$  heißt ein **2-3-Baum**, wenn gilt:

- (a) Jeder Knoten enthält 1 oder 2 Schlüssel  
(also hat jeder Knoten **2** oder **3** Unterbäume, was der Struktur den Namen gibt);
- (b) Für jeden Knoten  $v$  in  $T$  gilt: Wenn  $v$  **einen** leeren Unterbaum hat, so sind **alle** Unterbäume unter  $v$  leer, d.h.  $v$  ist dann **Blatt**;

---

## Spezialfall: 2-3-Bäume

### Definition 4.4.2

Ein Mehrweg-Suchbaum  $T$  heißt ein **2-3-Baum**, wenn gilt:

- (a) Jeder Knoten enthält 1 oder 2 Schlüssel  
(also hat jeder Knoten **2** oder **3** Unterbäume, was der Struktur den Namen gibt);
- (b) Für jeden Knoten  $v$  in  $T$  gilt: Wenn  $v$  **einen** leeren Unterbaum hat, so sind **alle** Unterbäume unter  $v$  leer, d.h.  $v$  ist dann **Blatt**;
- (c) Alle Blätter von  $T$  haben dieselbe Tiefe.

---

*Beispiele:*

---

*Beispiele:*

Leerer 2-3-Baum:

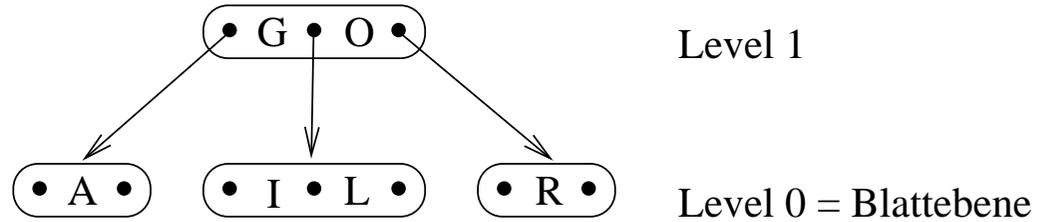


---

*Beispiele:*

Leerer 2-3-Baum: □

2-3-Baum mit zwei Ebenen:

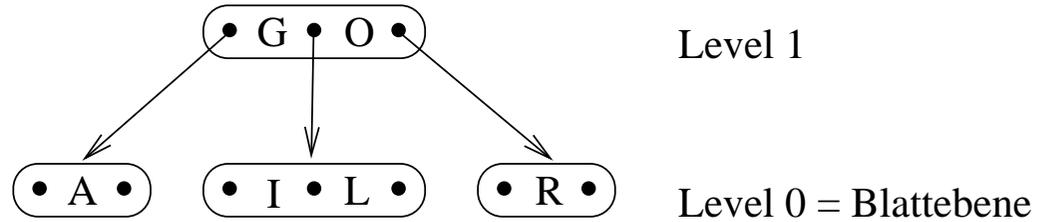


---

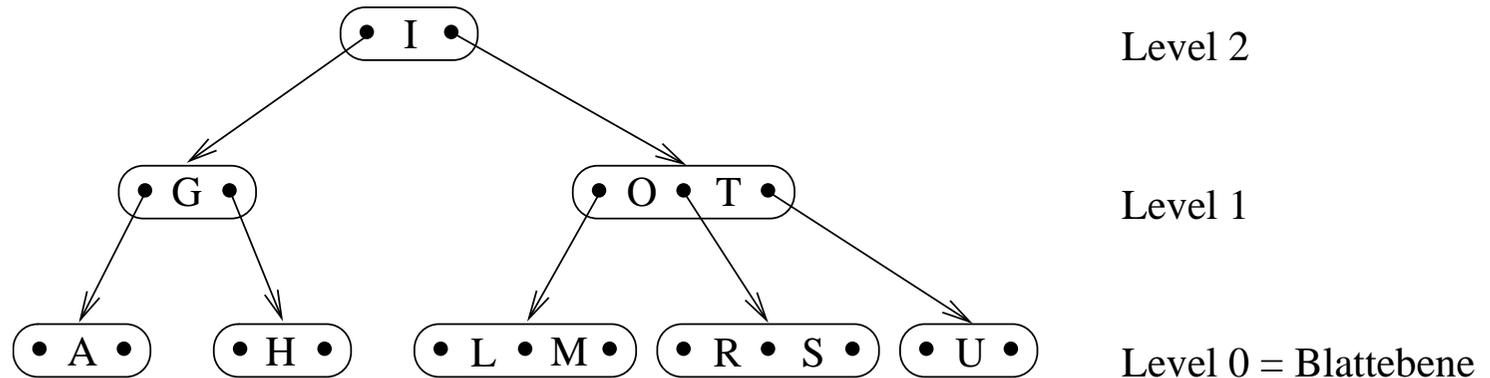
*Beispiele:*

Leerer 2-3-Baum: □

2-3-Baum mit zwei Ebenen:



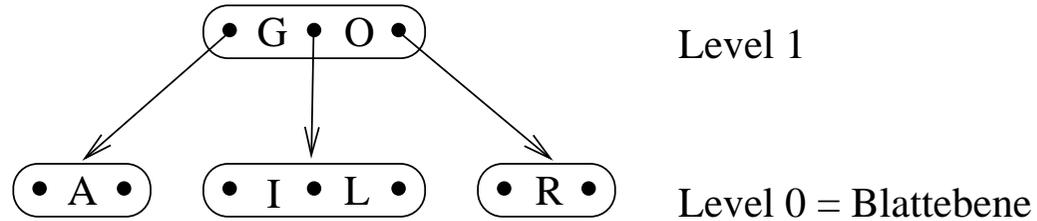
2-3-Baum mit drei Ebenen:



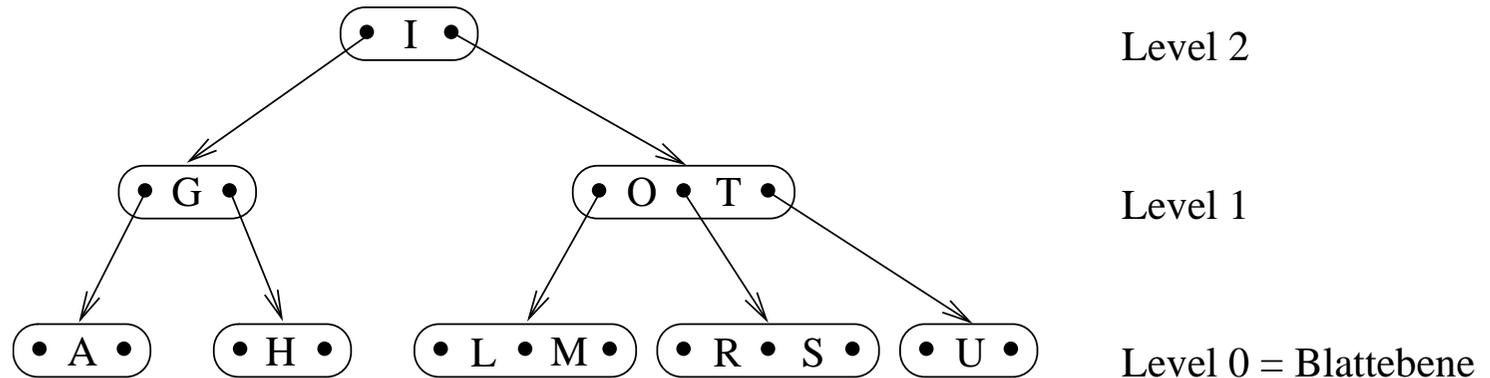
*Beispiele:*

Leerer 2-3-Baum: □

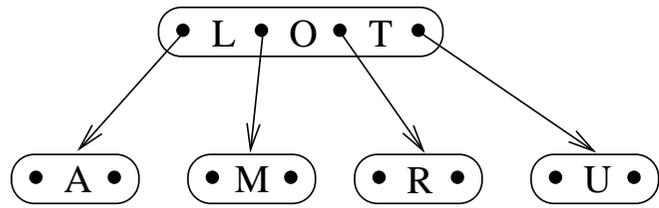
2-3-Baum mit zwei Ebenen:

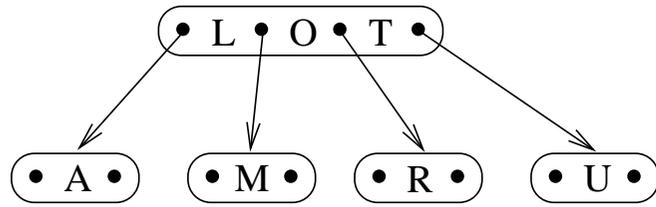


2-3-Baum mit drei Ebenen:

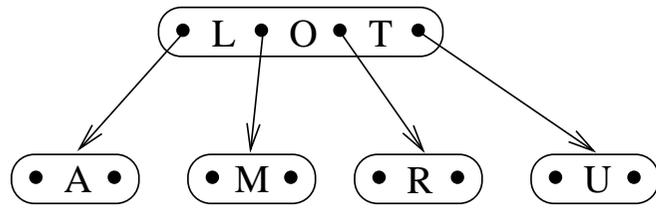


**NB:** Die Ebenen werden von den Blättern her nummeriert.

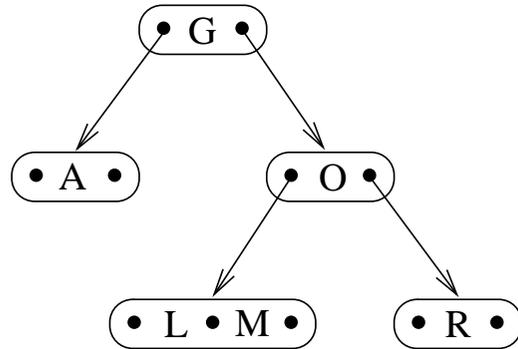


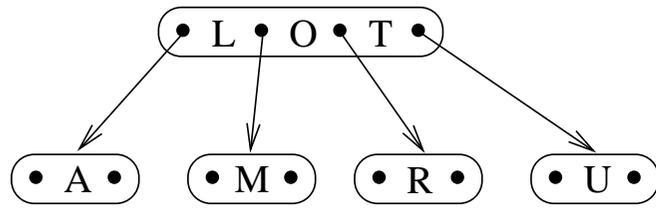


Kein 2-3-Baum (zu großer Grad).

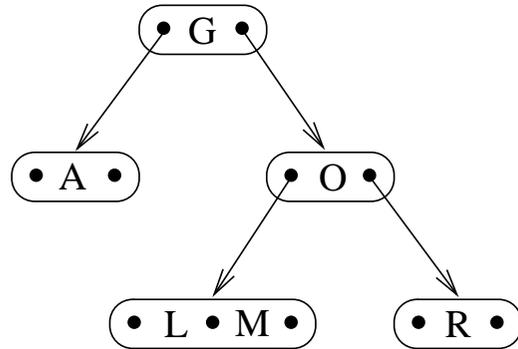


Kein 2-3-Baum (zu großer Grad).

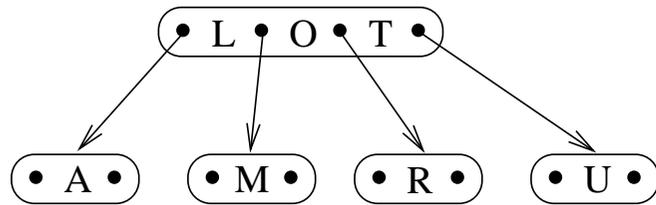




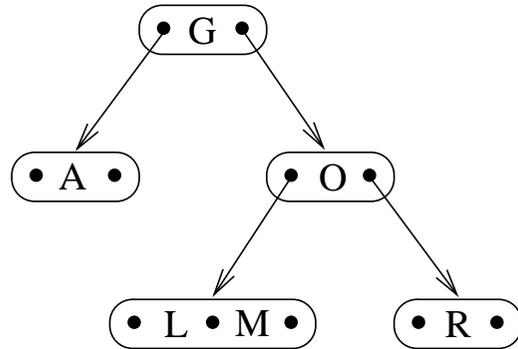
Kein 2-3-Baum (zu großer Grad).



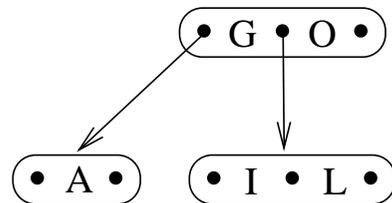
Kein 2-3-Baum (Blätter in verschiedenen Tiefen).

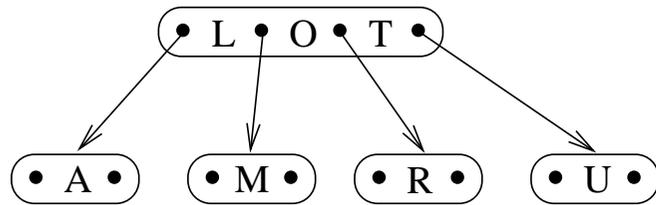


Kein 2-3-Baum (zu großer Grad).

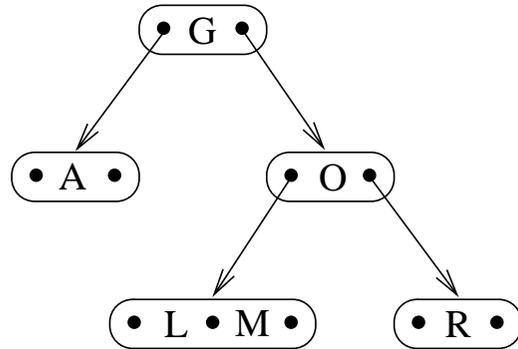


Kein 2-3-Baum (Blätter in verschiedenen Tiefen).

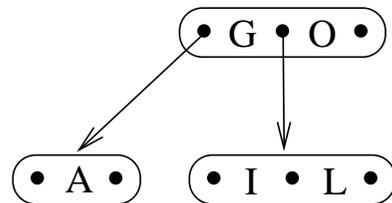




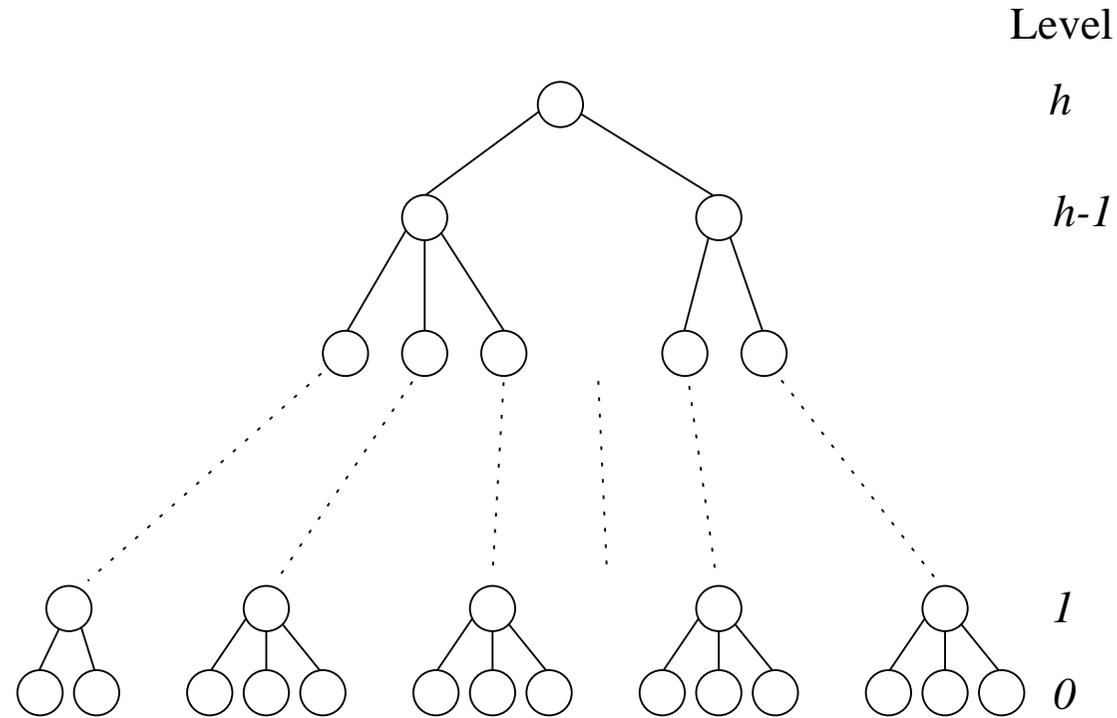
Kein 2-3-Baum (zu großer Grad).

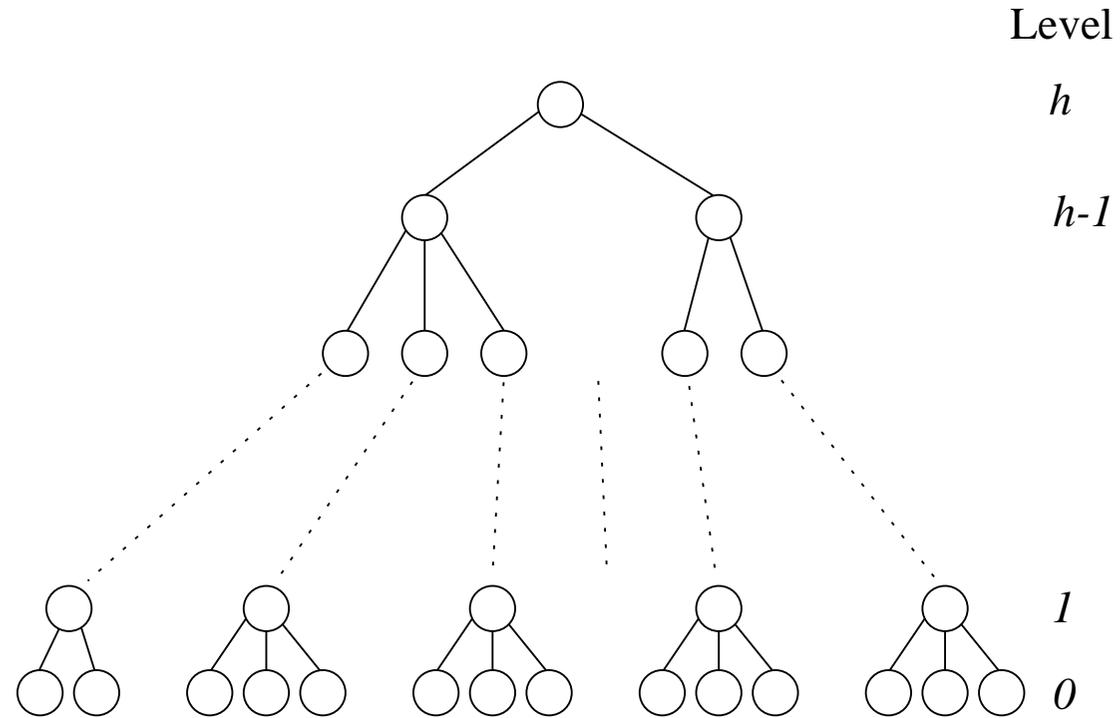


Kein 2-3-Baum (Blätter in verschiedenen Tiefen).

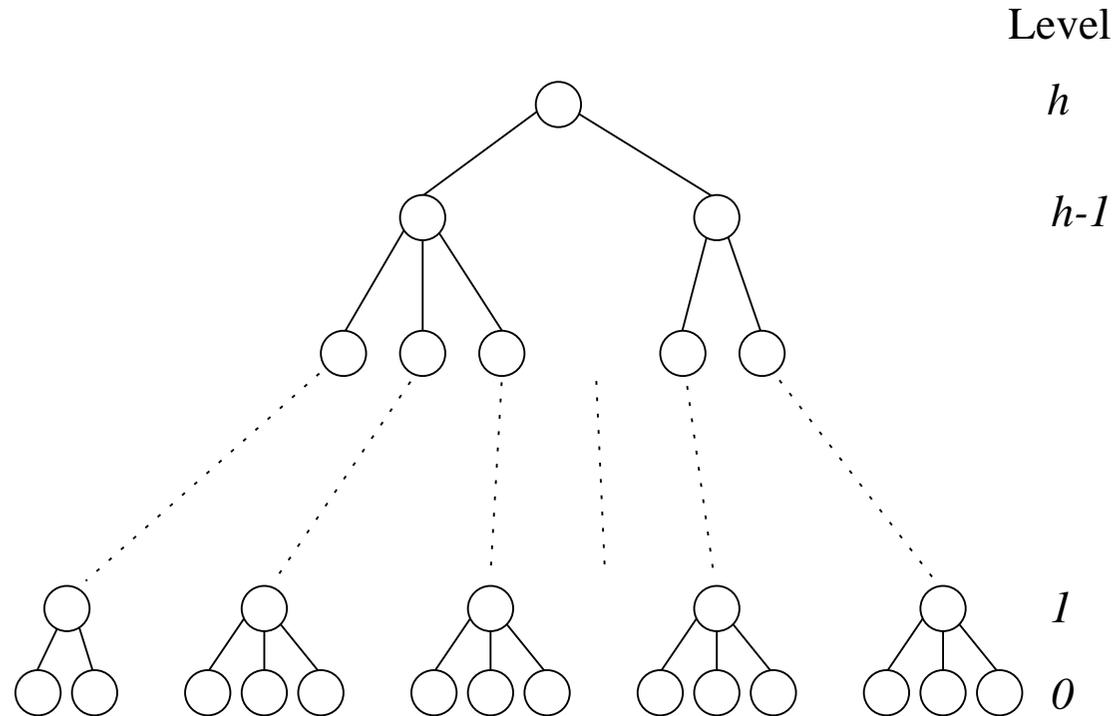


Kein 2-3-Baum (Leerer Unterbaum in Nicht-Blatt).



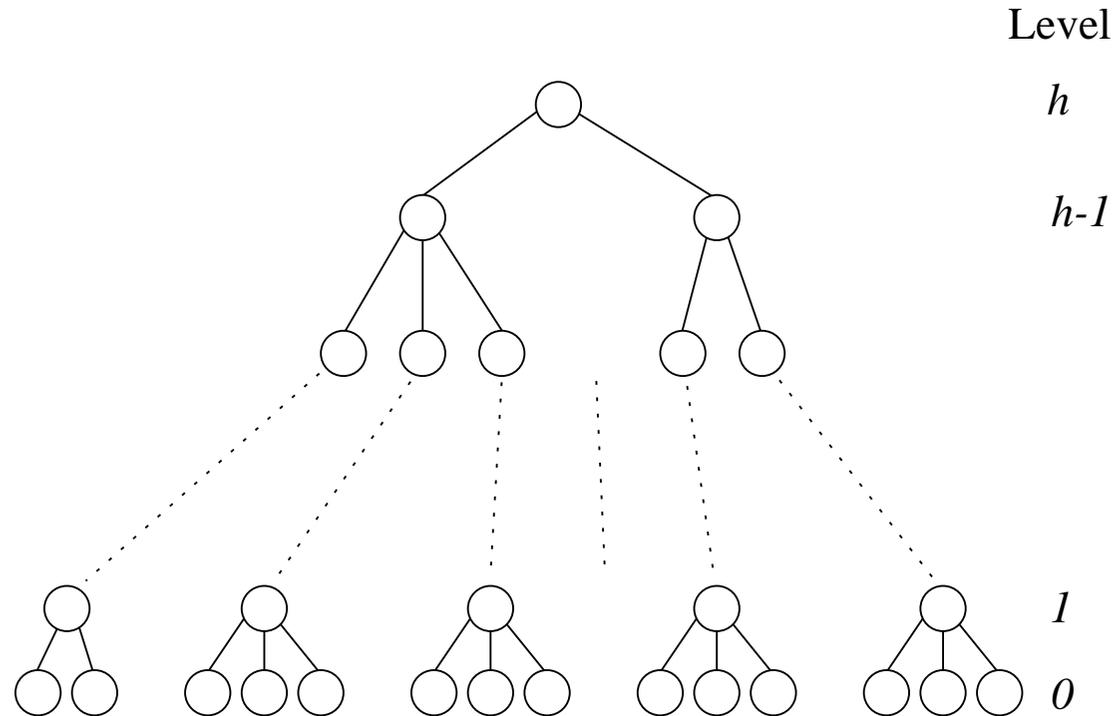


### Proposition 4.4.3



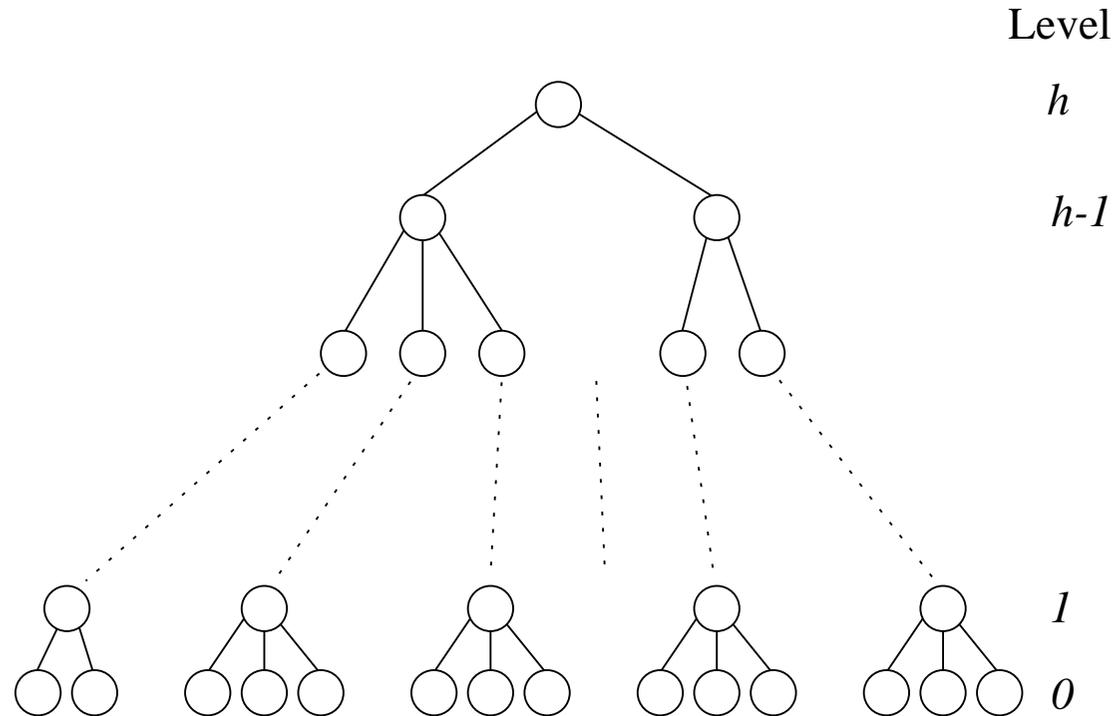
### Proposition 4.4.3

Die Tiefe eines 2-3-Baums mit  $n$  Einträgen ist mindestens  $\lceil \log_3(n + 1) \rceil - 1$  und höchstens  $\lfloor \log_2(n + 1) \rfloor - 1$ .



### Proposition 4.4.3

Die Tiefe eines 2-3-Baums mit  $n$  Einträgen ist mindestens  $\lceil \log_3(n + 1) \rceil - 1$  und höchstens  $\lfloor \log_2(n + 1) \rfloor - 1$ .  $\lceil \log_3 x \rceil \approx 0.63 \log_2 x$



### Proposition 4.4.3

Die Tiefe eines 2-3-Baums mit  $n$  Einträgen ist mindestens  $\lceil \log_3(n + 1) \rceil - 1$  und höchstens  $\lfloor \log_2(n + 1) \rfloor - 1$ .  $[\log_3 x \approx 0.63 \log_2 x]$  *Beweis: Übung.*

---

# Suche in 2-3-Bäumen

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T, x \in U$ .

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

**Zeitaufwand:**

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  sitzt:

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  sitzt: Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  entstehen Kosten  $O(1)$ . Erfolgreiche Suche endet auf Blattebene.

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  sitzt: Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  entstehen Kosten  $O(1)$ . Erfolgreiche Suche endet auf Blattebene.

$\Rightarrow$  Zeit für Suche in 2-3-Bäumen mit  $n$  Einträgen ist

---

## Suche in 2-3-Bäumen

**lookup**( $T, x$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

**1. Fall:**  $T = \square$ .      **return** „nicht vorhanden“.

**2. Fall:**  $T = (T_0, x_1, T_1)$  oder  $T = (T_0, x_1, T_1, x_2, T_2)$ , evtl. mit Datensatz  $r_1$  (und  $r_2$ ).

Sequentielle Abarbeitung der folgenden Tests:

**2a:**  $x < x_1$ : **return lookup**( $T_0, x$ ).

**2b:**  $x = x_1$ : **return** „gefunden“ bzw.  $r_1$ .

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **return lookup**( $T_1, x$ ).

**2d:**  $x = x_2$ : **return** „gefunden“ bzw.  $r_2$ .

**2e:**  $x_2 < x$ : **return lookup**( $T_2, x$ ).

**Zeitaufwand:** Wenn  $x$  im Knoten  $v_x$  sitzt: Für jeden Knoten  $v$  auf dem Weg von der Wurzel zu  $v_x$  entstehen Kosten  $O(1)$ . Erfolgreiche Suche endet auf Blattebene.

$\Rightarrow$  Zeit für Suche in 2-3-Bäumen mit  $n$  Einträgen ist  $O(\log n)$ .

---

# Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I):** (Diese werden wir benutzen und kontrollieren!)

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I):** (Diese werden wir benutzen und kontrollieren!)

$T'$  höher als  $T \Rightarrow T'$  hat in der Wurzel *nur einen* Schlüssel.

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I):** (Diese werden wir benutzen und kontrollieren!)

$T'$  höher als  $T \Rightarrow T'$  hat in der Wurzel *nur einen* Schlüssel.

**1. Fall:**  $T = \square$ .

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I):** (Diese werden wir benutzen und kontrollieren!)

$T'$  höher als  $T \Rightarrow T'$  hat in der Wurzel *nur einen* Schlüssel.

**1. Fall:**  $T = \square$ .

Erzeuge neuen 2-3-Baum-Knoten  $T$  mit Eintrag  $(x, r)$  und zwei leeren Unterbäumen.

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I):** (Diese werden wir benutzen und kontrollieren!)

$T'$  höher als  $T \Rightarrow T'$  hat in der Wurzel *nur einen* Schlüssel.

**1. Fall:**  $T = \square$ .

Erzeuge neuen 2-3-Baum-Knoten  $T$  mit Eintrag  $(x, r)$  und zwei leeren Unterbäumen.

**return** ( $T, \text{true}$ )

---

## Einfügen in 2-3-Bäumen

**insert**( $T, x, r$ ), für 2-3-Baum  $T$ ,  $x \in U$ .

Resultat: Neuer Baum  $T'$ , Boolescher Wert `higher` (gibt an, ob  $T'$  höher als  $T$  ist).

**Invariante (I)**: (Diese werden wir benutzen und kontrollieren!)

$T'$  höher als  $T \Rightarrow T'$  hat in der Wurzel *nur einen* Schlüssel.

**1. Fall:**  $T = \square$ .

Erzeuge neuen 2-3-Baum-Knoten  $T$  mit Eintrag  $(x, r)$  und zwei leeren Unterbäumen.

**return** ( $T, \text{true}$ )

**(I)** stimmt.

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert** $(T_0, x, r)$  liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ .

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ):

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **insert**( $T_1, x, r$ ) liefert  $T'_1$  (an Stelle von  $T_1$ ) und `sub_higher`.

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **insert**( $T_1, x, r$ ) liefert  $T'_1$  (an Stelle von  $T_1$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **insert**( $T_1, x, r$ ) liefert  $T'_1$  (an Stelle von  $T_1$ ) und **sub\_higher**.  
Dann: **Rebalancierung** (s. u.).

**2d:**  $x = x_2$ : Update: Ersetze  $r_2$  durch  $r$ .

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **insert**( $T_1, x, r$ ) liefert  $T'_1$  (an Stelle von  $T_1$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

**2d:**  $x = x_2$ : Update: Ersetze  $r_2$  durch  $r$ . **return** ( $T, false$ );

---

**2. Fall:**  $T = (T_0, (x_1, r_1), T_1)$  oder  $T = (T_0, (x_1, r_1), T_1, (x_2, r_2), T_2)$

**2a:**  $x < x_1$ : **insert**( $T_0, x, r$ ) liefert  $T'_0$  (an der Stelle von  $T_0$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

**2b:**  $x = x_1$ : Update: Ersetze  $r_1$  durch  $r$ . **return** ( $T, false$ );

**2c:** ( $x_2$  existiert nicht  $\vee x < x_2$ ): **insert**( $T_1, x, r$ ) liefert  $T'_1$  (an Stelle von  $T_1$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

**2d:**  $x = x_2$ : Update: Ersetze  $r_2$  durch  $r$ . **return** ( $T, false$ );

**2e:**  $x_2 < x$ : **insert**( $T_2, x, r$ ) liefert  $T'_2$  (an Stelle von  $T_2$ ) und `sub_higher`.  
Dann: **Rebalancierung** (s. u.).

---

# Rebalancierung

Fälle 2a/2c/2e:

---

# Rebalancierung

Fälle 2a/2c/2e:

Fall 2a(i)/2c(i)/2e(i): `sub_higher = false`.

// Der veränderte Teilbaum ist **nicht tiefer** geworden.

---

# Rebalancierung

Fälle 2a/2c/2e:

Fall 2a(i)/2c(i)/2e(i): `sub_higher = false`.

// Der veränderte Teilbaum ist **nicht tiefer** geworden.

**return**(*T*, *false*).

---

Fall 2a(ii)/2c(ii):  $x_2$  existiert nicht und `sub_higher = true`

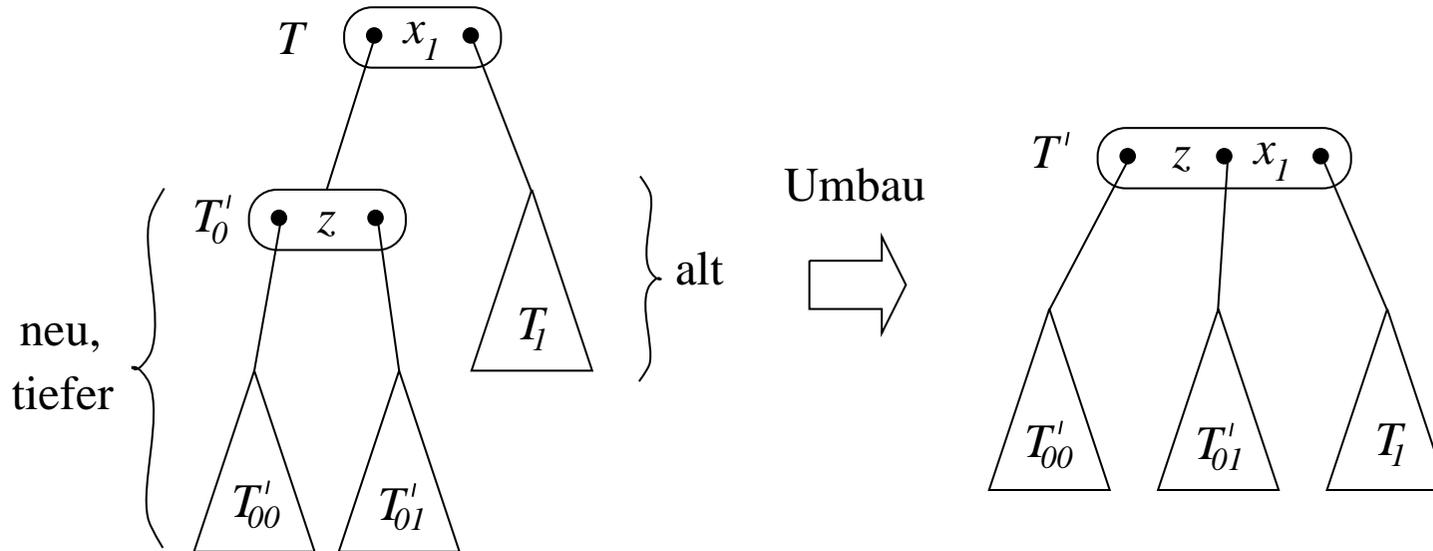
---

Fall 2a(ii)/2c(ii):  $x_2$  existiert nicht und `sub_higher = true`

2a(ii): Benutze **(I)**!

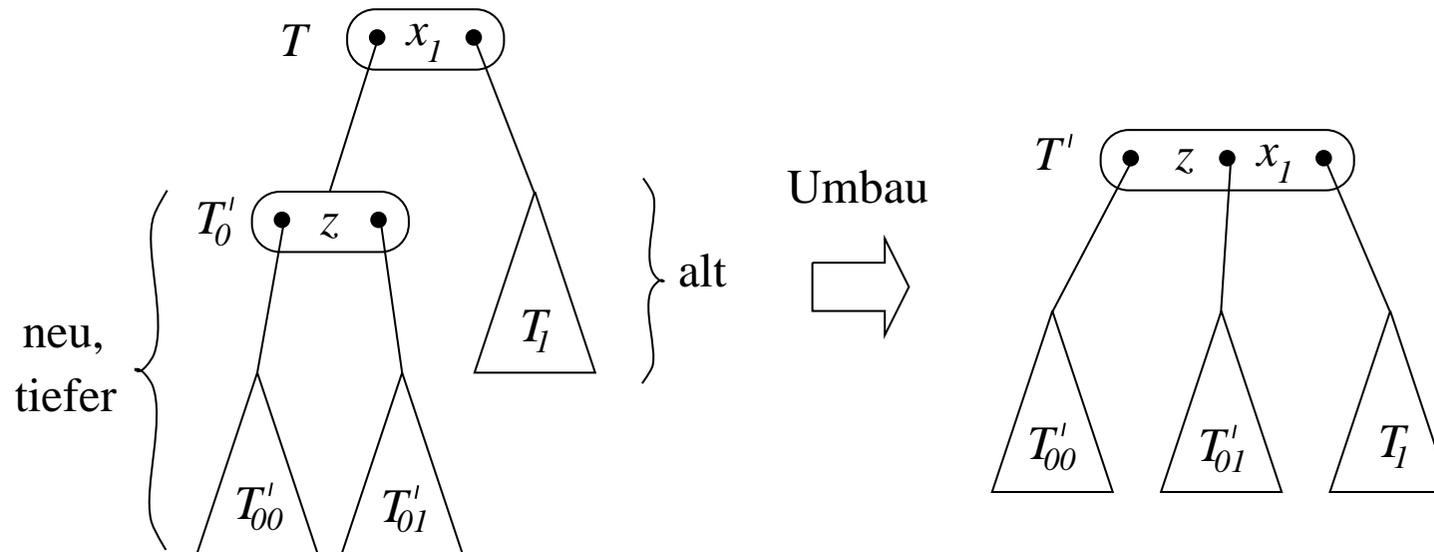
Fall 2a(ii)/2c(ii):  $x_2$  existiert nicht und `sub_higher = true`

2a(ii): Benutze **(I)**!



Fall 2a(ii)/2c(ii):  $x_2$  existiert nicht und `sub_higher = true`

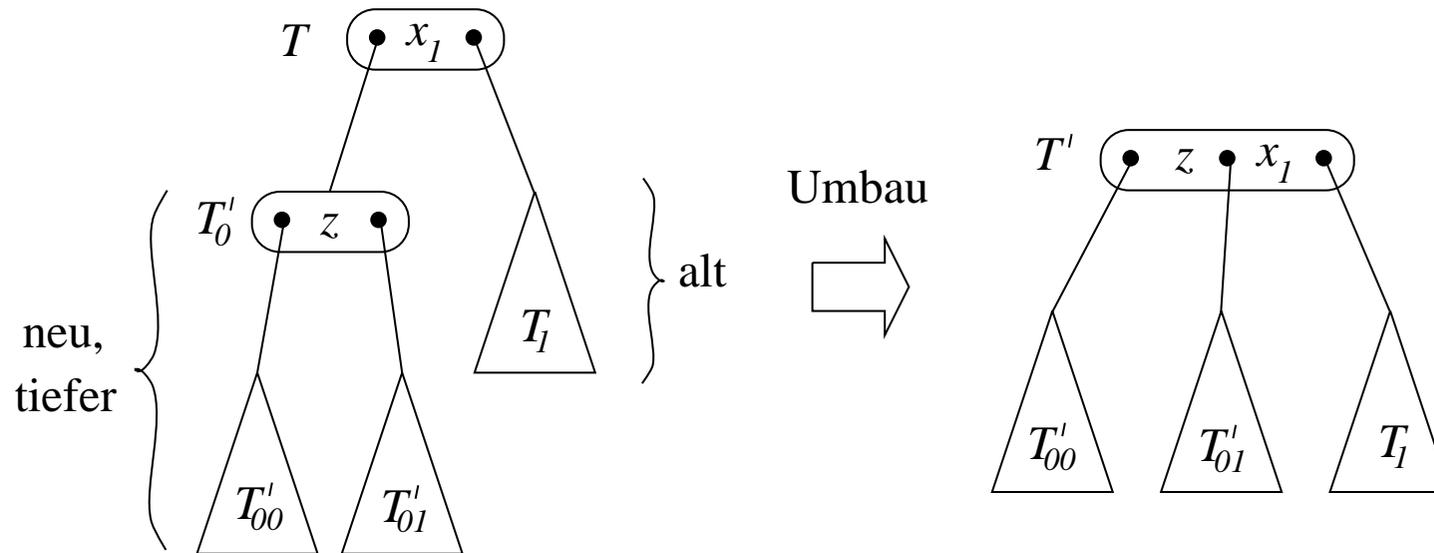
2a(ii): Benutze **(I)**!



**return**( $T'$ , *false*).

Fall 2a(ii)/2c(ii):  $x_2$  existiert nicht und `sub_higher = true`

2a(ii): Benutze **(I)**!



**return**( $T'$ , *false*).

2c(ii): analog.

---

Fall 2a(iii)/2c(iii)/2e(iii):  $x_2$  **existiert** und `sub_higher = true`

---

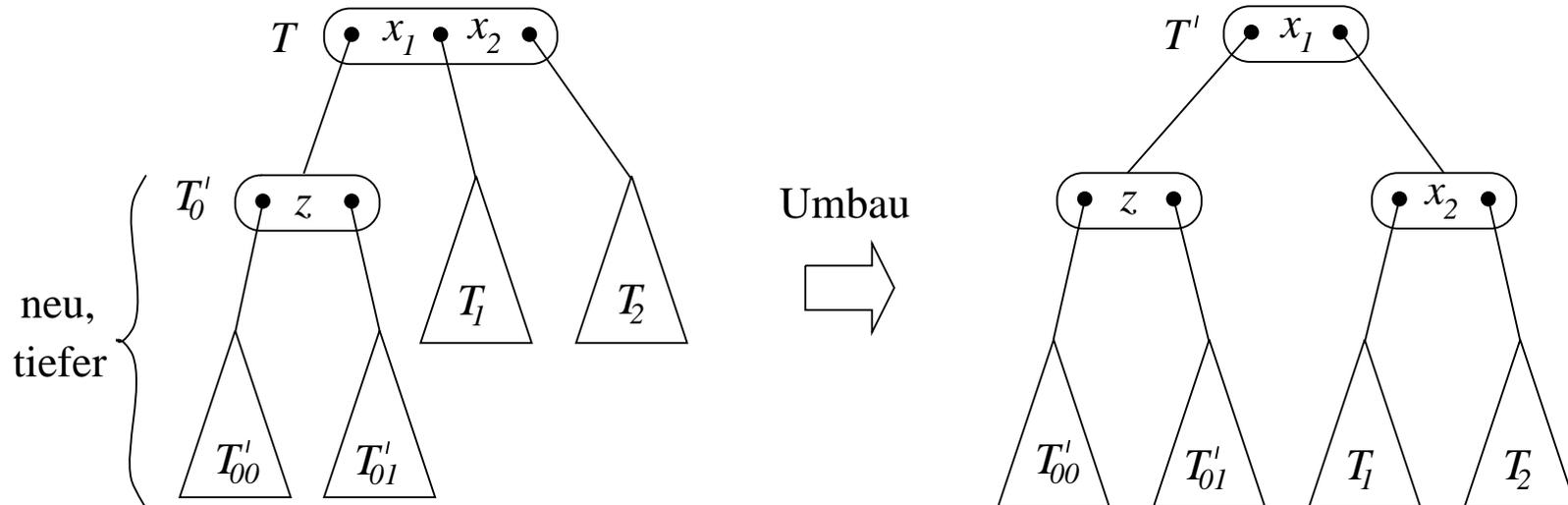
Fall 2a(iii)/2c(iii)/2e(iii):  $x_2$  **existiert** und `sub_higher = true`

Benutze jeweils **(I)**!

Fall 2a(iii)/2c(iii)/2e(iii):  $x_2$  **existiert** und `sub_higher = true`

Benutze jeweils **(I)**!

2a(iii):

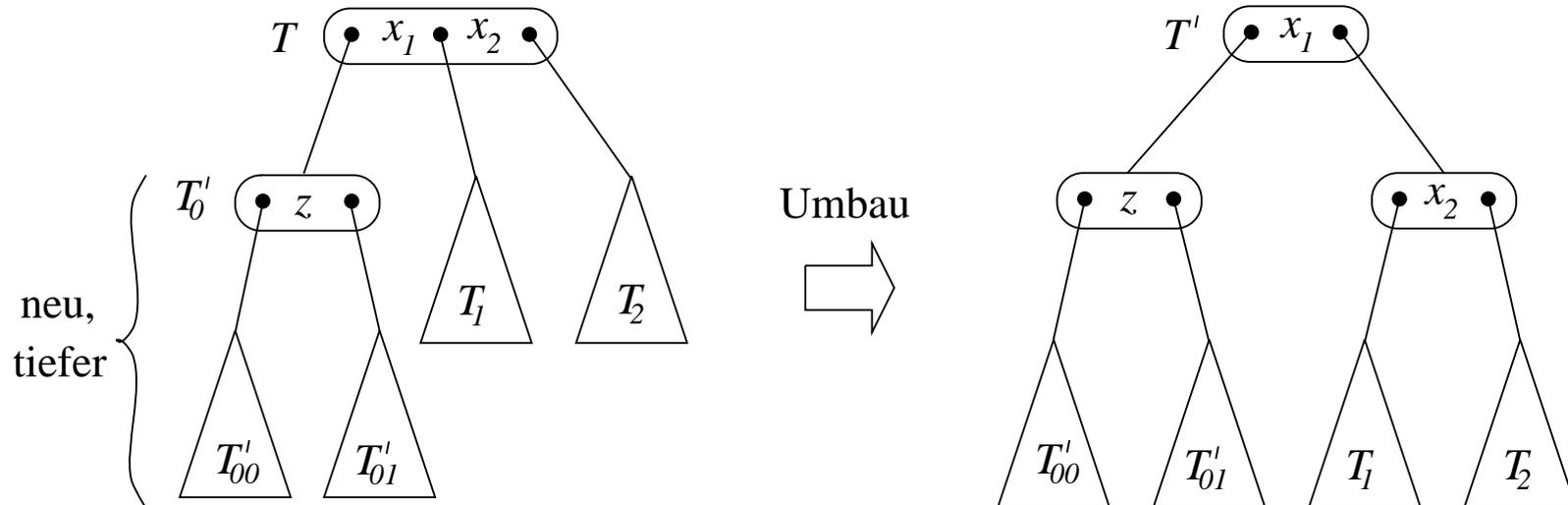


**return**( $T'$ , *true*).

Fall 2a(iii)/2c(iii)/2e(iii):  $x_2$  **existiert** und `sub_higher = true`

Benutze jeweils **(I)**!

2a(iii):



**return**( $T'$ , `true`).

Prüfe: **(I)** erfüllt.

2c(iii)/2e(iii): analog.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

*Beweis:*

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

*Beweis:*

- (a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

*Beweis:*

- (a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.
- (b) Für die Bearbeitung **eines Levels** in der *insert*-Prozedur wird Zeit  $O(1)$  benötigt.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

*Beweis:*

- (a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.
- (b) Für die Bearbeitung **eines Levels** in der *insert*-Prozedur wird Zeit  $O(1)$  benötigt. Auf jedem der  $\leq \log n$  Levels wird höchstens 1 Knoten neu gebildet.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

*Beweis:*

- (a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.
- (b) Für die Bearbeitung **eines Levels** in der *insert*-Prozedur wird Zeit  $O(1)$  benötigt. Auf jedem der  $\leq \log n$  Levels wird höchstens 1 Knoten neu gebildet.

**Folgerung:** Für jedes  $n \geq 0$  existiert ein 2-3-Baum mit  $n$  Schlüsseln.

---

**Beobachtung:** Die Tiefe von 2-3-Bäumen wächst durch „Spalten der Wurzel“.

### Proposition 4.4.4

- (a) Die Prozedur **insert** ist korrekt, d. h. der Aufruf **insert**( $T, x, r$ ) liefert einen 2-3-Baum für  $insert(f_T, x, r)$ .
- (b) Das Einfügen eines Schlüssels in einen 2-3-Baum mit  $n$  Schlüsseln benötigt Zeit  $O(\log n)$  und bewirkt die Erzeugung von höchstens  $\log_2 n$  neuen Knoten.

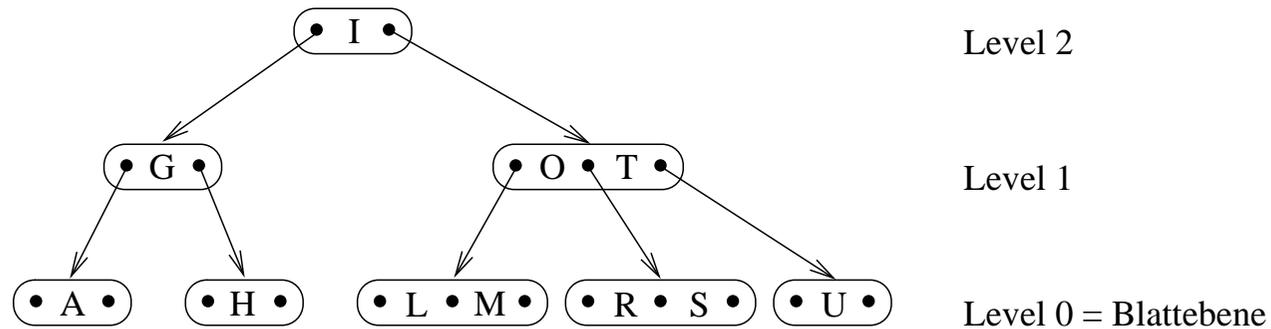
*Beweis:*

- (a) Man kontrolliert in jedem Fall des Algorithmus nach, dass die 2-3-Baum-Struktur wiederhergestellt wird.
- (b) Für die Bearbeitung **eines Levels** in der *insert*-Prozedur wird Zeit  $O(1)$  benötigt.  
Auf jedem der  $\leq \log n$  Levels wird höchstens 1 Knoten neu gebildet.

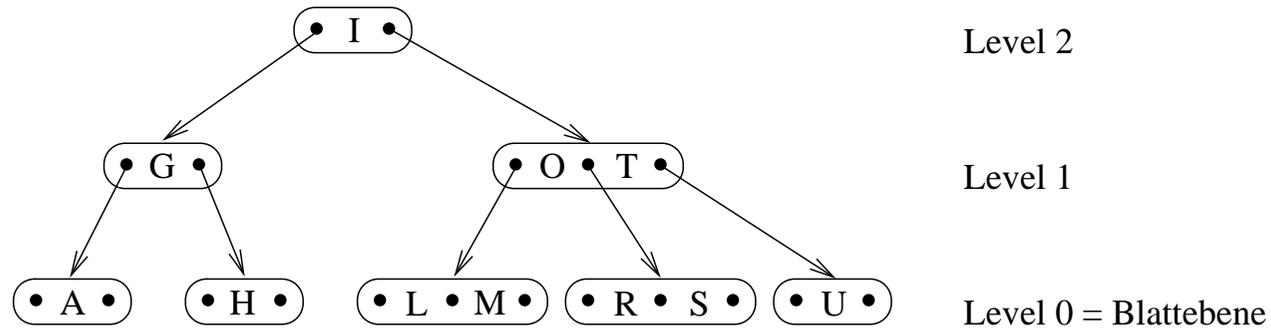
**Folgerung:** Für jedes  $n \geq 0$  existiert ein 2-3-Baum mit  $n$  Schlüsseln. (Man füge in einen anfangs leeren Baum die Schlüssel  $1, \dots, n$  ein.)

---

# Löschen in 2-3-Bäumen

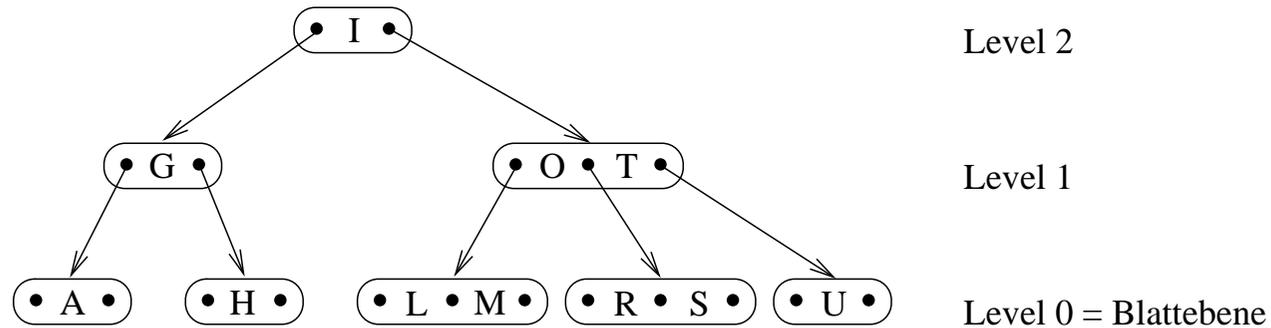


# Löschen in 2-3-Bäumen



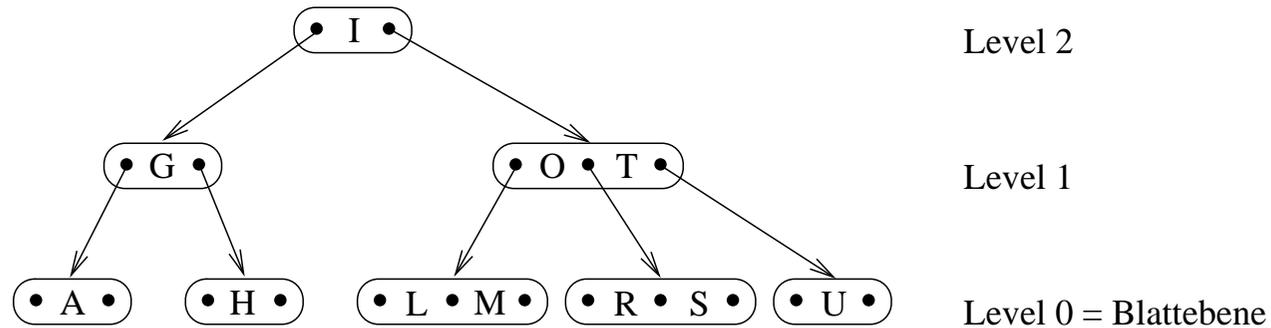
**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist.

# Löschen in 2-3-Bäumen



**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist.  
( $x$  oder Inorder-Nachfolger von  $x$ .)

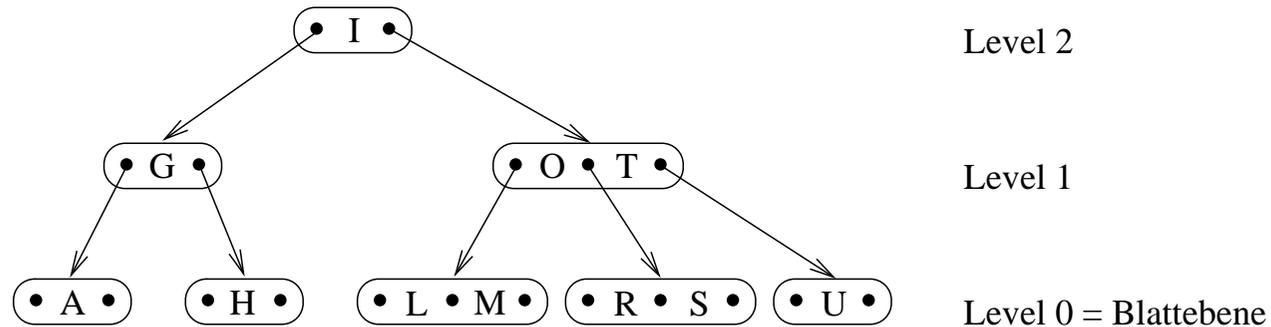
# Löschen in 2-3-Bäumen



**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist.  
( $x$  oder Inorder-Nachfolger von  $x$ .)

Setze  $y$  an die Stelle von  $x$  und entferne  $y$ .

## Löschen in 2-3-Bäumen

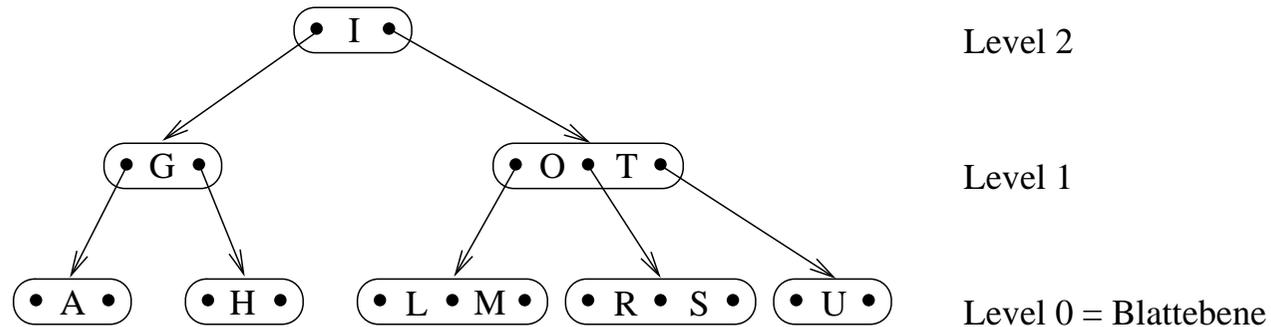


**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist.  
( $x$  oder Inorder-Nachfolger von  $x$ .)

Setze  $y$  an die Stelle von  $x$  und entferne  $y$ .

*Beispiele:* U wird einfach gelöscht.

# Löschen in 2-3-Bäumen

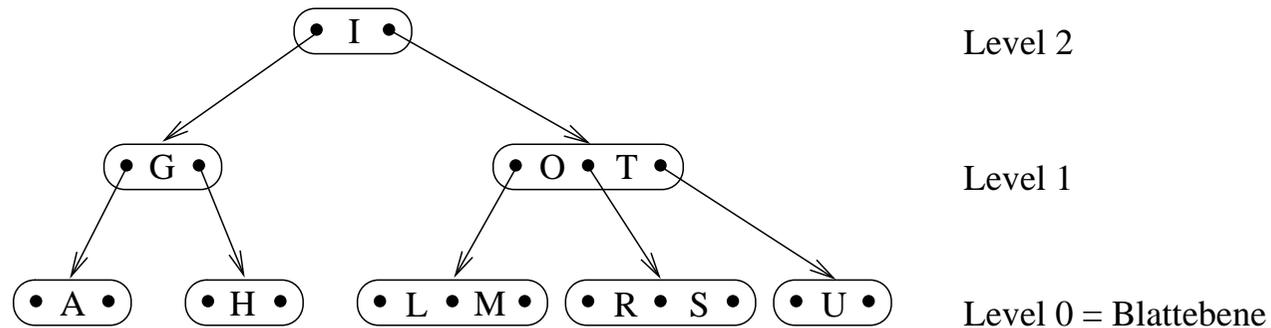


**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist. ( $x$  oder Inorder-Nachfolger von  $x$ .)

Setze  $y$  an die Stelle von  $x$  und entferne  $y$ .

*Beispiele:*  $U$  wird einfach gelöscht. Um  $I$  zu löschen:  $L$  in die Wurzel ziehen.

## Löschen in 2-3-Bäumen

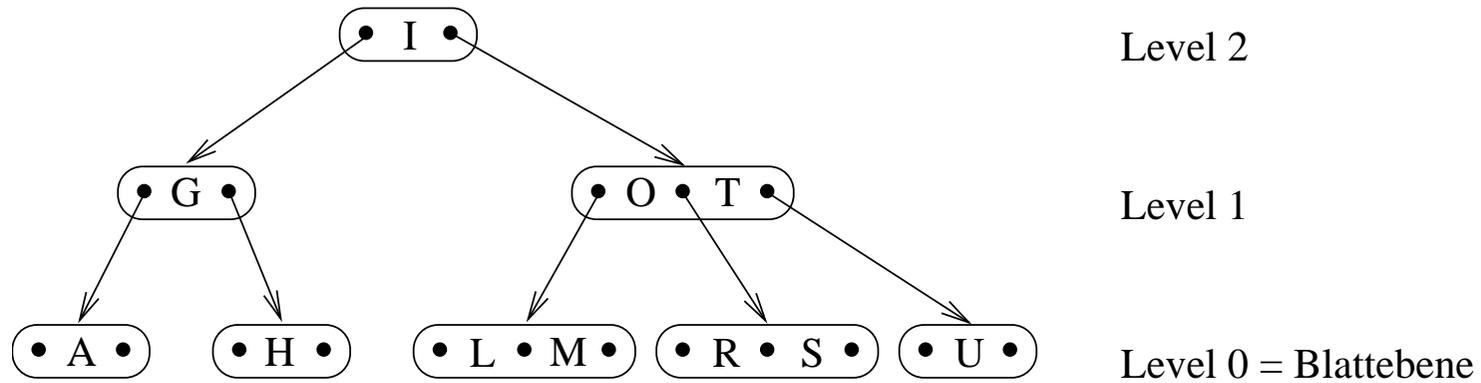


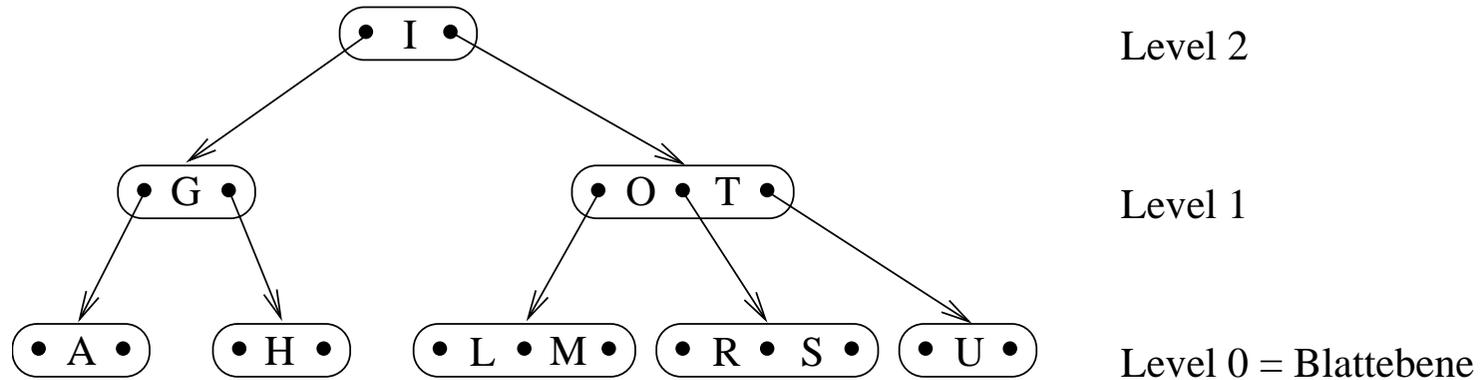
**Prinzip:** Suche kleinsten Schlüssel  $y \geq x$ , der in einem Blatt gespeichert ist. ( $x$  oder Inorder-Nachfolger von  $x$ .)

Setze  $y$  an die Stelle von  $x$  und entferne  $y$ .

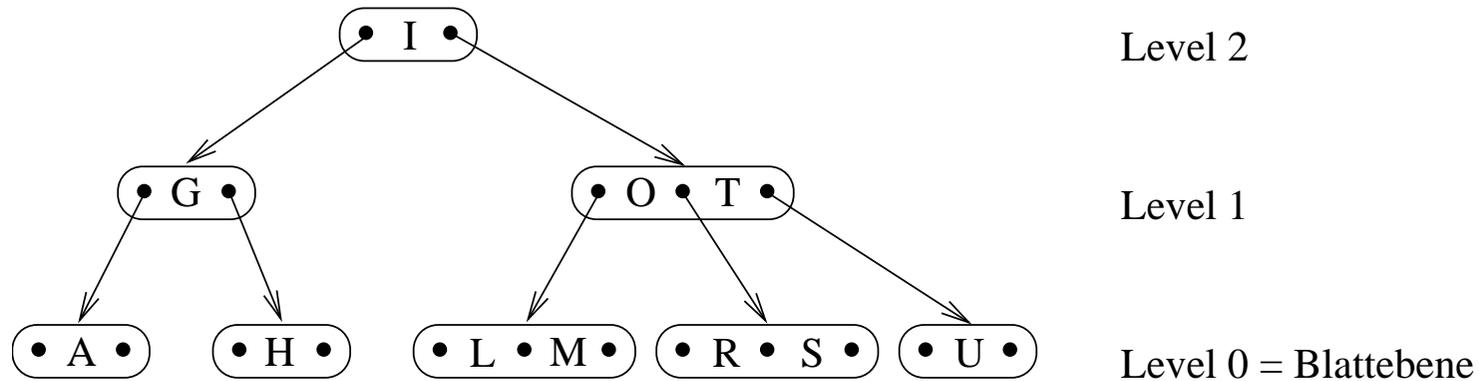
*Beispiele:*  $U$  wird einfach gelöscht. Um  $I$  zu löschen:  $L$  in die Wurzel ziehen.

**Daher:** Nur Löschen eines Eintrags  $y$  in einem **Blatt** ist relevant.

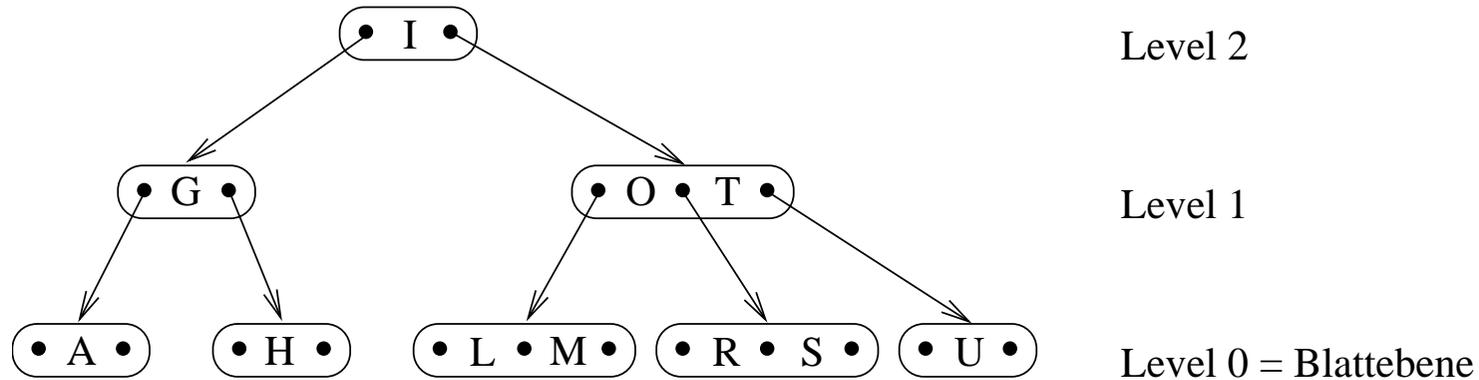




- Einfacher Fall: Neben  $y$  hat das Blatt noch andere Einträge (Beispiel: M).



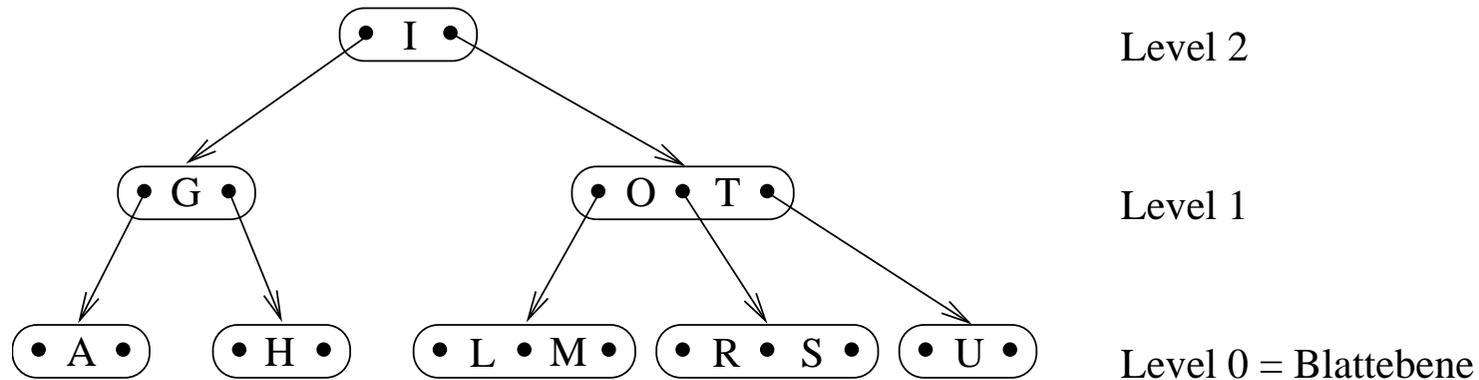
- Einfacher Fall: Neben  $y$  hat das Blatt noch andere Einträge (Beispiel: M).  
Keine Umstrukturierung nötig.



- Einfacher Fall: Neben  $y$  hat das Blatt noch andere Einträge (Beispiel: M).

Keine Umstrukturierung nötig.

- Rebalancierungsfall: Durch das Streichen von  $y$  wird das Blatt leer, verliert also an Höhe (Beispiel: A).

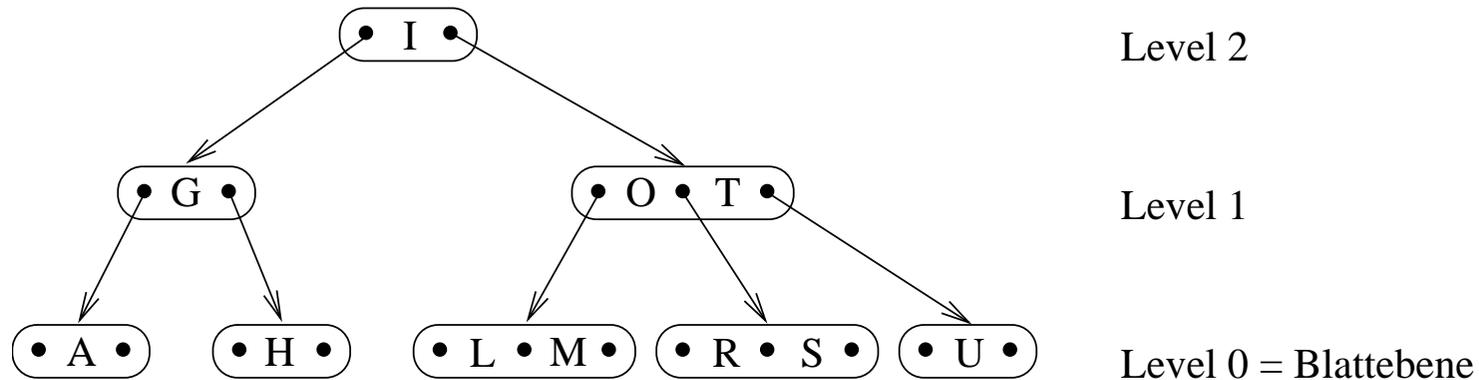


- Einfacher Fall: Neben  $y$  hat das Blatt noch andere Einträge (Beispiel: M).

Keine Umstrukturierung nötig.

- Rebalancierungsfall: Durch das Streichen von  $y$  wird das Blatt leer, verliert also an Höhe (Beispiel: A).

Ähnlich wie bei den Einfügungen werden **flacher gewordene Teilbäume** durch lokale Umstellungen, von unten nach oben, wieder **auf die Höhe der anderen Teilbäume gebracht**.



- Einfacher Fall: Neben  $y$  hat das Blatt noch andere Einträge (Beispiel: M).

Keine Umstrukturierung nötig.

- Rebalancierungsfall: Durch das Streichen von  $y$  wird das Blatt leer, verliert also an Höhe (Beispiel: A).

Ähnlich wie bei den Einfügungen werden **flacher gewordene Teilbäume** durch lokale Umstellungen, von unten nach oben, wieder **auf die Höhe der anderen Teilbäume gebracht**. (Details weggelassen.)

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)
- (b) Für jeden Knoten  $v$  in  $T$  gilt:  $v$  hat **einen** leeren Unterbaum  $\Rightarrow$  **alle** Unterbäume unter  $v$  sind leer, d. h.  $v$  ist **Blatt**;

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)
- (b) Für jeden Knoten  $v$  in  $T$  gilt:  $v$  hat **einen** leeren Unterbaum  $\Rightarrow$  **alle** Unterbäume unter  $v$  sind leer, d. h.  $v$  ist **Blatt**;
- (c) Alle Blätter von  $T$  haben dieselbe Tiefe.

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)
- (b) Für jeden Knoten  $v$  in  $T$  gilt:  $v$  hat **einen** leeren Unterbaum  $\Rightarrow$  **alle** Unterbäume unter  $v$  sind leer, d. h.  $v$  ist **Blatt**;
- (c) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Wörterbuchoperationen können auch mit 2-3-4-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Die Rebalancierung lässt sich alternativ „top-down“ implementieren, auf dem Weg zur Einfügestelle bzw. Löschstelle, ohne Zurücklaufen.

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)
- (b) Für jeden Knoten  $v$  in  $T$  gilt:  $v$  hat **einen** leeren Unterbaum  $\Rightarrow$  **alle** Unterbäume unter  $v$  sind leer, d. h.  $v$  ist **Blatt**;
- (c) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Wörterbuchoperationen können auch mit 2-3-4-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Die Rebalancierung lässt sich alternativ „top-down“ implementieren, auf dem Weg zur Einfügestelle bzw. Löschstelle, ohne Zurücklaufen.

Lit.: [\[D./Mehlhorn/Sanders\]](#). (Dort: Einträge nur in Blättern.)

---

## 2-3-4-Bäume

**Definition** Ein **2-3-4-Baum** ist ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)
- (b) Für jeden Knoten  $v$  in  $T$  gilt:  $v$  hat **einen** leeren Unterbaum  $\Rightarrow$  **alle** Unterbäume unter  $v$  sind leer, d. h.  $v$  ist **Blatt**;
- (c) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Wörterbuchoperationen können auch mit 2-3-4-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Die Rebalancierung lässt sich alternativ „top-down“ implementieren, auf dem Weg zur Einfügestelle bzw. Löschstelle, ohne Zurücklaufen.

Lit.: [D./Mehlhorn/Sanders]. (Dort: Einträge nur in Blättern.)

Man erhält den allgemeineren Begriff des  **$(a, b)$ -Baums**, wenn man in der obigen Definition „2, 3 oder 4 Kinder“ durch „zwischen  $a$  und  $b$  Kindern, für  $2 \leq a \leq (b + 1)/2$  (Ausnahme: die Wurzel hat zwischen 2 und  $b$  Kinder)“ ersetzt.

**Vorlesungsvideo:**

**Rot-Schwarz-Bäume und B-Bäume**

---

# Rot-Schwarz-Bäume

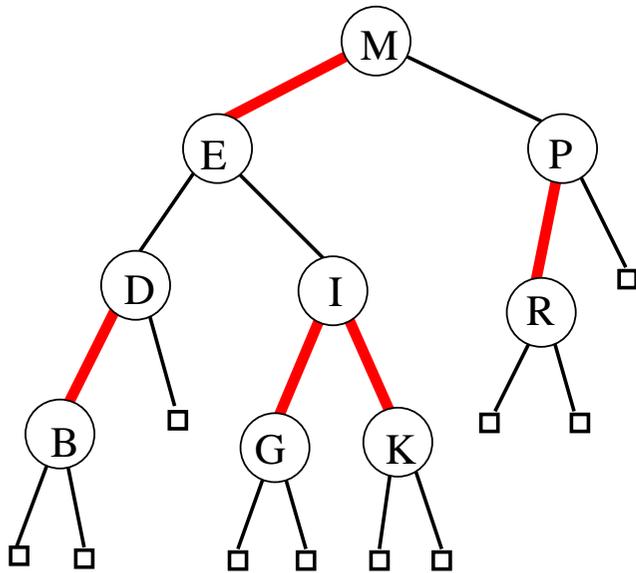
## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

# Rot-Schwarz-Bäume

## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

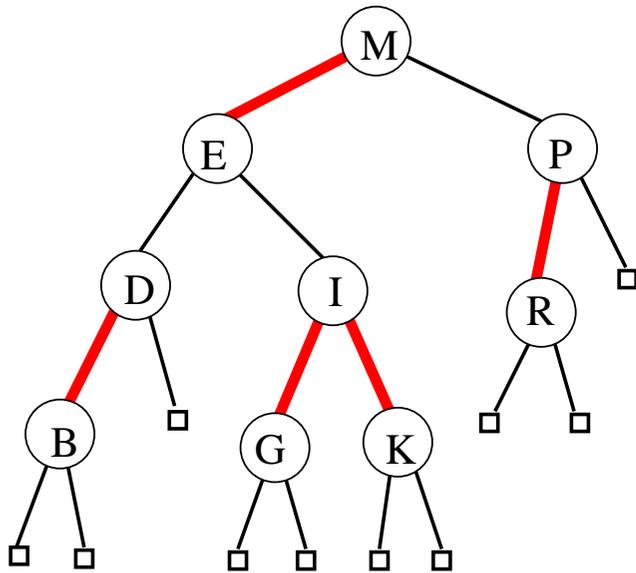


# Rot-Schwarz-Bäume

## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

- (a) Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).

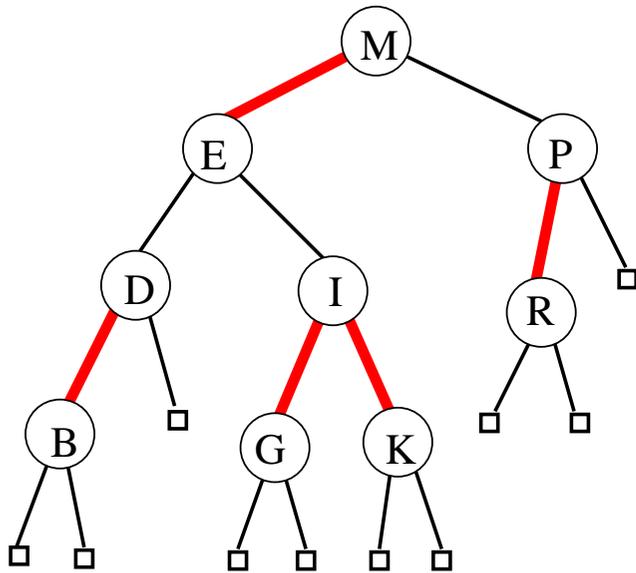


# Rot-Schwarz-Bäume

## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

- (a) Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).
- (b) Kanten zu (leeren, externen!) Blättern sind **schwarz**.

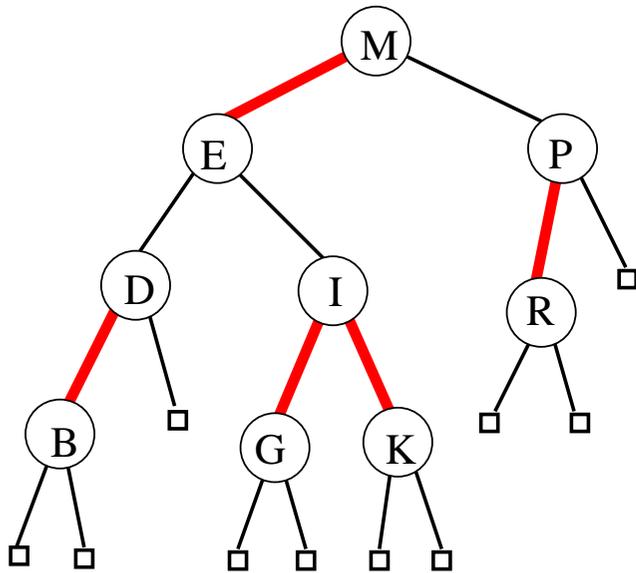


# Rot-Schwarz-Bäume

## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

- (a) Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).
- (b) Kanten zu (leeren, externen!) Blättern sind **schwarz**.
- (c) Auf keinem Weg folgen zwei rote Kanten aufeinander.

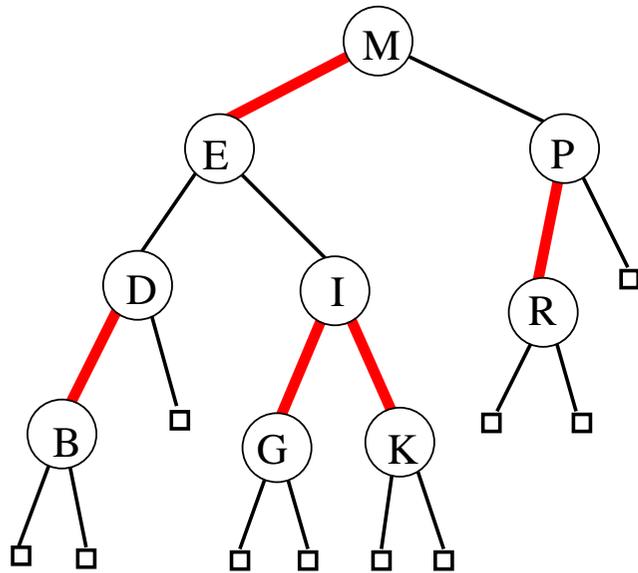


# Rot-Schwarz-Bäume

## Definition

Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:

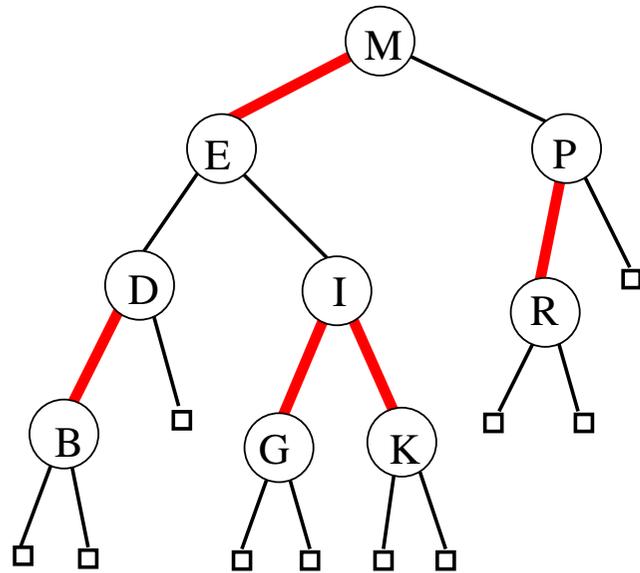
- Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).
- Kanten zu (leeren, externen!) Blättern sind **schwarz**.
- Auf keinem Weg folgen zwei rote Kanten aufeinander.
- Wenn ein Knoten nur eine rote Ausgangskante hat, so führt diese zum linken Kind.



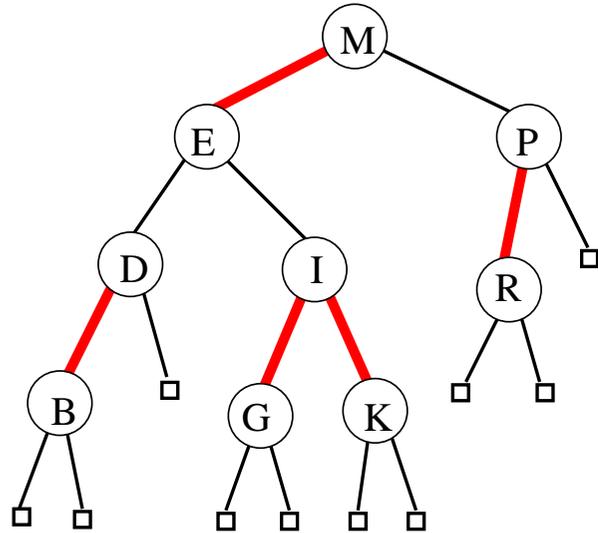
# Rot-Schwarz-Bäume

## Definition

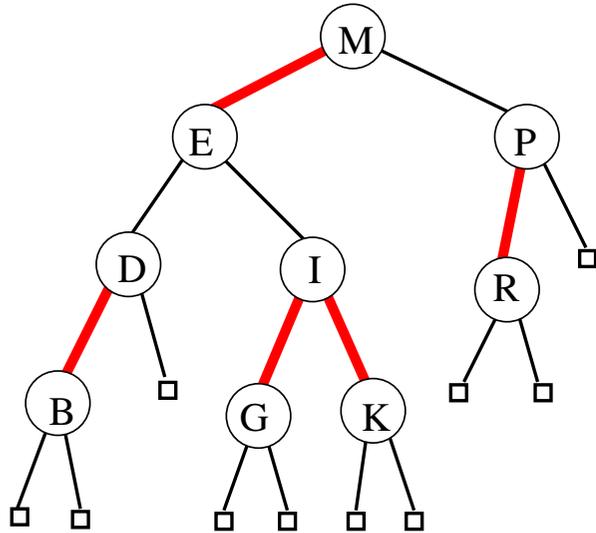
Ein (linksgeneigter) **Rot-Schwarz-Baum** ist ein **binärer Suchbaum** mit:



- Jede Kante hat eine Farbe **rot** oder **schwarz** (1 Flagbit!).
- Kanten zu (leeren, externen!) Blättern sind **schwarz**.
- Auf keinem Weg folgen zwei rote Kanten aufeinander.
- Wenn ein Knoten nur eine rote Ausgangskante hat, so führt diese zum linken Kind.
- Auf jedem Weg von der Wurzel zu einem Blatt liegen gleich viele schwarze Kanten.

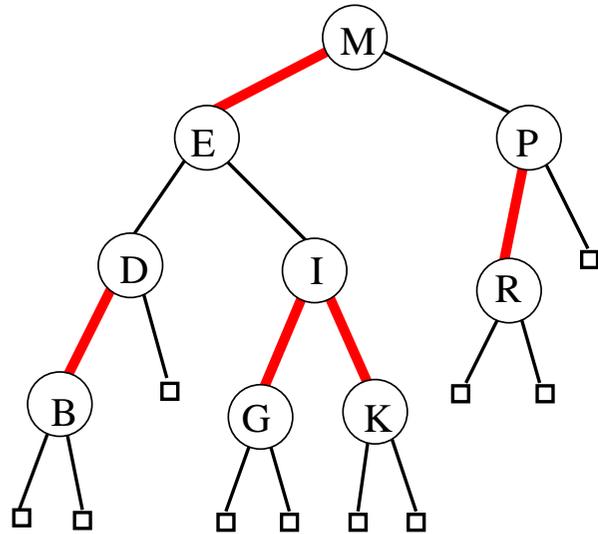


Die **Schwarz-Tiefe**  $b(v)$  eines Knotens  $v$  ist die Anzahl der schwarzen Kanten auf einem Weg von  $v$  zu einem Blatt.



Die **Schwarz-Tiefe**  $b(v)$  eines Knotens  $v$  ist die Anzahl der schwarzen Kanten auf einem Weg von  $v$  zu einem Blatt.

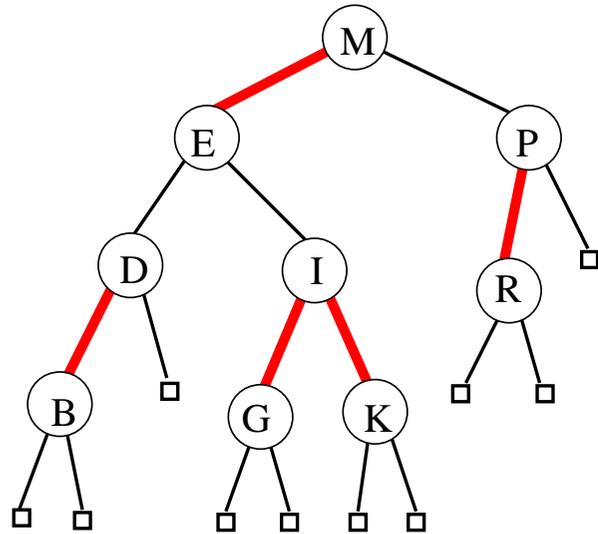
Die **Schwarz-Tiefe**  $b(T)$  des Rot-Schwarz-Baums  $T$  ist die Schwarz-Tiefe des Wurzelknotens von  $T$ .



Schwarz-Tiefe: 2.

Die **Schwarz-Tiefe**  $b(v)$  eines Knotens  $v$  ist die Anzahl der schwarzen Kanten auf einem Weg von  $v$  zu einem Blatt.

Die **Schwarz-Tiefe**  $b(T)$  des Rot-Schwarz-Baums  $T$  ist die Schwarz-Tiefe des Wurzelknotens von  $T$ .



Schwarz-Tiefe: 2.

## Mitteilung

Die Wörterbuchoperationen können auch mit Rot-Schwarz-Bäumen so implementiert werden, dass sie **logarithmische Zeit** benötigen. Weiterführende Operationen (wie das Spalten von Bäumen oder die Vereinigung von Bäumen) lassen sich mit Rot-Schwarz-Bäumen gut implementieren.

Lit.: [\[Cormen, Leiserson, Rivest, Stein\]](#) und [\[Sedgewick\]](#).

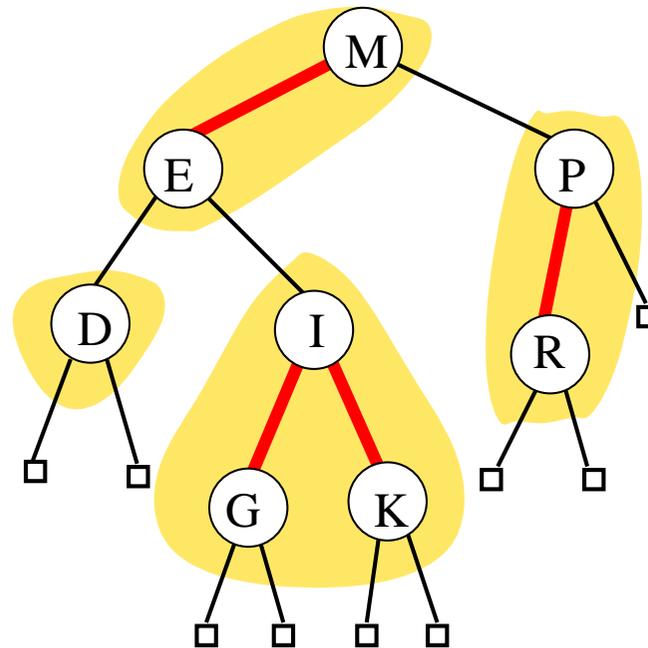
Die **Schwarz-Tiefe**  $b(v)$  eines Knotens  $v$  ist die Anzahl der schwarzen Kanten auf einem Weg von  $v$  zu einem Blatt.

Die **Schwarz-Tiefe**  $b(T)$  des Rot-Schwarz-Baums  $T$  ist die Schwarz-Tiefe des Wurzelknotens von  $T$ .

---

Wenn man einen schwarzen Knoten mit seinen roten Kindern (0, 1 oder 2) zu einem „Super-Knoten“ zusammenfasst, erhält man einen 2-3-4-Baum.

D. h.: Rot-Schwarz-Bäume sind spezielle Darstellungen von 2-3-4-Bäumen, die nur Binärbaumknoten benutzen.



---

# B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);

---

## B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);

---

## B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;

---

## B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

---

# B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$

---

# B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$ , genauer mindestens  $(\log n)/\log(2t) - 1$ , höchstens  $(\log n)/\log t$ .

---

# B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$ , genauer mindestens  $(\log n)/\log(2t) - 1$ , höchstens  $(\log n)/\log t$ .

*Beispiel:* Die Tiefe eines B-Baums zu  $t = 32$  ist höchstens  $(\log n)/5$ .

---

## B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$ , genauer mindestens  $(\log n)/\log(2t) - 1$ , höchstens  $(\log n)/\log t$ .

*Beispiel:* Die Tiefe eines B-Baums zu  $t = 32$  ist höchstens  $(\log n)/5$ .

Jede Wörterbuchoperation betrifft höchstens zwei Knoten auf jedem Level, also höchstens  $2(\log n)/\log t$  viele.

---

# B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$ , genauer mindestens  $(\log n)/\log(2t) - 1$ , höchstens  $(\log n)/\log t$ .

*Beispiel:* Die Tiefe eines B-Baums zu  $t = 32$  ist höchstens  $(\log n)/5$ .

Jede Wörterbuchoperation betrifft höchstens zwei Knoten auf jedem Level, also höchstens  $2(\log n)/\log t$  viele.

B-Bäume werden für die Implementierung von Wörterbüchern auf externen Speichermedien (Festplatte, SSD) benutzt, die eine höhere Latenzzeit haben und Lesen in größeren Blöcken erlauben.

---

## B-Bäume

Ein **B-Baum** zum **Parameter**  $t \geq 2$  ist ein  $(t, 2t)$ -Baum, also ein **Mehrweg-Suchbaum** mit:

- (a) Jeder Knoten hat maximal  $2t$  Kinder ( $\leq 2t - 1$  Schlüssel);
- (b) Jeder Knoten außer der Wurzel hat mindestens  $t$  Kinder ( $\geq t - 1$  Schlüssel);  
die Wurzel hat mindestens 2 Kinder ( $\geq 1$  Schlüssel);
- (c)  $v$  hat **einen** leeren Unterbaum  $\Rightarrow v$  ist **Blatt**;
- (d) Alle Blätter von  $T$  haben dieselbe Tiefe.

**Mitteilung** Die Tiefe eines B-Baums mit Parameter  $t$  für  $n$  Einträge ist  $\Theta(\log_t n) = \Theta((\log n)/(\log t))$ , genauer mindestens  $(\log n)/\log(2t) - 1$ , höchstens  $(\log n)/\log t$ .

*Beispiel:* Die Tiefe eines B-Baums zu  $t = 32$  ist höchstens  $(\log n)/5$ .

Jede Wörterbuchoperation betrifft höchstens zwei Knoten auf jedem Level, also höchstens  $2(\log n)/\log t$  viele.

B-Bäume werden für die Implementierung von Wörterbüchern auf externen Speichermedien (Festplatte, SSD) benutzt, die eine höhere Latenzzeit haben und Lesen in größeren Blöcken erlauben. Details: Vorlesungen zu Datenbanktechniken.