SS17

Algorithmen und Datenstrukturen

4. Kapitel

Suchbäume

Martin Dietzfelbinger

April/Mai 2017

4.1 Binäre Suchbäume

Binäre Suchbäume

implementieren den Datentyp Wörterbuch

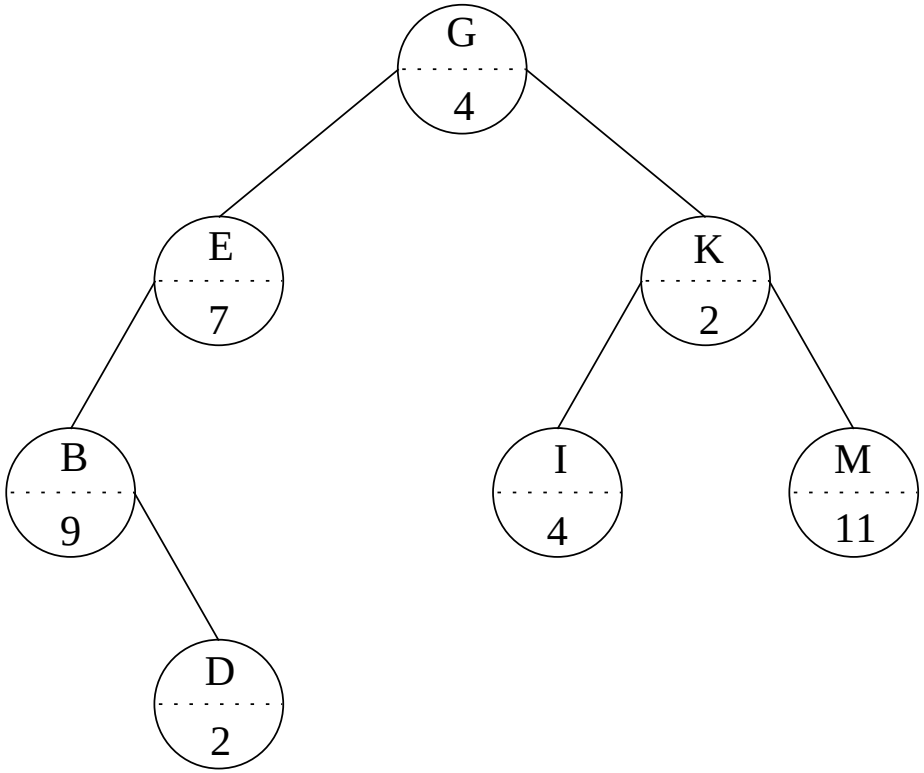
Schlüsselmenge $(U, <)$ (total geordnet) und Wertemenge R

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

- *empty*: leeres Wörterbuch erzeugen
- *lookup*: falls $x \in S$: $f(x)$ ausgeben (sonst: „undefined“)
- *insert*: neue Paare $(x, f(x))$ einfügen (bzw. aktualisieren)
- *delete*: anhand von x Paar $(x, f(x))$ löschen (falls $x \in S$)

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

Ein binärer Suchbaum T:



Dargestellte Funktion $f = f_T$:

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

Binärer Suchbaum

Knoteneinträge (hier in inneren Knoten):

$key(v)$: Schlüssel (aus U)

$data(v)$: Daten, Wert (aus R) (oft: Zeiger, Referenz)

Definition 4.1.1

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

Knoten w im **linken** Unterbaum von $T_v \Rightarrow key(w) < key(v)$

Knoten w im **rechten** Unterbaum von $T_v \Rightarrow key(v) < key(w)$

---

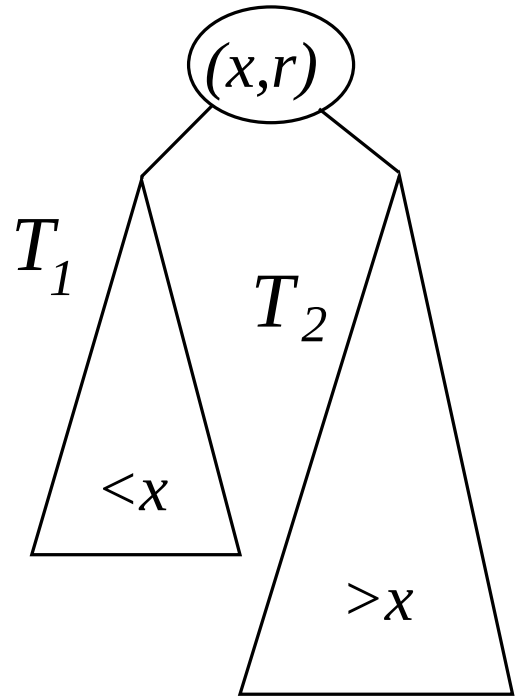
engl.: *binary search tree*. Abk.: **(U, R)-BSB** oder **(U, R)-BST**

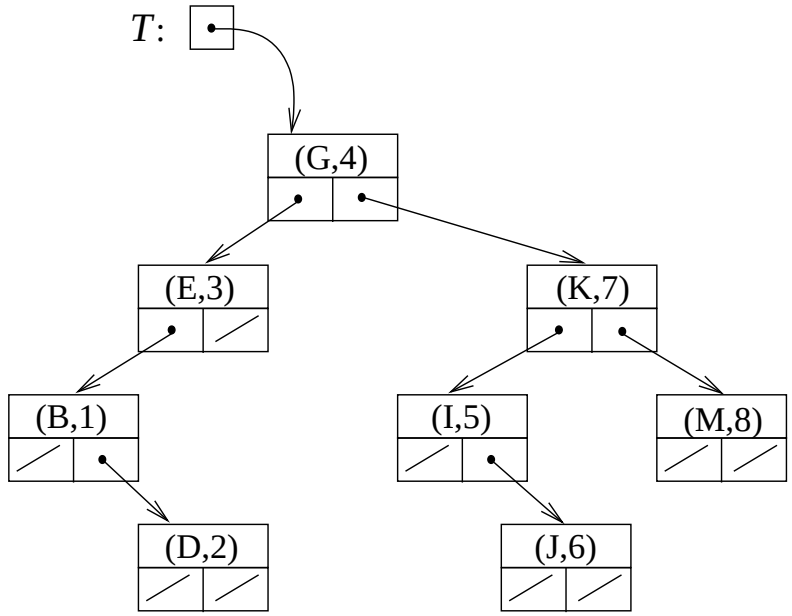
Alternative Definition: Rekursive Auffassung.

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

(ii) **Wenn** $x \in U$ und $r \in R$ **und**
T_1, T_2 (U, R)-BSB
mit Schlüsselmengen S_1, $S_2 \subseteq U$,
und wenn $\forall y \in S_1$: $y < x$
und $\forall z \in S_2$: $x < z$,
dann ist $(T_1, (x, r), T_2)$
ein (U, R)-BSB
mit Schlüsselmenge $S_1 \cup \{x\} \cup S_2$.

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





Implementierung von binären Suchbäumen:
Verzeigerte Struktur.

Knoten hat Komponenten für Schlüssel, Daten und Zeiger/Referenzen auf linkes und rechtes Kind:
$T.key$, $T.data$, $T.left$, $T.right$

Baum ist gegeben durch Zeiger/Referenz auf Wurzel.

Suchen, rekursiv

lookup(T, x), für BSB T, $x \in U$.

// Ergebnis: Zeiger auf Knoten mit x, falls vorhanden

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

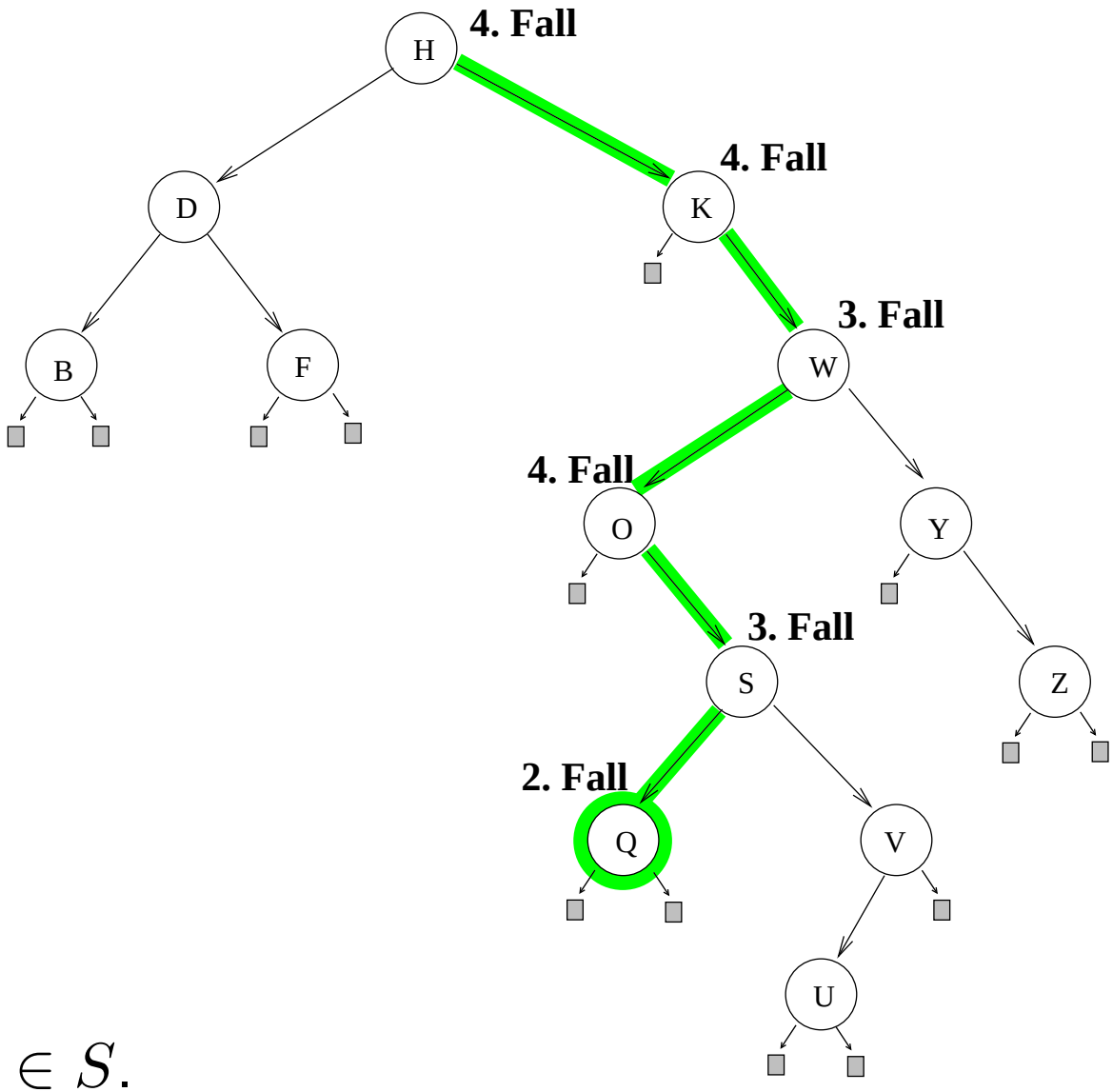
Ab hier: $T \neq \square$, also $T = (T_1, (y, r), T_2)$.

2. Fall: $x = y$.
return $(r,$ [Zeiger auf den Wurzelknoten von] $T)$.

3. Fall: $x < y$.
return lookup$(T.\mathit{left}, x)$ // rekursiver Aufruf .

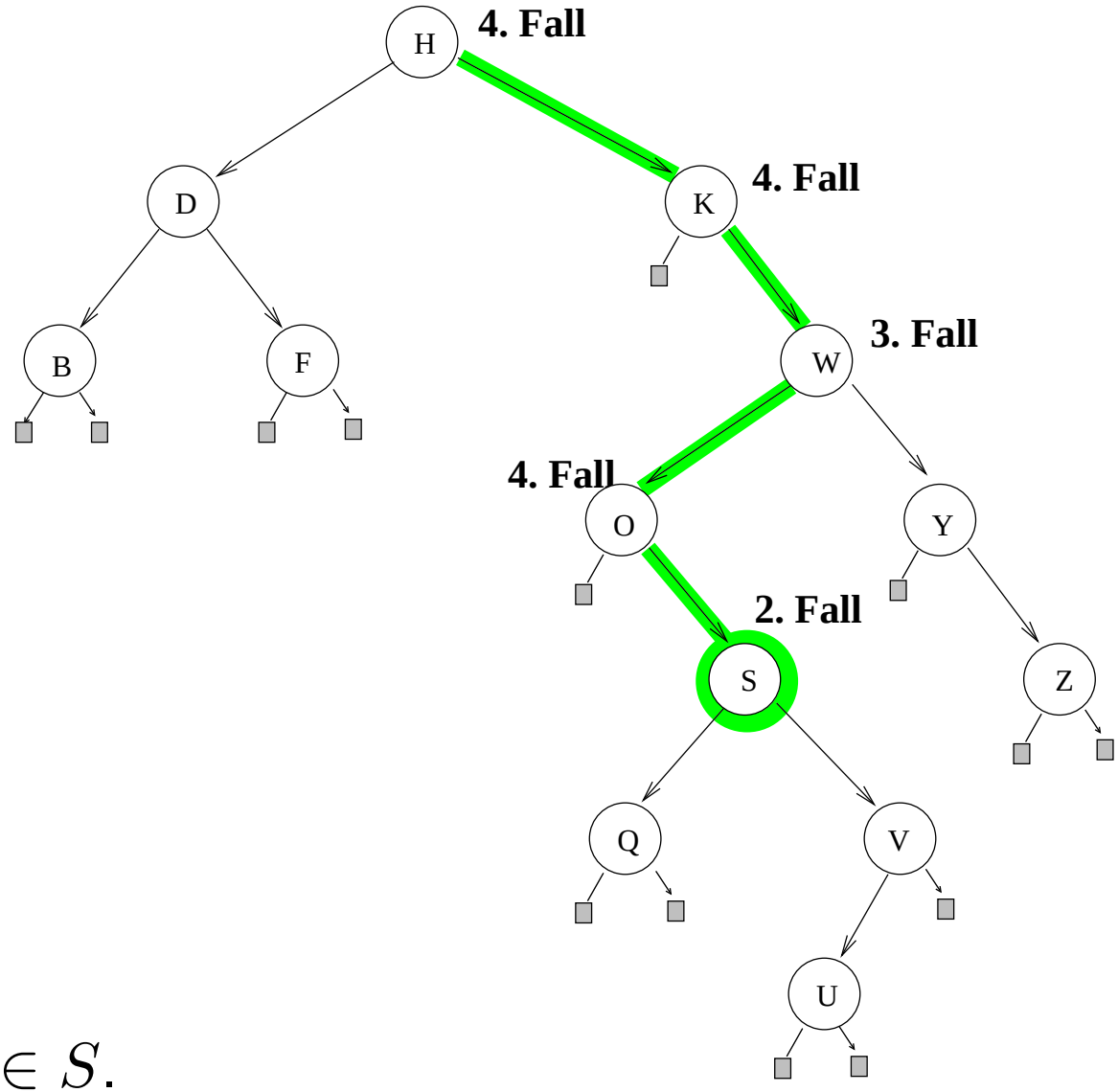
4. Fall: $x > y$.
return lookup$(T.\mathit{right}, x)$ // rekursiver Aufruf .

Beispiel:



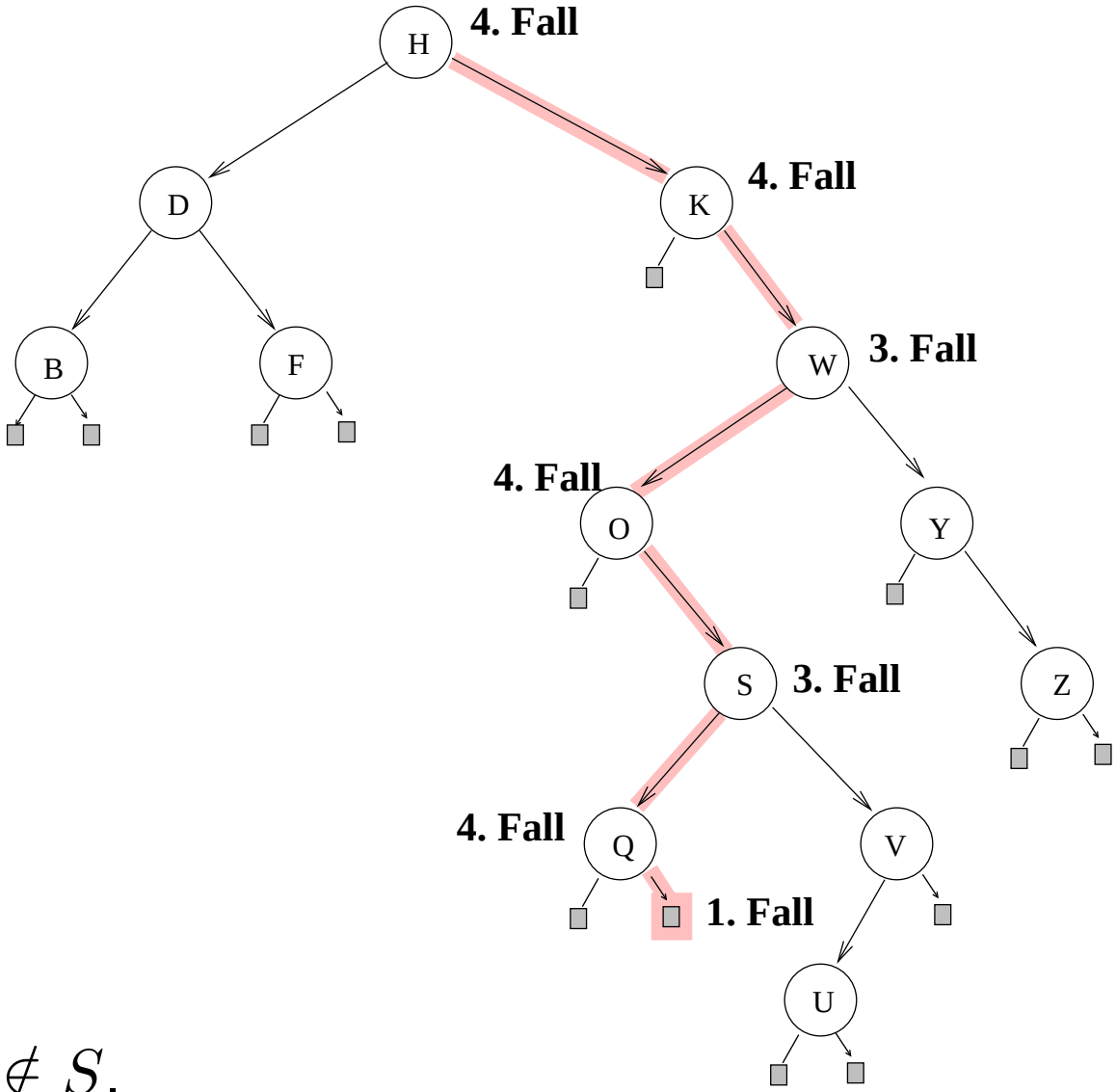
Suche: $\mathrm{Q} \in S$.

Beispiel:



Suche: $\mathrm{S} \in S$.

Beispiel:



Suche: $\mathrm{R} \notin S$.

Korrektheit folgt sofort aus der Struktur von BSB und der rekursiven Struktur des Algorithmus.

(Formal: Per Induktion über den Aufbau von Binärbäumen zeigt man: Ausgabe von **lookup**(T) ist korrekt.)

Zeitaufwand: Wenn x im Knoten v_x sitzt:
Für jeden Knoten v auf dem Weg von der Wurzel zu v_x entstehen Kosten $O(1)$.

Es gibt $d(v_x) \leq d(T)$ solche Knoten.

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

Es gibt $d(l_x) \leq d(T) + 1$ solche Knoten.

Übung: Suche **iterativ** programmieren.

Einfügen, rekursiv

insert(T, x, r), für BSB T, $x \in U, r \in R$. // T: Zeiger

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

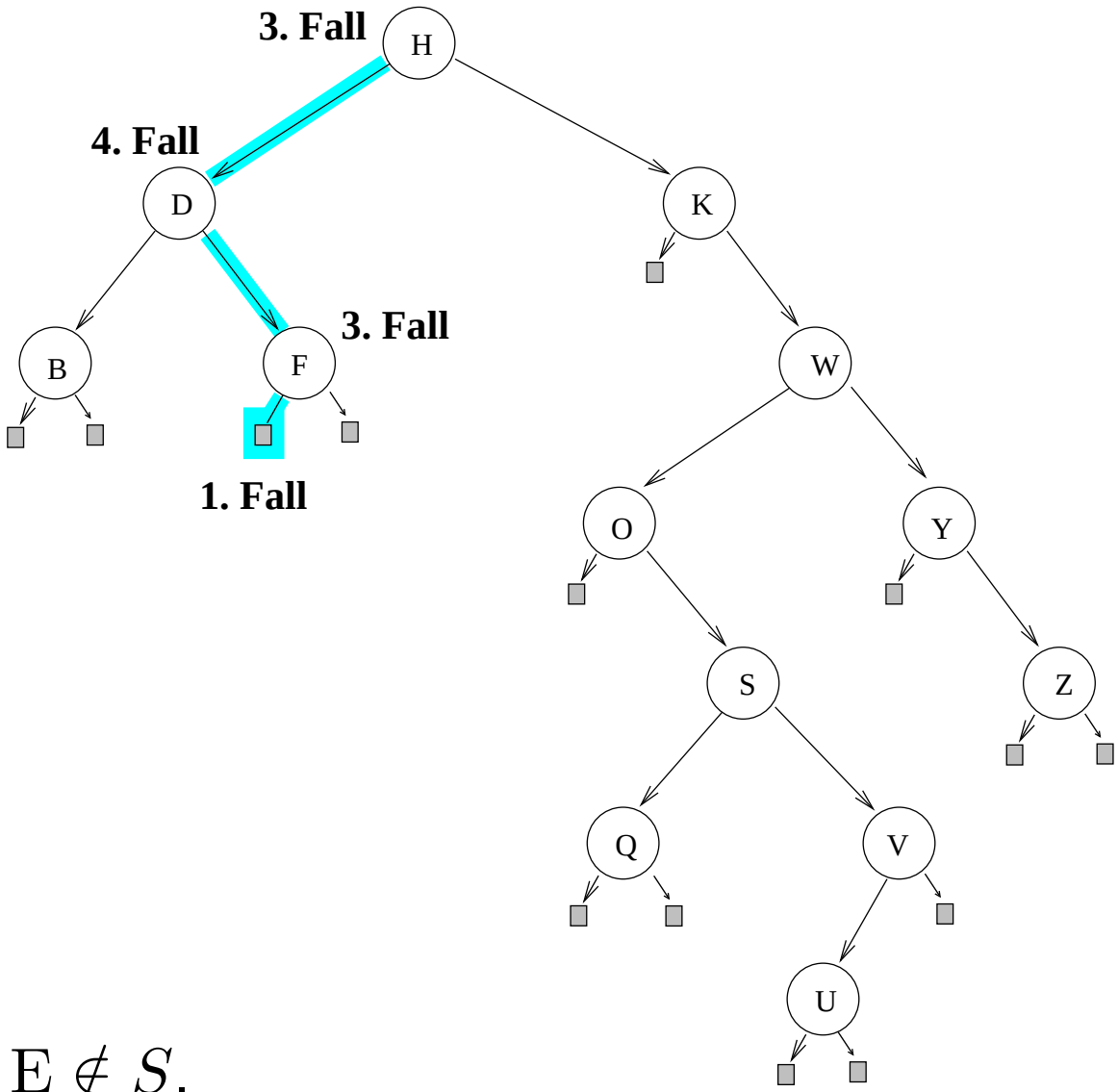
Ab hier: $T = (T_1, (y, r'), T_2)$.

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

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

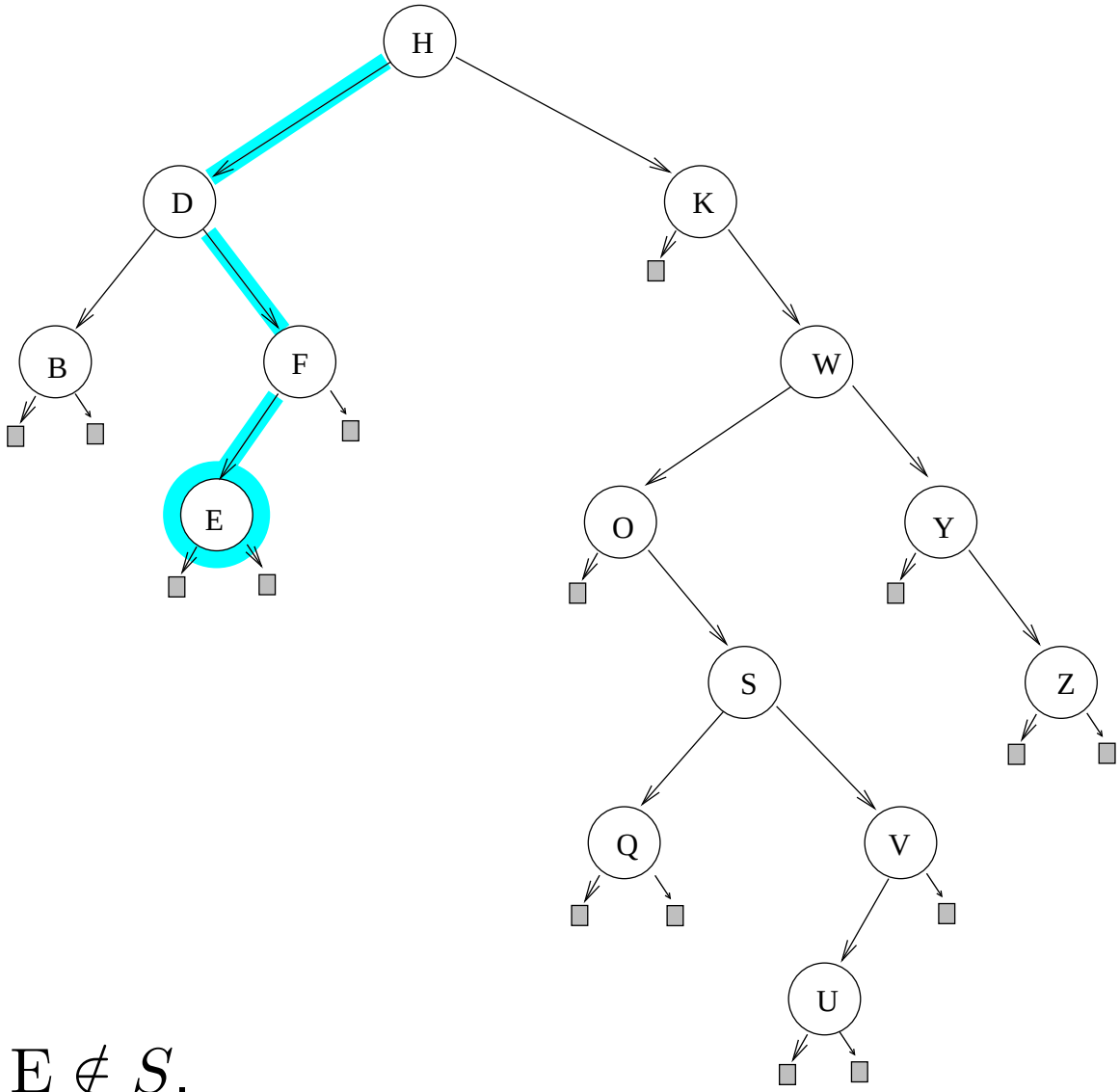
4. **Fall:** $y < x$.
   $T.right \leftarrow$ **insert**$(T.right, x, r)$; **return** T.

Beispiel:



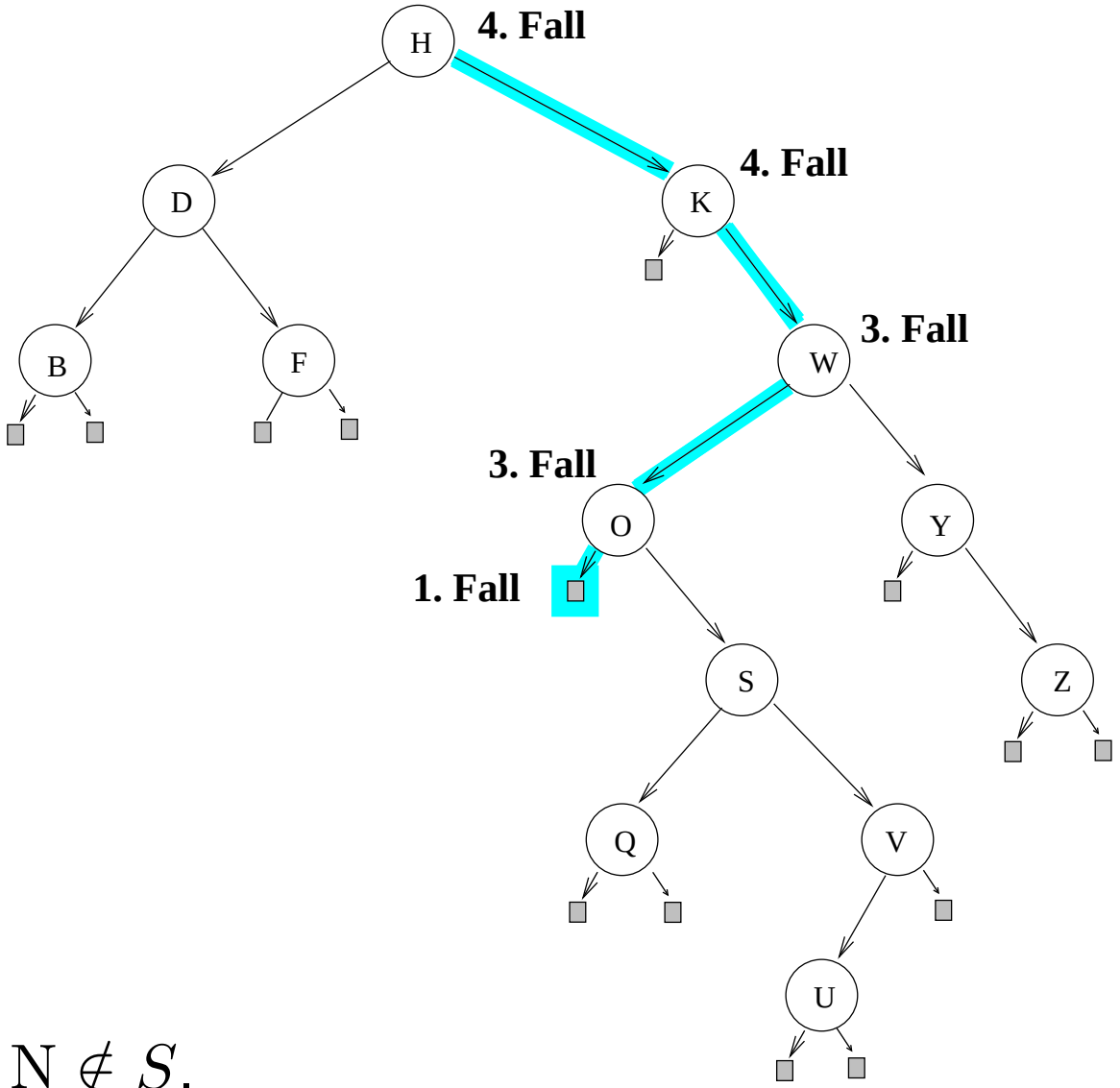
Füge ein: $\mathrm{E} \notin S$.

Beispiel:



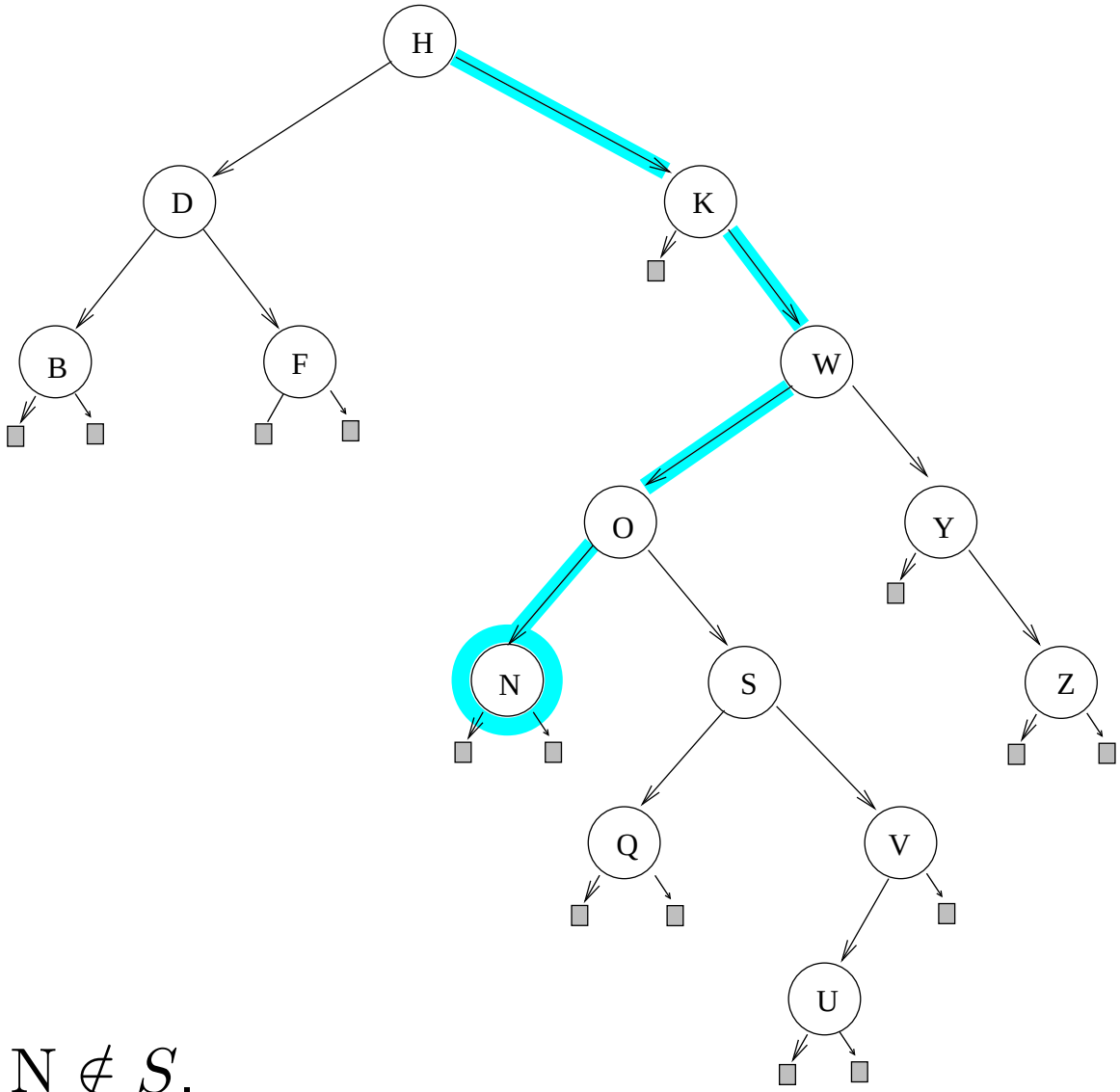
Füge ein: $\mathrm{E} \notin S$.

Beispiel:



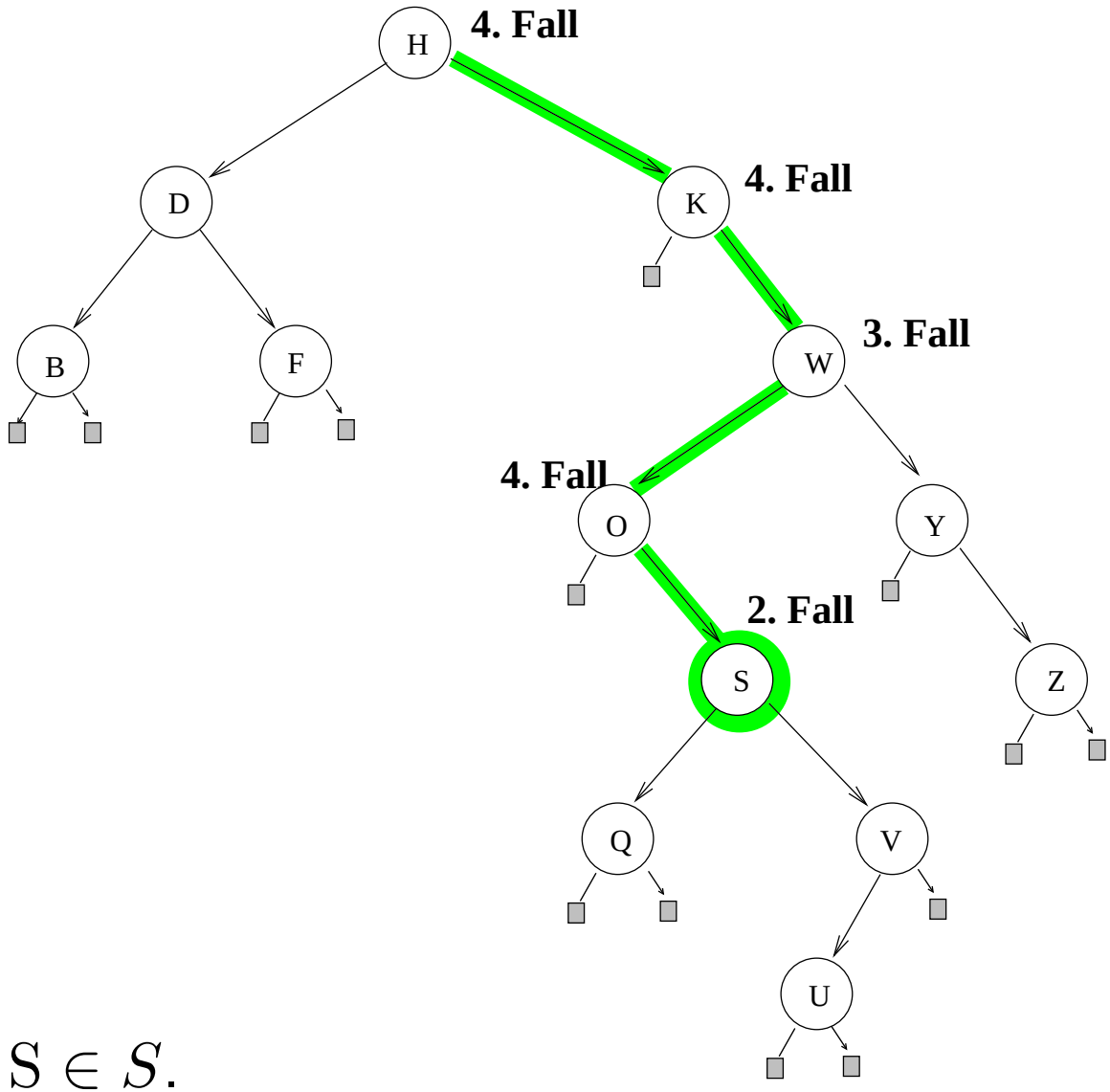
Füge ein: $\mathrm{N} \notin S$.

Beispiel:



Füge ein: $\mathrm{N} \notin S$.

Beispiel:



Füge ein: $\mathrm{S} \in S$.

Korrektheit:

Behauptung: Aufruf **insert**(T, x, r) erzeugt einen binären Suchbaum (der wieder T heißt),
der das modifizierte Wörterbuch *insert*(f_T, x, r) darstellt.

Beweis: Induktion über den Aufbau von Binärbäumen.

Zeitaufwand: Wenn x im Knoten v_x schon vorhanden:
Kosten $O(1)$ für jeden von $d(v_x) \leq d(T)$ Knoten auf dem Weg.

Wenn x in T nicht vorhanden:
Kosten $O(1)$ für jeden Knoten auf dem Weg von Wurzel zu l_x.
Es gibt $d(l_x) \leq d(T) + 1$ solche Knoten.

Übung: Einfügung **iterativ** programmieren.

Löschen, rekursiv

delete(T, x), für BSB T, $x \in U$.

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

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

2. Fall: $x < y$.

$T.\mathit{left} \leftarrow$ **delete**$(T.\mathit{left}, x)$; **return** T;

3. Fall: $y < x$.

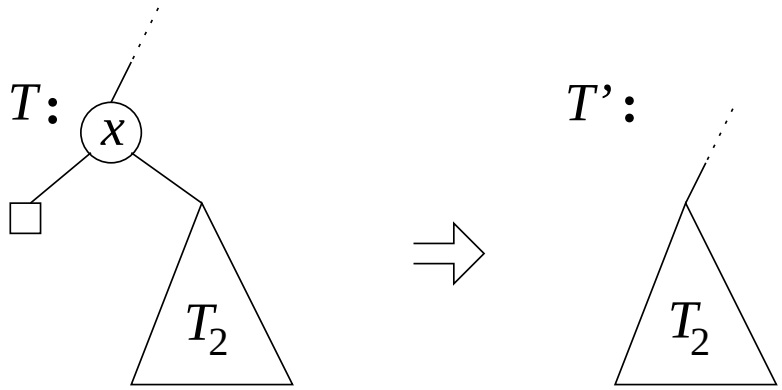
$T.\mathit{right} \leftarrow$ **delete**$(T.\mathit{right}, x)$; **return** T;

4. Fall: $x = y$.

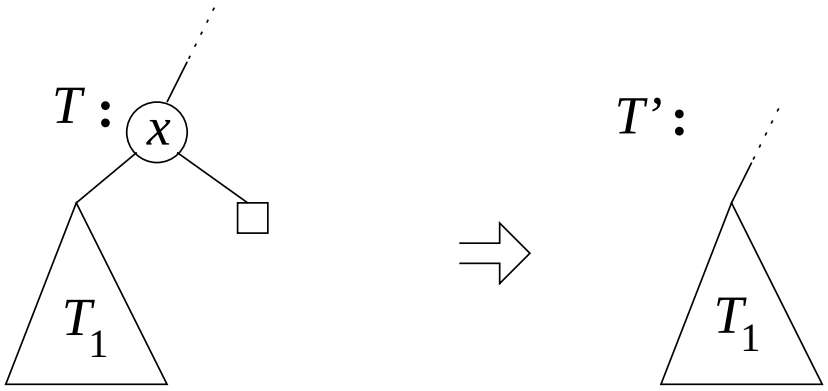
// Lösche Wurzel! Unterfälle 4a–4c, s.u.

return T.

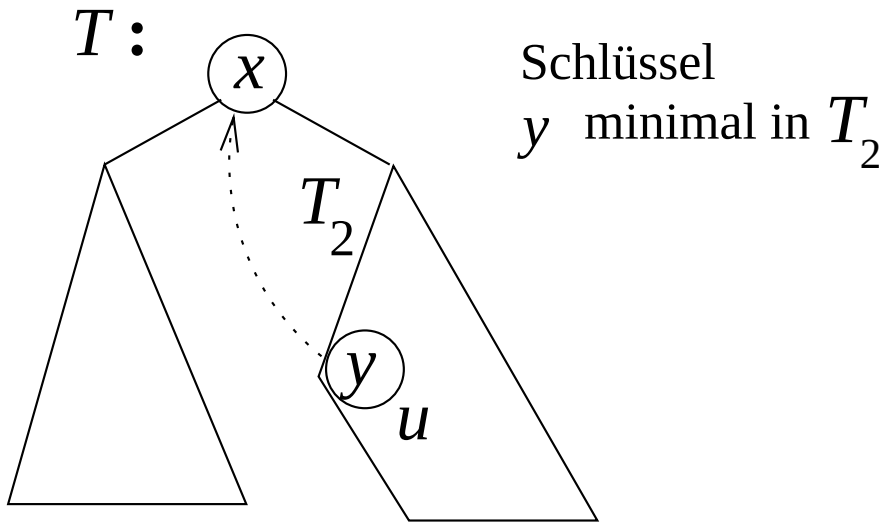
Fall 4a: $T.left = \square$. **return** $T.right$.
(auch wenn $T.right = \square$ ist!)



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



Fall 4c: $T_1 \neq \square$ und $T_2 \neq \square$.



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

(Dies ist der **Inorder-Nachfolger** der Wurzel von T.)

$T_2' \leftarrow T_2$ ohne u; T' besteht aus u als Wurzel, T_1 als linkem und T_2' als rechtem Unterbaum.

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

Extrahieren des Minimums, rekursiv

Extrahieren des Knotens mit kleinstem Schlüssel aus einem **nichtleeren** BSB $T = (T_1, (z, r), T_2)$: **extractMin**(T).

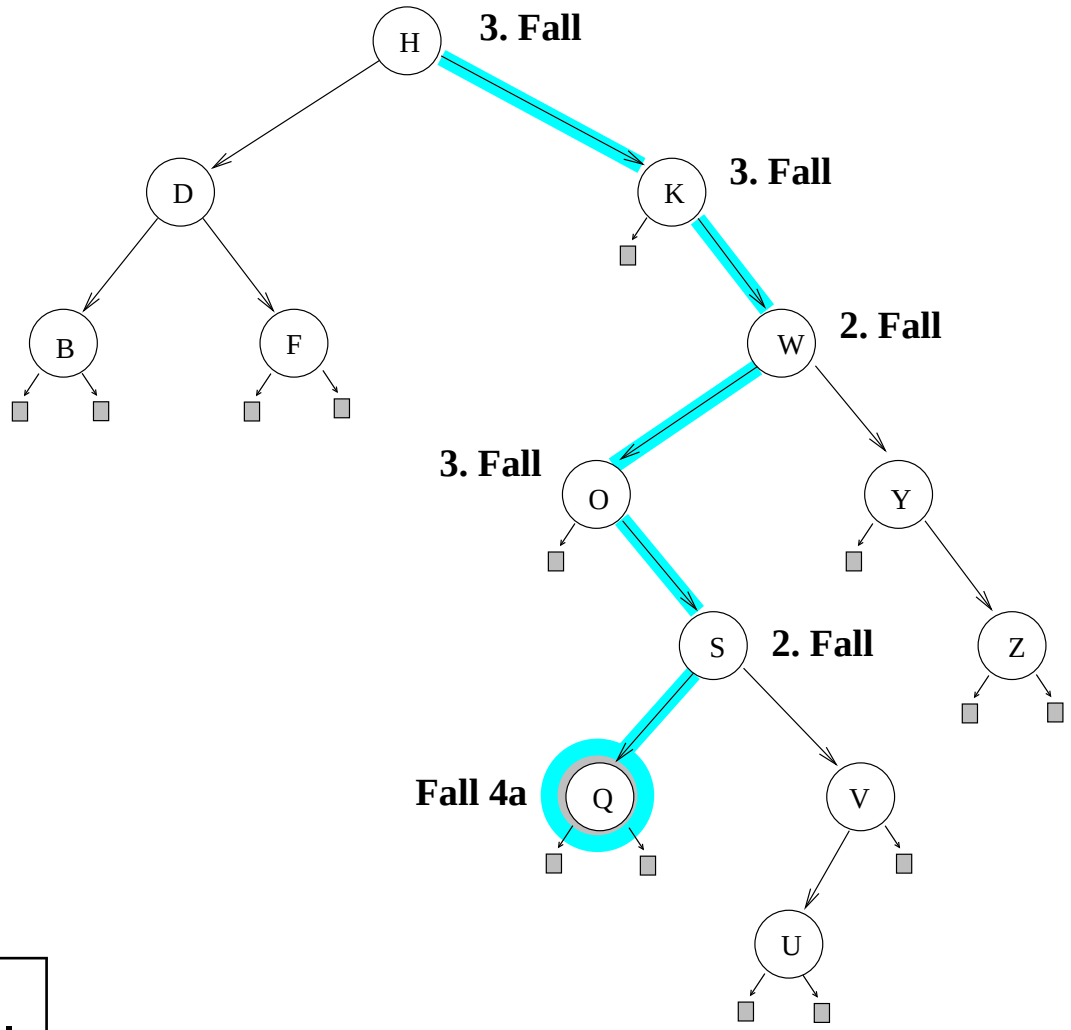
Ebenfalls rekursiv zu realisieren.

Rückgabewerte: Veränderter Baum T' und ein separater Baumknoten (via Zeiger auf diesen Knoten).

1. **Fall:** $T.\mathit{left} = \square$. Schlüssel z in der Wurzel ist minimal in T.
   $T' \leftarrow T.\mathit{right}$; $T.\mathit{right} \leftarrow \square$;
   return([Baum] T', [Knoten] T).
2. **Fall:** $T.\mathit{left} \neq \square$.

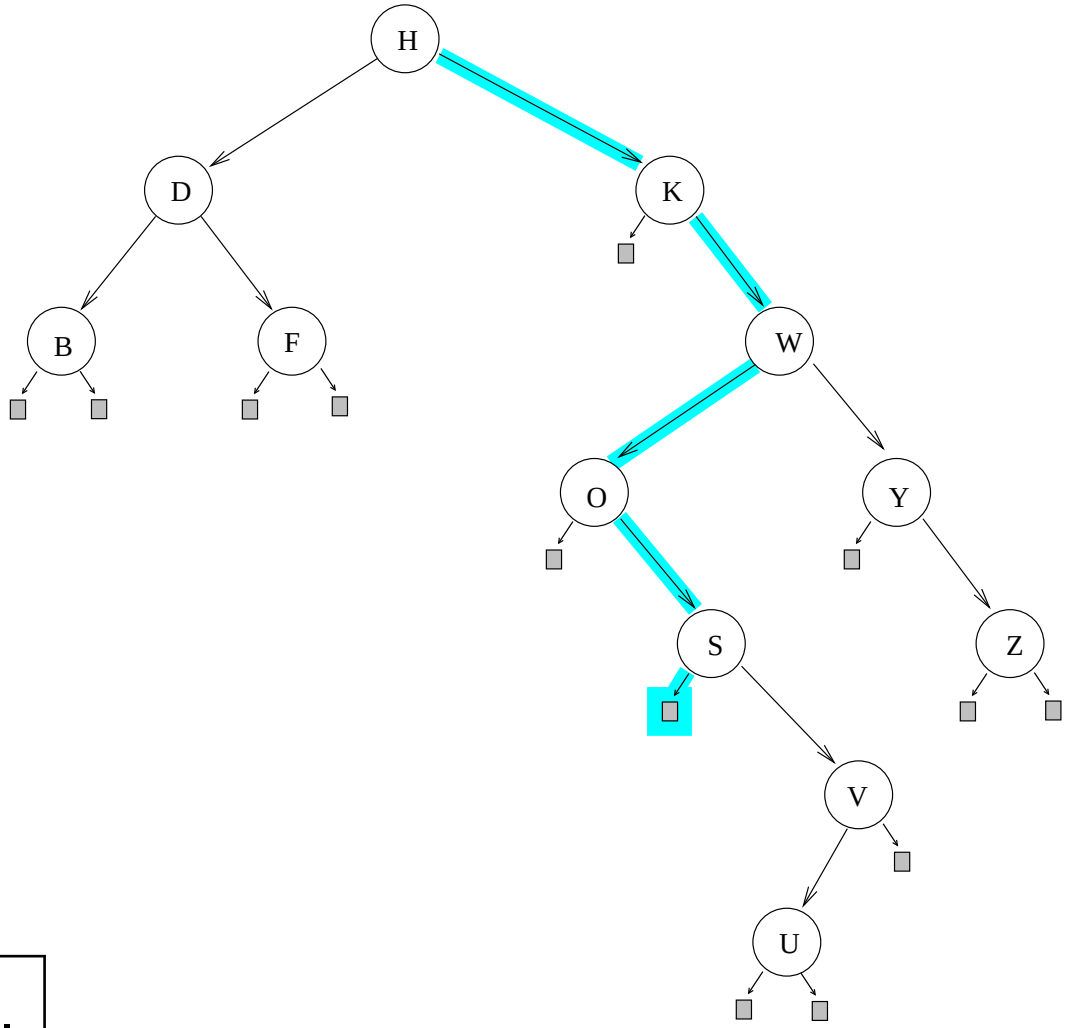
   $(T.\mathit{left}, u) \leftarrow$ **extractMin**$(T.\mathit{left})$;
   return(T, u).

Beispiel:



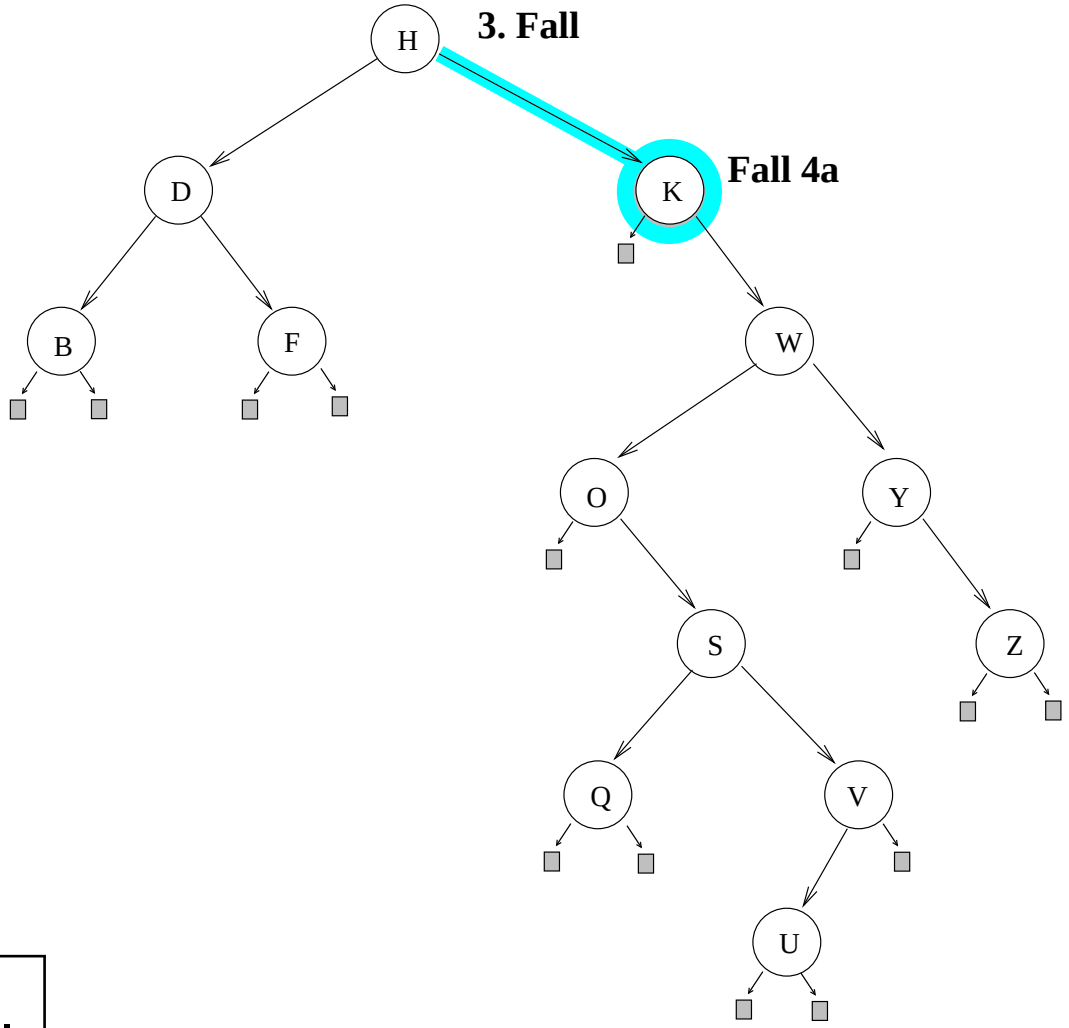
Lösche: Q.

Beispiel:



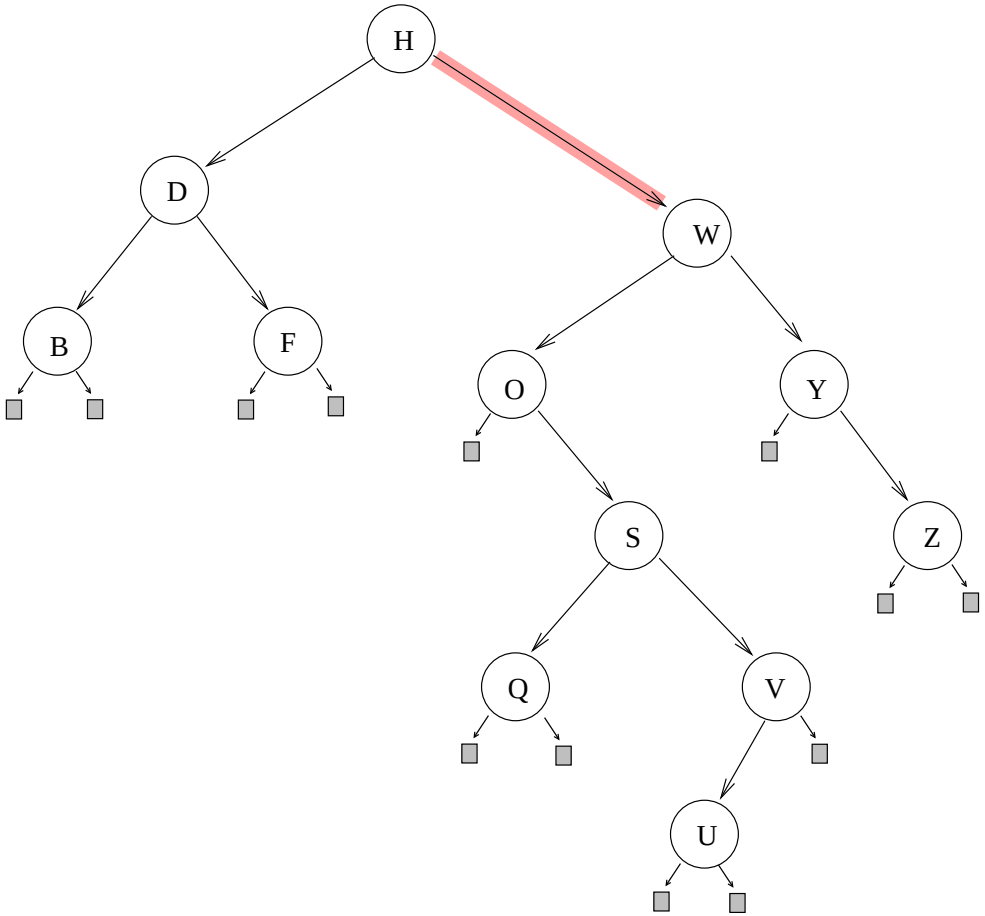
Lösche: Q.

Beispiel:



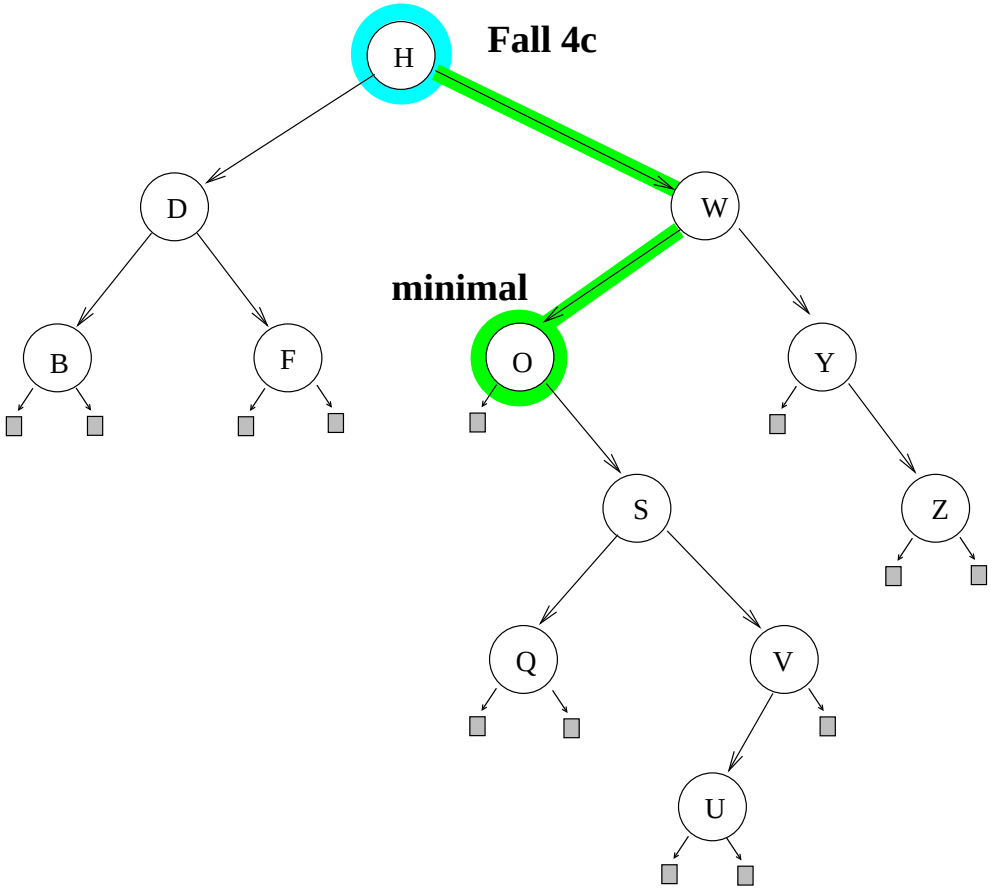
Lösche: K.

Beispiel:



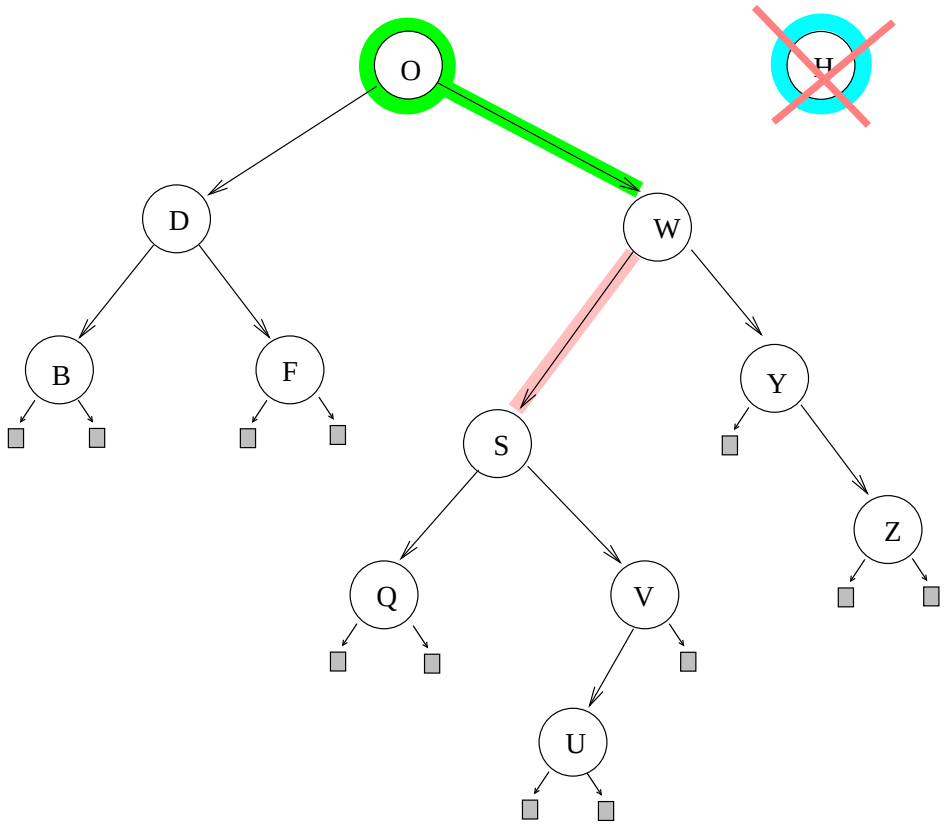
Lösche: K.

Beispiel:



Lösche: H.

Beispiel:



Lösche: H.

Intuitive, **iterative** Beschreibung des Löschvorgangs:

1) Suche Schlüssel x in T.

2) Falls nicht gefunden, ist nichts zu tun.

3) Falls x in Knoten v, gibt es drei Fälle:

3a) v hat keinen linken Unterbaum.

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

Achtung! Der Fall, dass v Blatt ist, ist in 3a) inbegriffen.

3b) v hat keinen rechten Unterbaum.

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

3c) v hat linken **und** rechten Unterbaum.

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!

Entferne u aus seinem Unterbaum durch „Umbiegen“ des Zeigers vom Vater von u auf den rechten Unterbaum an u.

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

Knoten v ist jetzt „frei“, kann gelöscht werden.

Übung: Löschung **iterativ** programmieren.

Korrektheit:

Behauptung: Aufruf *delete*(T, x) erzeugt einen binären Suchbaum (der wieder T heißt),
der das modifizierte Wörterbuch *delete*(f_T, x) darstellt.

Beweis: Induktion über den Aufbau von Binärbäumen.

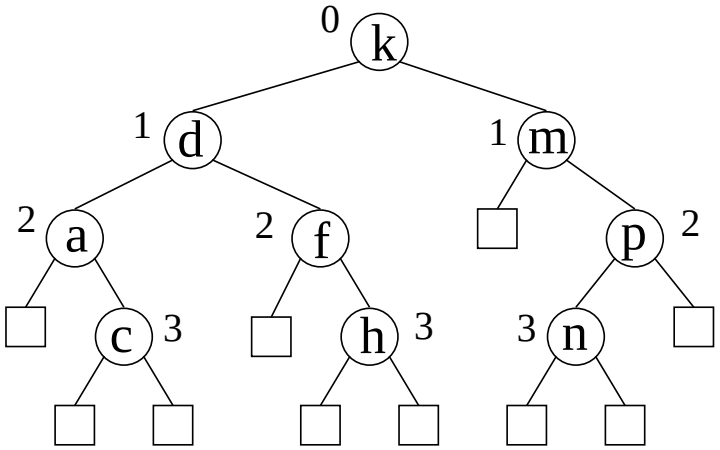
Dabei muss man gesondert beweisen, und dann benutzen, dass die Prozedur **extractMin**(T), angewendet auf einen nichtleeren BSB, das verlangte Resultat liefert.

Zeitaufwand:

- Wenn x im Knoten v_x vorhanden, Fälle 2 und 3:
  Kosten $O(1)$ für jeden von $d(v_x) \leq d(T)$ Knoten auf dem Weg.
- Wenn x im Knoten v_x vorhanden, Fall 4:
  Kosten $O(1)$ für jeden von $d(v_x^+) \leq d(T)$ Knoten auf dem Weg zum **Inorder-Nachfolger** v_x^+ von v_x.
- Wenn x im Baum T nicht vorhanden, Fall 1:
  Kosten $O(1)$ für jeden der $d(l_x) \leq d(T) + 1$ Knoten auf dem Weg zu l_x.

4.2 Erwartete mittlere Tiefe in zufälligen BSBs

Schon gesehen:



Summe der Tiefen: $0 + 2 \cdot 1 + 3 \cdot 2 + 3 \cdot 3 = 17$

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

Mittlere innere Weglänge: $\frac{1}{n}\mathrm{TIPL}(T)$.

Zufällig erzeugte binäre Suchbäume

T **zufällig erzeugt**: Starte mit leerem Baum, füge Schlüssel aus Menge $S = \{x_1, x_2, \ldots, x_n\}$ ein.
Eingabereihenfolge der Schlüssel $x_1 < x_2 < \cdots < x_n$ ist „rein zufällig“ – jede Reihenfolge hat Wahrscheinlichkeit $1/n!$.

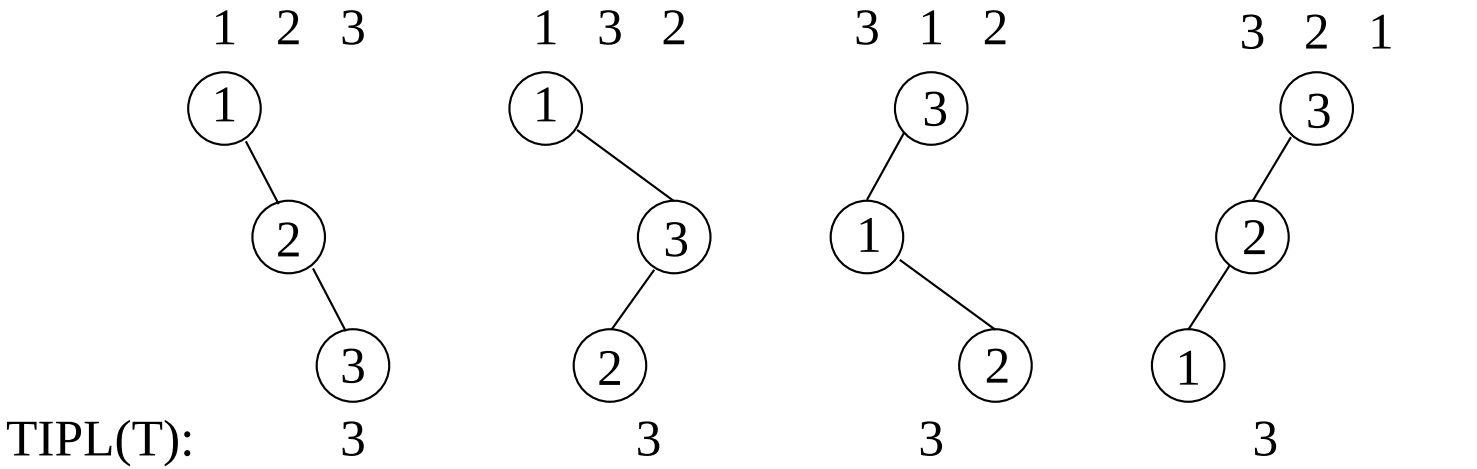
$$\boldsymbol{A(n)} := \mathbf{E}(\mathrm{TIPL}(T)).$$

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

$$O\left(1 + \frac{1}{n} A(n)\right).$$

Was ist $A(n)$? $\quad A(0) = 0, \;\; A(1) = 0, \;\; A(2) = 1.$

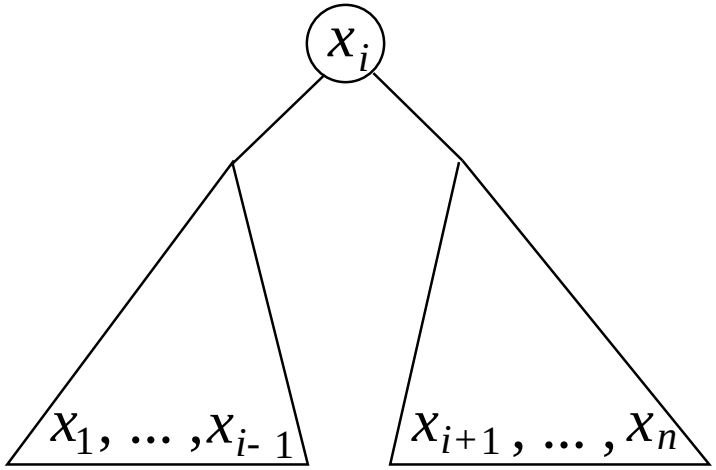
$A(3) =$? – 6 Einfügereihenfolgen für $(x_1, x_2, x_3) = (1, 2, 3)$:



$$A(3) = \mathbf{E}(\mathrm{TIPL}(T)) = \tfrac{1}{6}\,(3+3+3+3+2+2) = \tfrac{8}{3}.$$

Rekursionsformel

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



Die Unterbäume sind selbst zufällig erzeugte binäre Suchbäume. Also: $\mathbf{E}(\text{TIPL}(T) \mid \textbf{falls } x_i \text{ erster}) = \ldots$

$$\ldots = ((i-1) + A(i-1)) + ((n-i) + A(n-i)).$$

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 $(i-1)$, $(n-i)$.

Vereinfachung:

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

Jeder der n Schlüssel $x_1, \ldots, x_n$ hat dieselbe Wahrscheinlichkeit $1/n$, als erster eingefügt zu werden und die Wurzel zu bilden.

Also Mittelung:

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

Das heißt:

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

Umgruppieren und Beachtung der Gleichung $A(0) = 0$ liefert:

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

Damit:

$$\begin{aligned}
A(1) &= (1-1) + \tfrac{2}{1} \textstyle\sum_{1 \leq j \leq 0} A(j) && = 0; \\
A(2) &= (2-1) + \tfrac{2}{2}(0) && = 1; \\
A(3) &= (3-1) + \tfrac{2}{3}(0+1) && = \tfrac{8}{3}; \\
A(4) &= (4-1) + \tfrac{2}{4}(0+1+\tfrac{8}{3}) && = \tfrac{29}{6}; \\
A(5) &= (5-1) + \tfrac{2}{5}(0+1+\tfrac{8}{3}+\tfrac{29}{6}) && = \tfrac{37}{5};
\end{aligned}$$

Geschlossene Form?

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

Multiplizieren mit n:

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

Dasselbe für $n-1$:

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

Subtrahieren:

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

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

D.h.:

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

Teile durch $n(n+1)$:

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

Abkürzung: $Z(n) := A(n)/(n+1)$.

Damit: $Z(0) = 0$ und

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

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

Also, für $n \geq 0$:

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

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

und

$\frac{1}{1\cdot 2} + \frac{1}{2\cdot 3} + \cdots + \frac{1}{n\cdot(n+1)} = (1-\frac{1}{2})+(\frac{1}{2}-\frac{1}{3})+\cdots+(\frac{1}{n}-\frac{1}{n+1}) = 1-\frac{1}{n+1}$

gibt das:

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

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

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

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

Erwartete mittlere Knotentiefe

(Einfügereihenfolge **zufällig**; **Mittelung** über $x_1, \ldots, x_n$):

Teile durch n.

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

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:

$H_n - 1 = \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n} < \int_1^n \frac{dt}{t} = \ln n.$

$\ln n = \int_1^n \frac{dt}{t} \leq 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n-1} < H_n.$

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

Genauer (ohne Beweis): $(H_n - \ln n) \nearrow \gamma$, für $n \to \infty$,

wobei $\gamma = 0{,}57721\ldots$ („Euler-Konstante“).

Hatten schon:

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

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

Satz 4.2.1

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

Wenn man Zweierlogarithmen lieber mag:

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

Dabei: $2\ln 2 = \mathbf{1{,}386\ldots}$

Satz 4.2.1 (Kurzform)
In zufällig erzeugten „natürlichen“ binären Suchbäumen ist die **erwartete mittlere** Knotentiefe $1{,}39 \log n + O(1)$.

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

Leider: Normalerweise Einfügereihenfolge **nicht zufällig**.

Ungünstige Reihenfolge: Daten partiell sortiert.

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.

4.3 Balancierte binäre Suchbäume

Idee:

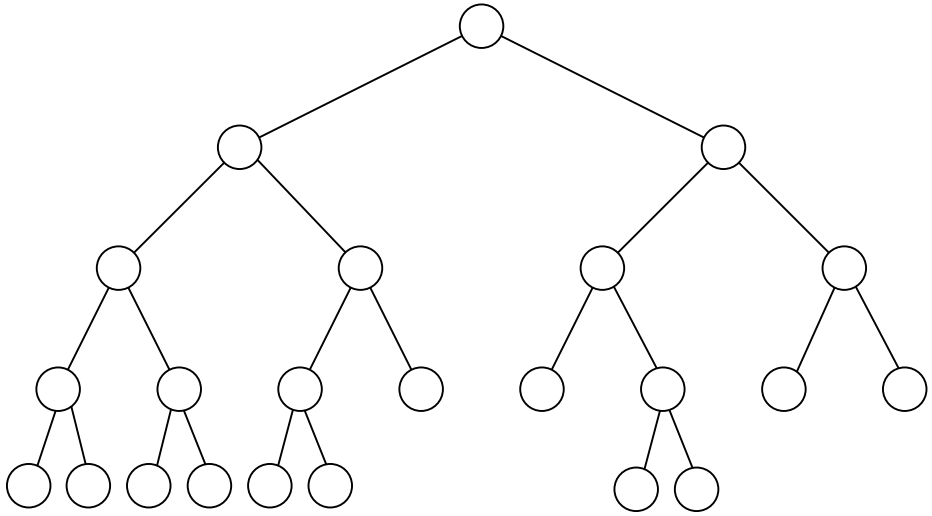
- Lasse nur Bäume zu, die bestimmte **Strukturbedingungen** erfüllen.
- Strukturbedingungen **erzwingen geringe Tiefe** (z. B. $O(\log n)$ bei n Einträgen).
- **Implementiere** Wörterbuch-Operationen

  insert und *delete*,

  so dass sie die **Strukturbedingungen erhalten** und in Zeit $O(\text{Baumtiefe})$ durchführbar sind.

?? 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 $d = \lceil \log(n+1) \rceil - 1$.

(Dies ist auch $|\mathrm{bin}(n)| - 1$.)

Attraktiv ??

Ungeeignet, da Einfüge- und Lösch-Operationen zu teuer.

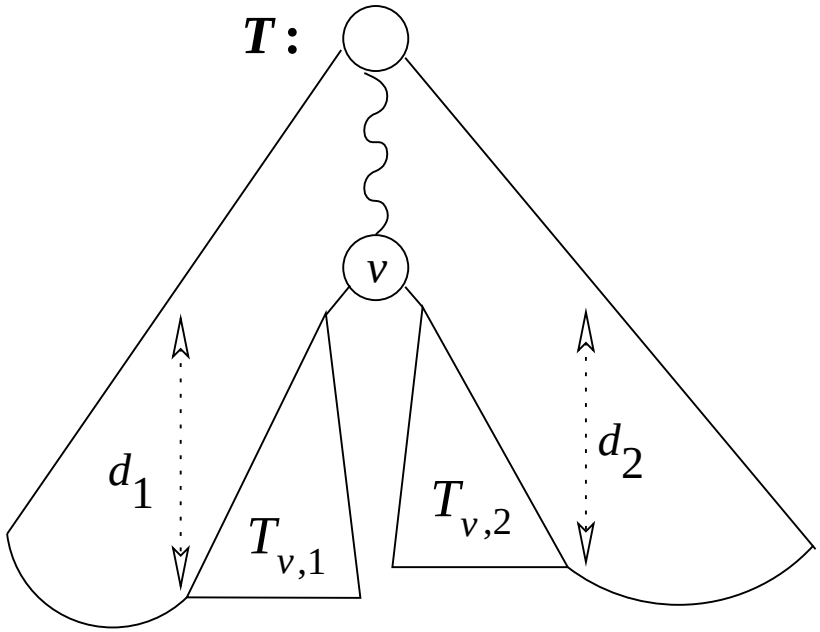
Häufig benutzte Strukturbedingungen:

- **AVL-Bäume**
- **Rot-Schwarz-Bäume** [Cormen et al.], [Sedgewick]
- **2-3-Bäume**
- **2-3-4-Bäume** [Cormen et al.], [Sedgewick], [D./Mehlhorn/Sanders]
- **B-Bäume** [Cormen et al.]

4.3.1 AVL-Bäume

Höhenbalancierte binäre Suchbäume

[G. M. **A**delson-**V**elski und J. M. **L**andis 1962]



$$|d_2 - d_1| \leq 1$$

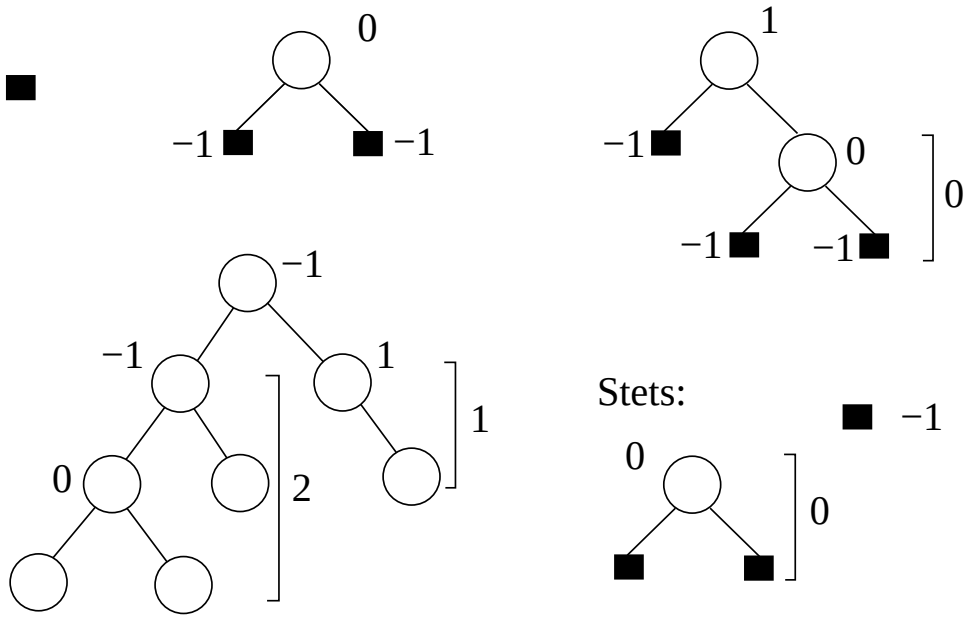
Definition 4.3.1

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:

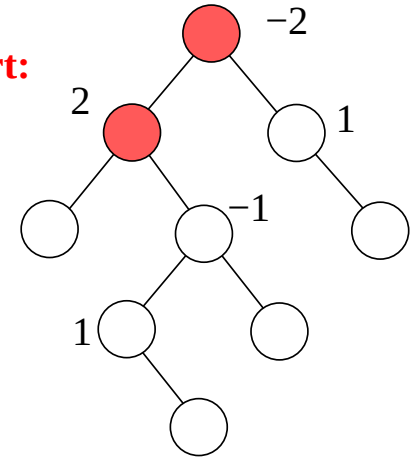
$$\underbrace{d(T_{v,2}) - d(T_{v,1})}_{\text{„Balancefaktor } bal \text{ in } v\text{“}} \in \{-1, 0, 1\} \; .$$

Äquivalent ist die **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:



Nicht höhenbalanciert:



Definition 4.3.2

Ein **höhenbalancierter binärer Suchbaum** heißt **AVL-Baum**.

Für die **Implementierung** von AVL-Bäumen muss man in jedem inneren Knoten v seinen Balancefaktor speichern.

Notation: **bal(v)** ist der Balancefaktor in Knoten v.

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,

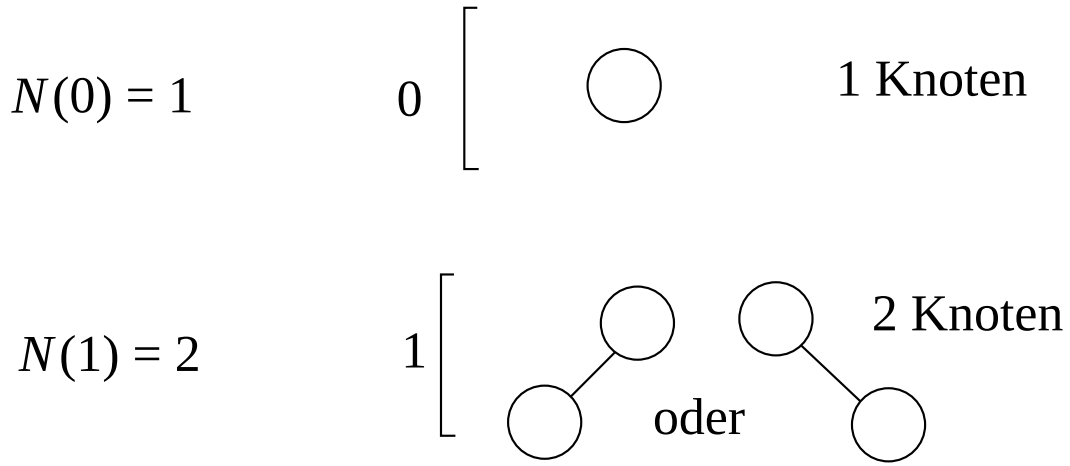
nämlich mindestens Φ^d viele für eine Konstante $\Phi > 1$,

mit $\log_\Phi 2 \leq 1{,}4405$.

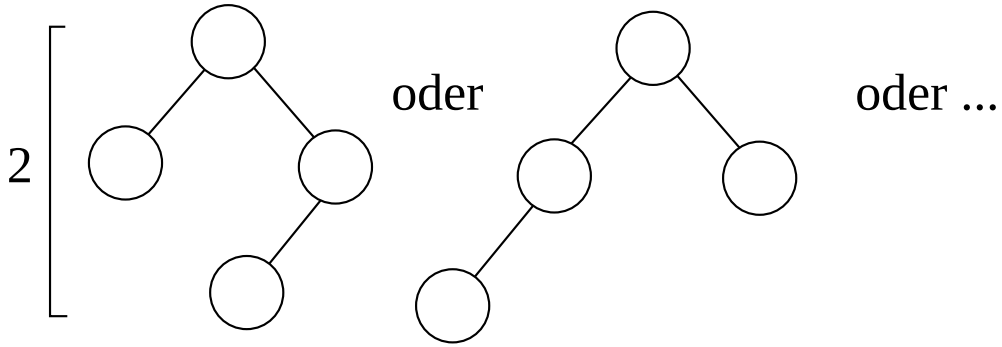
Beweis des Satzes:

Für $d = 0, 1, 2, \ldots$ setze

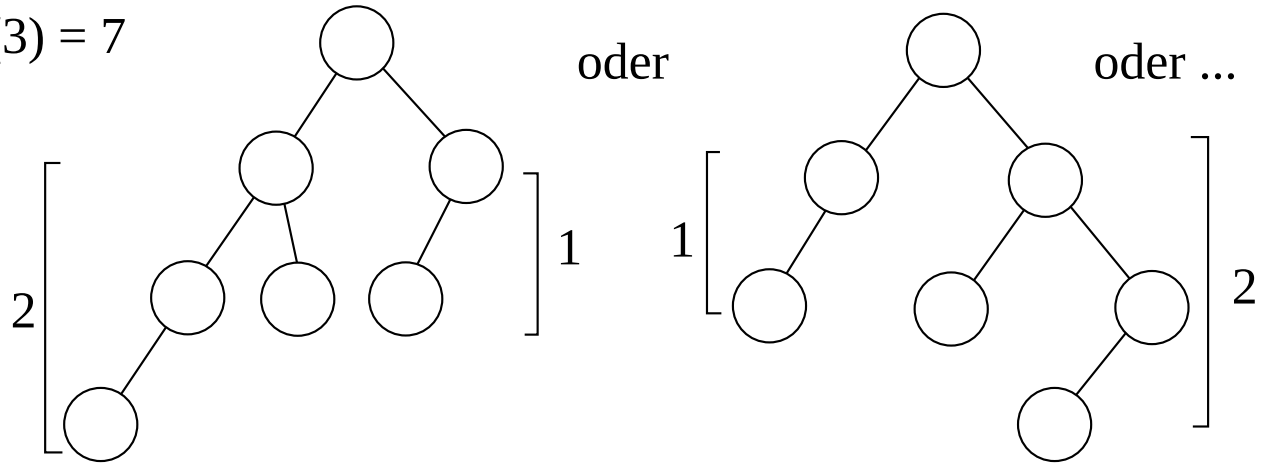
$N(d) :=$ die **minimale Zahl innerer Knoten** in einem AVL-Baum T mit $d(T) = d$.



$N(2) = 4$

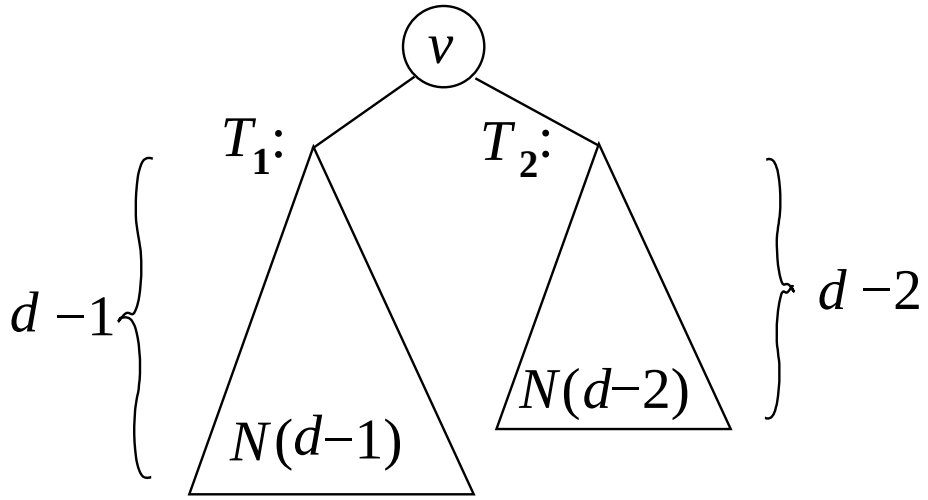


$N(3) = 7$



Behauptung 1:

$N(d) = 1 + N(d-1) + N(d-2)$, für $d \geq 2$.



Beweis: Es ist klar, dass $N(d) > N(d-1)$ für alle d.
Damit der höhenbalancierte Baum der Tiefe d minimale Knotenzahl hat, müssen die Unterbäume Tiefe $d-1$ und $d-2$ haben und selbst minimale Knotenzahl für diese Tiefe haben.

$N(d) = 1 + N(d-1) + N(d-2)$, 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 **Fibonacci-Zahlen** $F_0, F_1, F_2, \ldots$.

Behauptung 2:

$$N(d) \geq \Phi^d \text{ , für } d \geq 0,$$

wo $\Phi = \frac{1}{2}(1+\sqrt{5}) \approx 1{,}618\ldots$ („goldener Schnitt“).

$(*)$ Φ ist Lösung der quadratischen Gleichung $x^2 = x + 1$.

Beweis von Beh. 2: Vollständige Induktion.
Für $d = 0$: $N(0) = 1 = \Phi^0$; für $d = 1$: $N(1) = 2 > \Phi$.
Sei nun $d \geq 2$.

$$\begin{aligned} N(d) &\overset{\text{Beh. 1}}{=} 1 + N(d-1) + N(d-2) \\ &\overset{\text{I.V.}}{\geq} \Phi^{d-1} + \Phi^{d-2} = \Phi^{d-2}(\Phi + 1) \\ &\overset{(*)}{=} \Phi^{d-2} \cdot \Phi^2 = \Phi^d. \end{aligned}$$

Nun sei T ein höhenbalancierter Baum mit Höhe $d(T) \geq 0$ und n Knoten.

Nach Behauptung 2: $\Phi^{d(T)} \leq n$.

Durch **Logarithmieren**:

$d(T) \leq \log_\Phi(n)$, d. h.

$$d(T) \leq \log_\Phi(2) \cdot \log_2(n).$$

Es gilt: $\log_\Phi(2) = (\ln 2)/(\ln \Phi) = 1{,}440420\ldots < 1{,}4405$.

Damit ist Satz 4.3.3 bewiesen. $\square$

AVL-Bäume: höhenbalancierte binäre Suchbäume.
Wissen: haben logarithmische Tiefe.

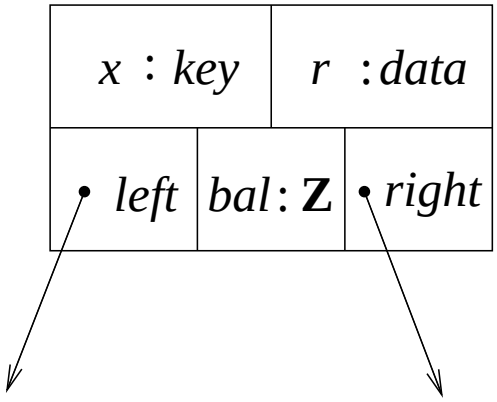
Müssen noch zeigen:

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

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).

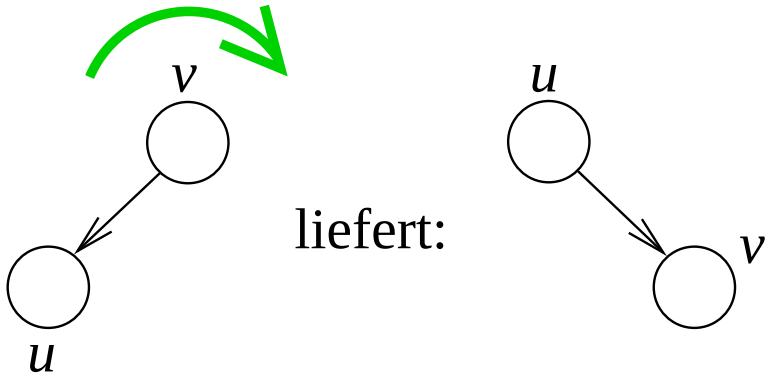
Grundansatz:
Führe Update-Operation aus wie bei gewöhnlichem BSB.

Eventuell wird dadurch Bedingung „Höhenbalancierung“ verletzt.

„Reparatur“: **Rebalancierung**, erfolgt rekursiv.

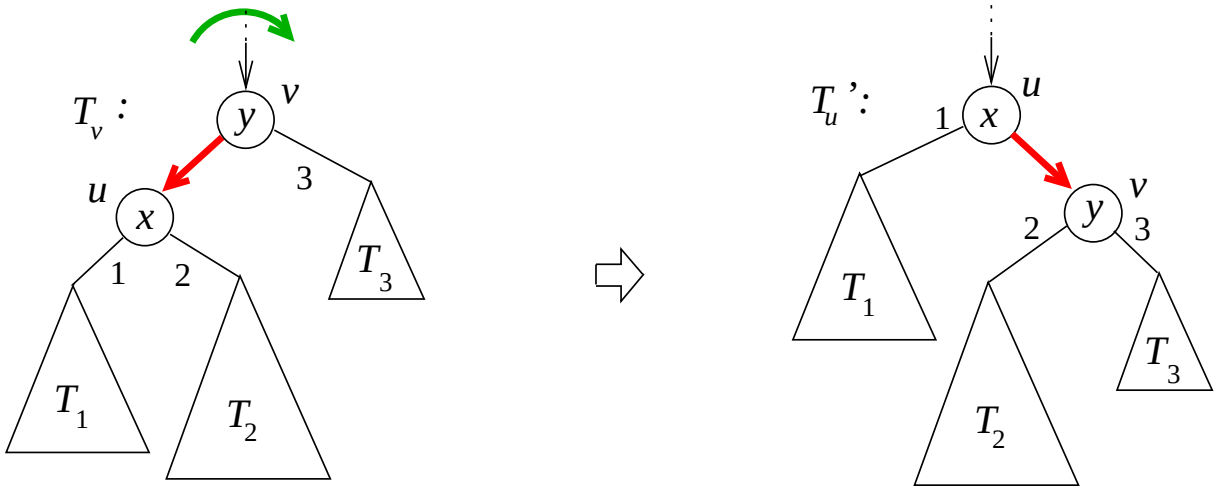
Rotationen: Hilfsoperationen

Rechtsrotation: „kippe 1 Kante nach rechts“:



Rotationen: Hilfsoperationen

Rechtsrotation: Mit Unterbäumen:



Beobachte: Knotenmenge gleich $\wedge$ T'_u binärer Suchbaum.

Denn: Weil T_v binärer Suchbaum ist, gilt für v_1 in T_1, v_2 in T_2, v_3 in T_3: $\mathit{key}(v_1) < x < \mathit{key}(v_2) < y < \mathit{key}(v_3)$. Also ist T'_u auch BSB.

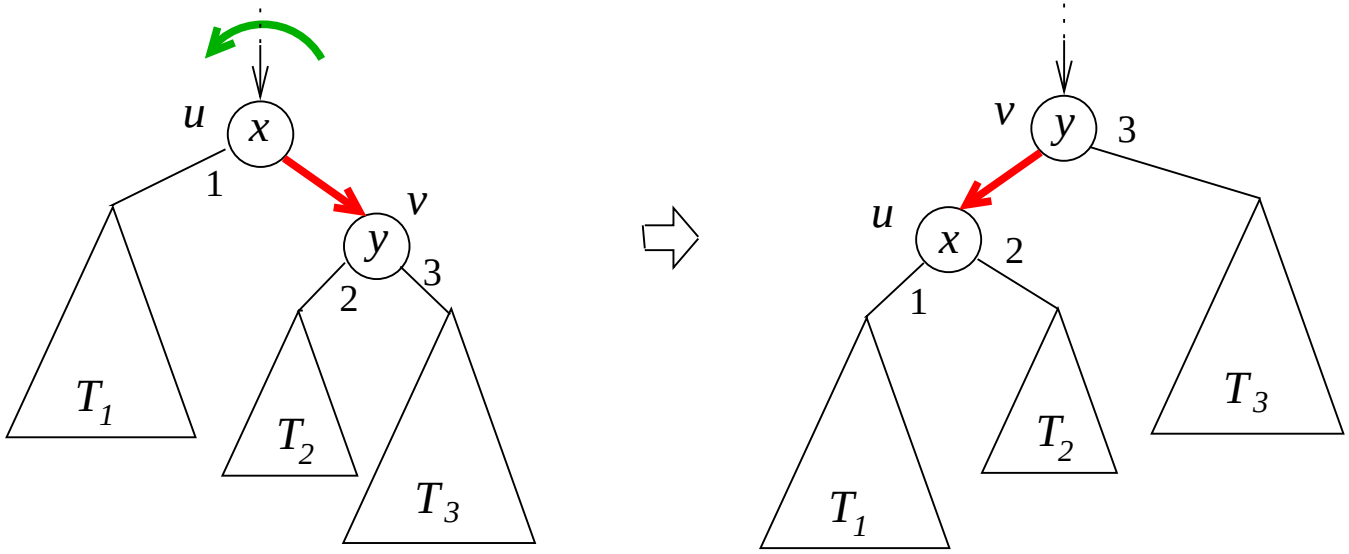
Beim Umbau ändert sich nur: rechtes Kind von u und linkes Kind von v, sowie der Zeiger von außen (von v auf u umsetzen).

Rechtsrotation als Programm:

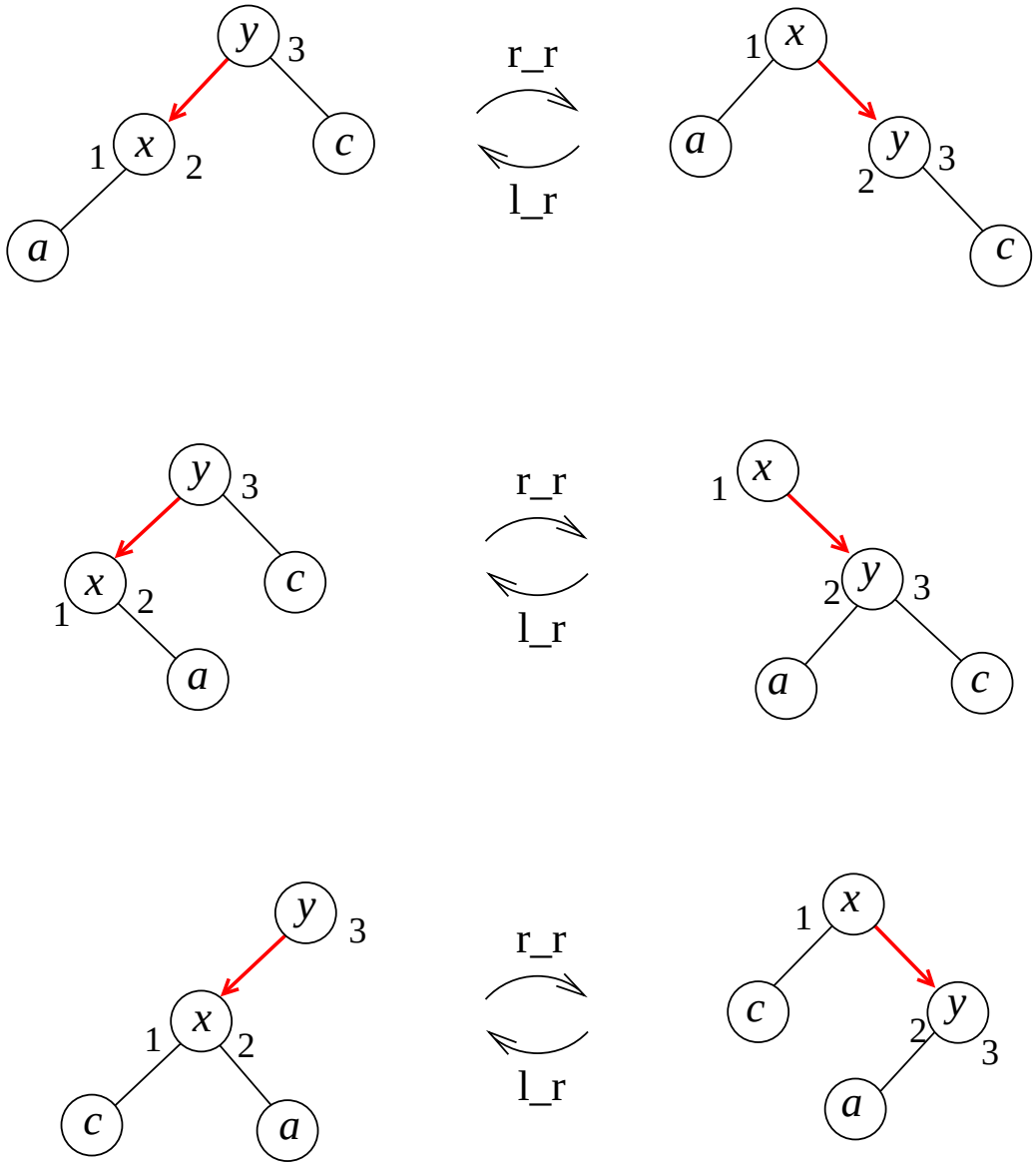
Funktion rotateR(v : AVL_Tree): AVL_Tree

```
(1)       // v ≠ □, v.left ≠ □
(2)   u : AVL_Tree ;  // Hilfsvariable
(3)   u ← v.left ;
(4)   v.left ← u.right ;
(5)   u.right ← v ;
(6)   return u
(7)    // !! Balancefaktoren in u und v sind falsch!
```

Linksrotation: Umkehrung von Rechtsrotation,
Implementierung analog.

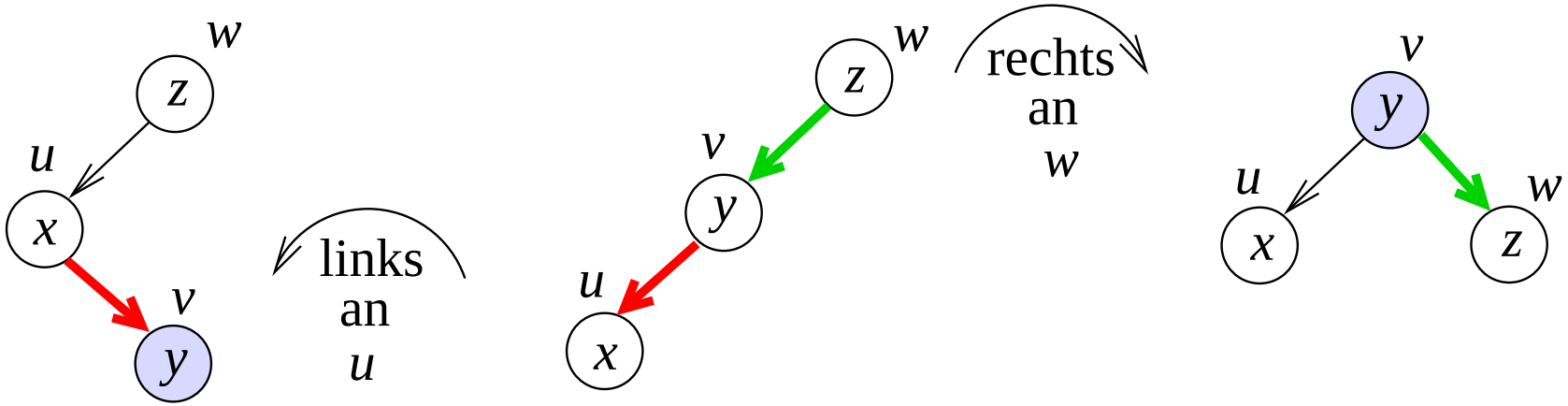


Hier und in den folgenden *Beispielen*:
Nummern bezeichnen Anschlussstellen für die Unterbäume.


y
3
x
1
2
c
a
r_r
l_r
x
1
a
y
2
3
c
y
3
x
1
2
c
a
r_r
l_r
x
1
y
2
3
a
c
y
3
x
1
2
c
a
r_r
l_r
x
1
c
y
2
3
a

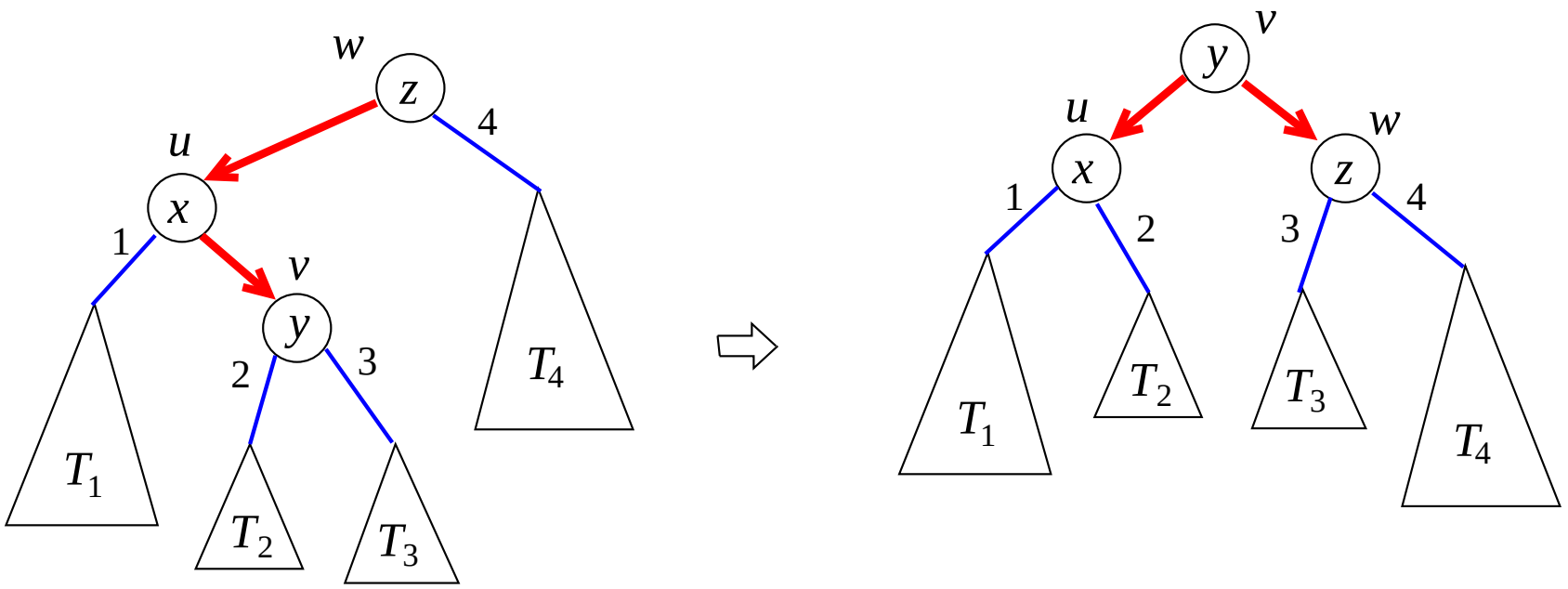
Doppelrotationen: Links-Rechts, Rechts-Links

Beispiel: **Links-Rechts-Doppelrotation**



Anzuwenden auf Zick-Zack-Weg aus zwei Kanten,
Form „**links-rechts**“.
Effekt: Tiefster Knoten v wird Wurzel.

Links-Rechts-Doppelrotation: Mit Unterbäumen



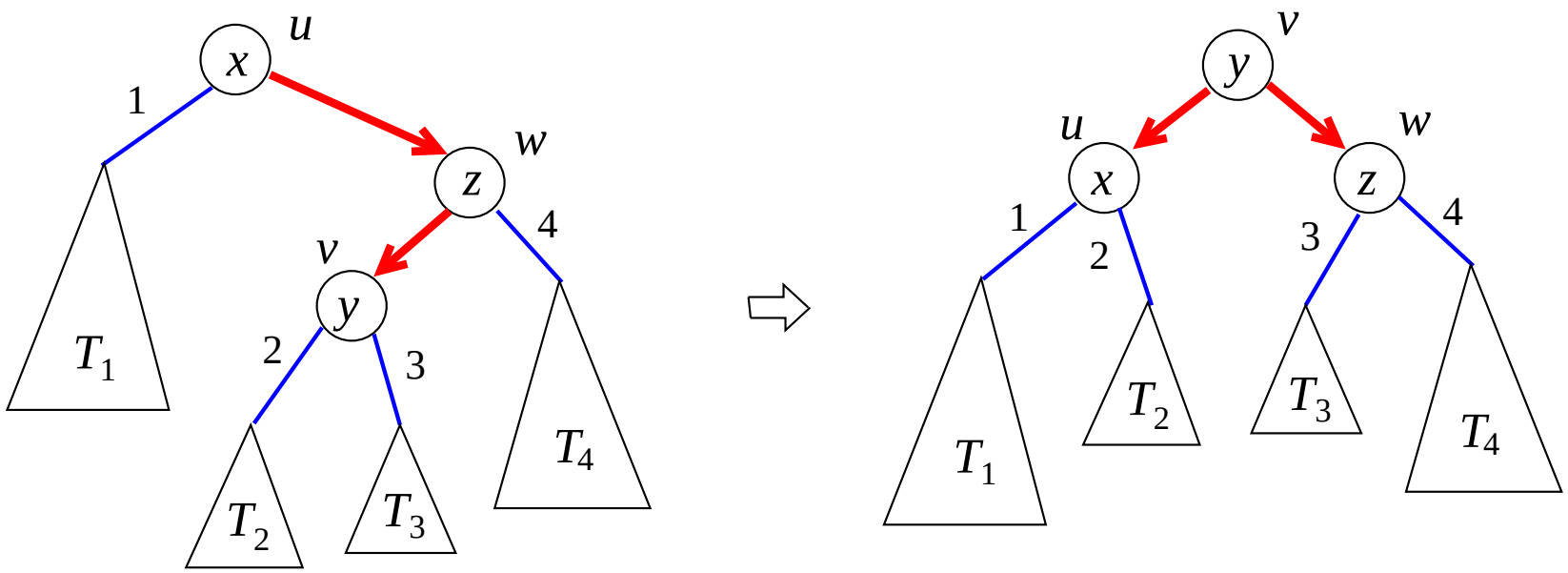
Gesamteffekt: Der **unterste** Knoten v des Zick-Zack-Wegs („links-rechts“) wandert nach oben, wird **Wurzel**; die beiden anderen Knoten u und w werden linkes und rechtes Kind.

LR-Doppelrotation als Programm

Funktion rotateLR(w : AVL_Tree) : AVL_Tree

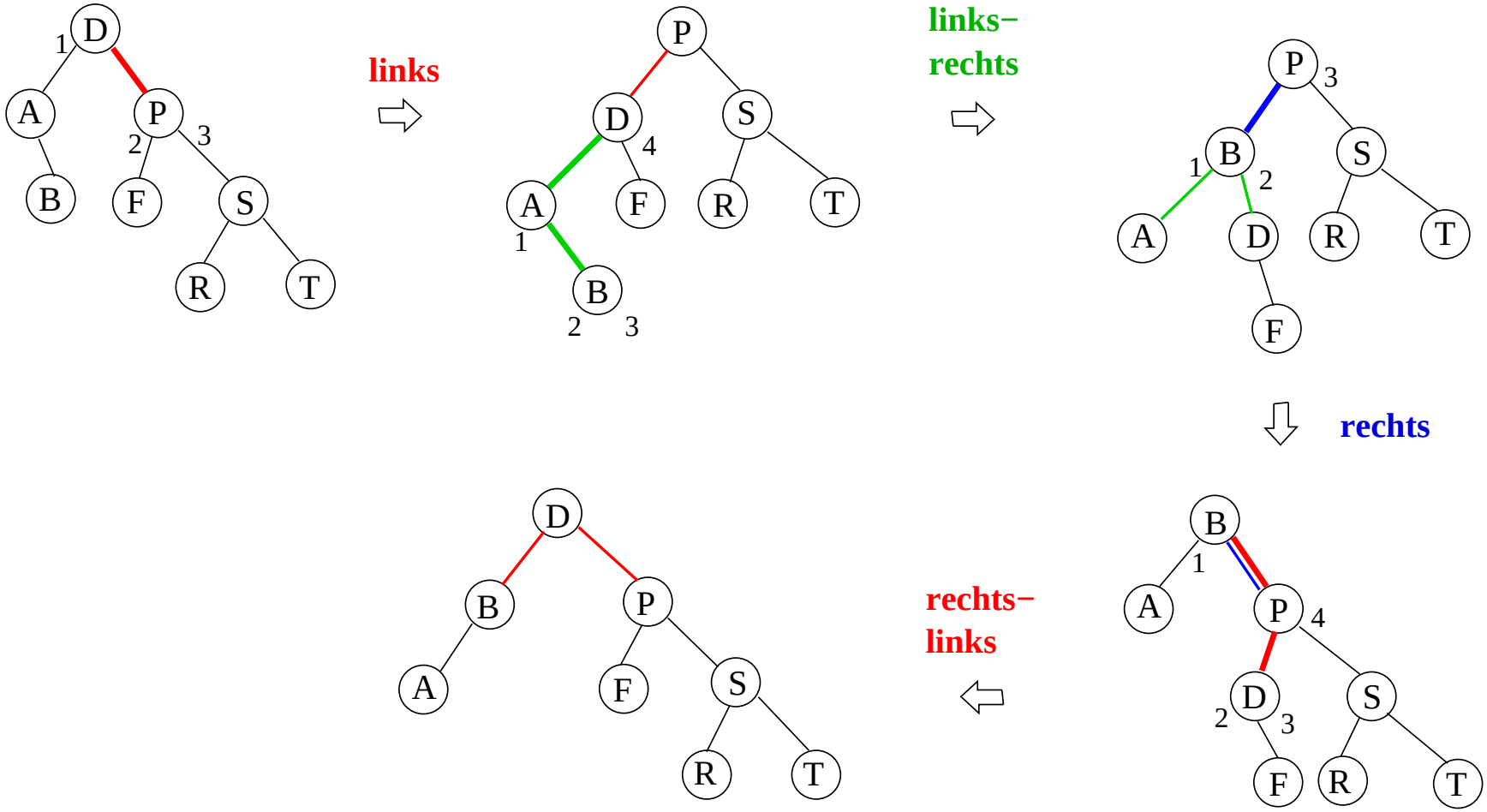
```
(1)  // Eingabe: w ≠ □ mit
(2)        // w.left ≠ □,  w.left.right ≠ □
(3)  u, v : AVL_Tree ;
(4)  u ← w.left ;
(5)  v ← u.right ;
(6)  w.left ← v.right ;
(7)  u.right ← v.left ;
(8)  v.left ← u ;
(9)  v.right ← w ;
(10) return v
(11)  // !! Balancefaktoren in u, v, w sind falsch!
⋮     ⋮
```

Rechts-Links-Doppelrotation:
Symmetrisch zu Links-Rechts-Doppelrotation



Implementierung: Analog zu Links-Rechts-Doppelrotation.

Beispiel:


D
1
A
P
2
3
B
F
S
R
T
links
P
D
S
4
A
F
R
T
1
B
2
3
links–
rechts
P
3
B
S
1
2
A
D
R
T
F
rechts
D
B
P
A
F
S
R
T
rechts–
links
B
1
A
P
4
D
S
2
3
F
R
T

AVL_insert(T, x, r)

Gesamteffekt:

Füge ein wie bei gewöhnlichem BSB: erzeugt neuen Knoten.

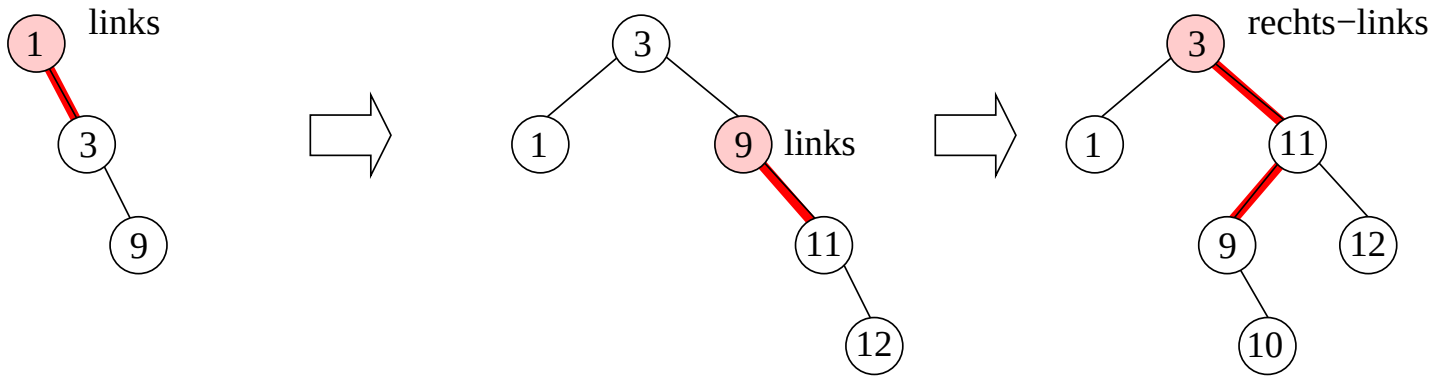
Laufe dann **Einfügeweg von unten nach oben** ab, kontrolliere Balancebedingung.

Wenn Balancebedingung nirgendwo verletzt: fertig.

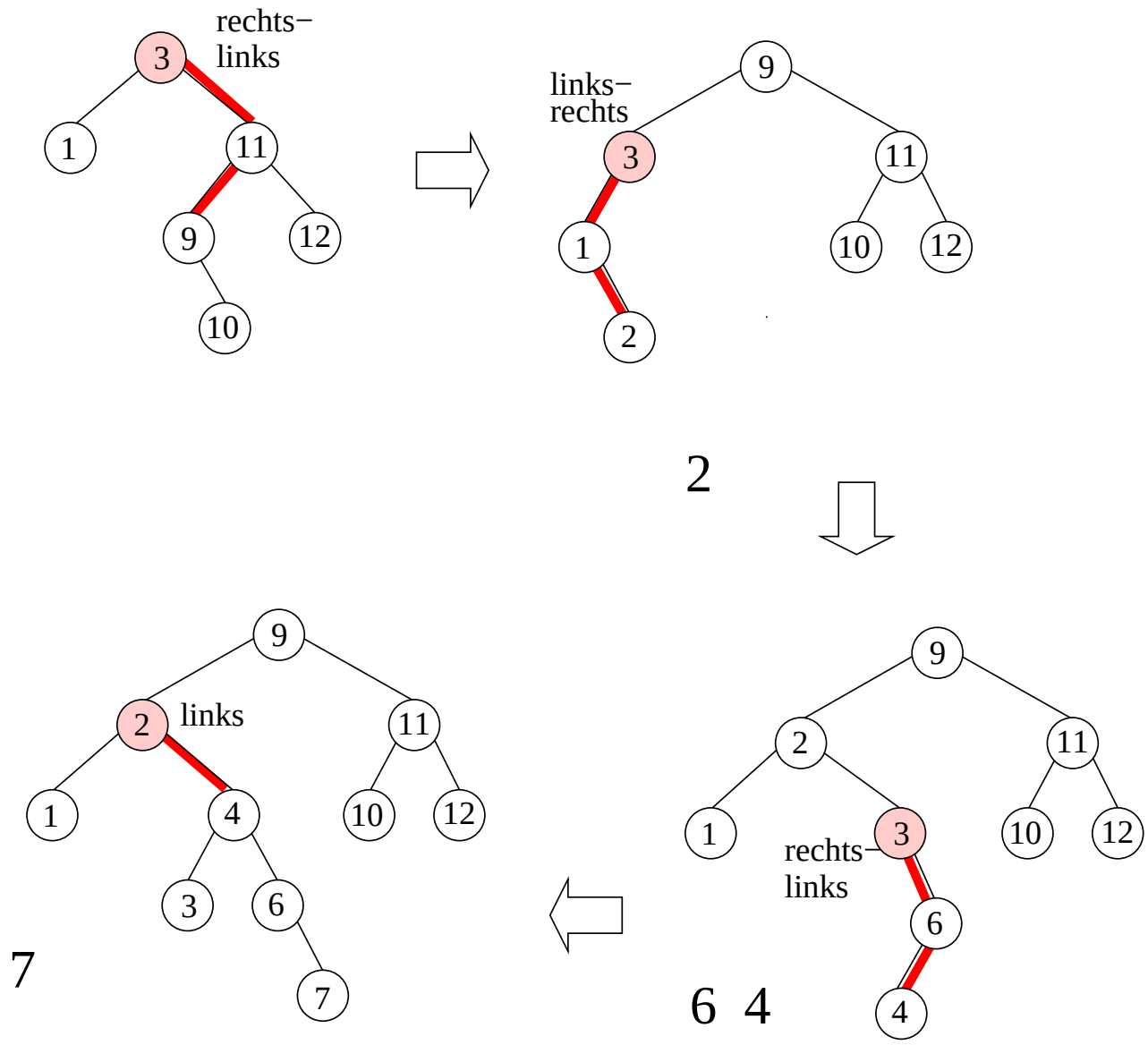
Sonst finde auf dem Weg vom neuen Knoten zur Wurzel den **tiefsten Knoten v**, an dem die Balancebedingung nicht erfüllt ist.

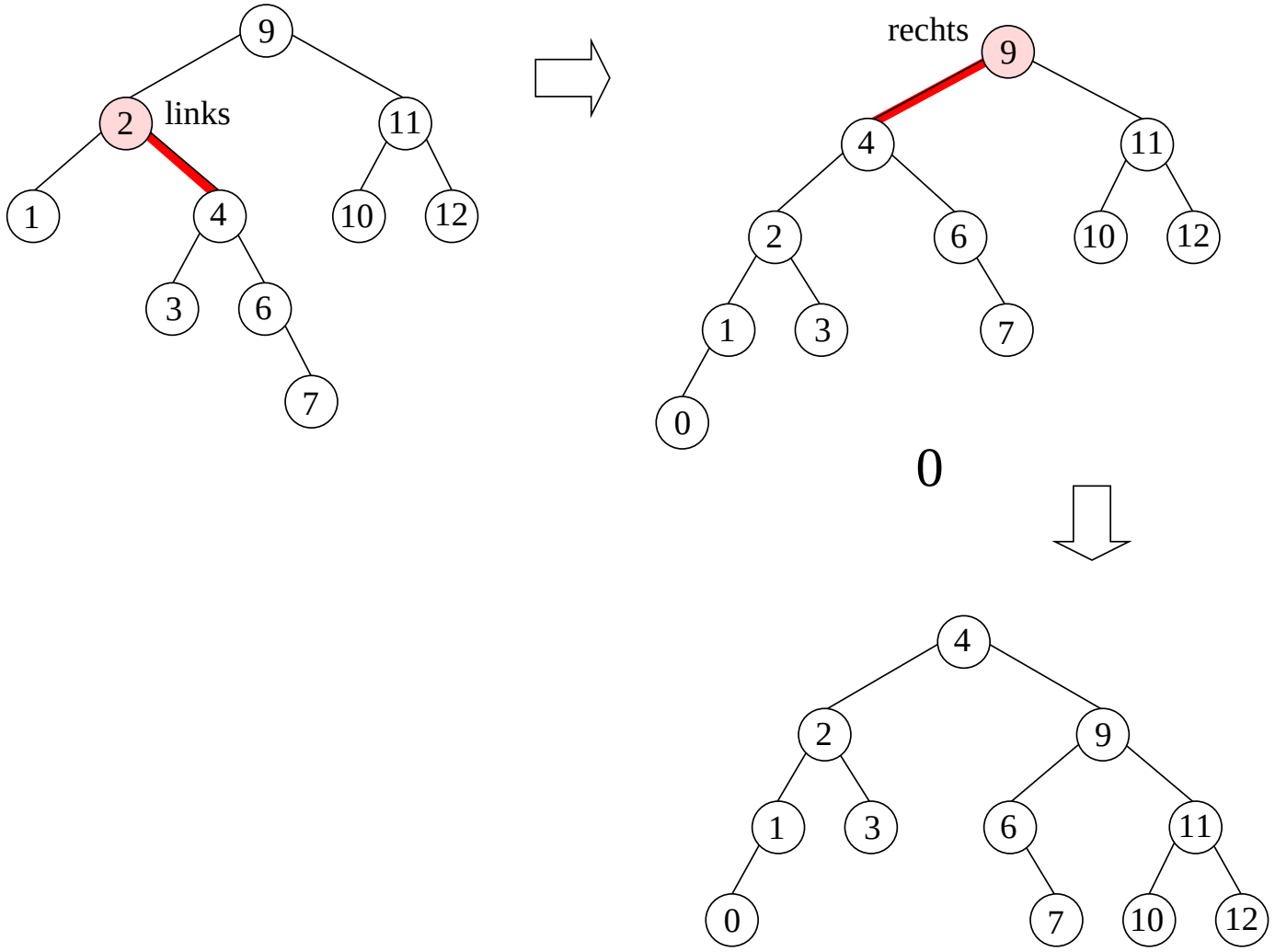
An v wird eine Einfach- oder Doppelrotation ausgeführt und dadurch die Balancebedingung wieder hergestellt.

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.



1 3 9 11 12 9 10


rechts−
links
3
1
11
9
12
10
links−
rechts
9
3
11
1
10
12
2
2
9
2
links
11
1
4
10
12
3
6
7
7
9
2
11
1
3
10
12
rechts−
links
6
4
6 4


9
2
links
11
1
4
10
12
3
6
7
rechts
9
4
11
2
6
10
12
1
3
7
0
0
4
2
9
1
3
6
11
0
7
10
12

Etwas knifflig: Rekursive Programmierung des Ablaufs.

Kann/will nicht verwenden: „Globale Sicht“, Vergleichen von Tiefen durch „Hinschauen“.

Benutzt werden:

- Balancefaktoren `T.bal` in den Knoten,

- Flagbit „`deeper`“, das als Resultat eines rekursiven Aufrufs mitteilt, ob der bearbeitete Unterbaum **tiefer** geworden ist.

Steuerung einer großen Fallunterscheidung durch alte Balancefaktoren und „deeper“-Meldungen aus rekursivem Aufruf.

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 $\leftarrow$ *NULL*;

(8) T.right $\leftarrow$ *NULL*;

(9) T.bal $\leftarrow 0$;

(10) **return** (T, *true*) .

(11) // Baumhöhe hat sich von -1 auf 0 erhöht.

(12) **2. Fall:** `T` $\neq \square$ **and** `T.key = x`.
(13) // Update-Situation!
(14) `T.data` $\leftarrow$ `r` ;
(15) **return** (`T`, *false*) .
(16) // Baumstruktur nicht verändert.

(17) **3. Fall:** `T` $\neq \square$ **and** `x < T.key`.
(18) (`T.left`, `left_deeper`) $\leftarrow$ **AVL_insert**(`T.left`, `x`, `r`) ;
(19) // Rekursives Einfügen in linken Unterbaum
(20) (`T`, `deeper`) $\leftarrow$ **RebalanceInsLeft**(`T`, `left_deeper`) ;
(21) // Rebalancierung in der Wurzel von `T`, **s. unten**
(22) **return** (`T`, `deeper`) .

(23) **4. Fall:** $\texttt{T} \neq \square$ **and** $\texttt{T.key} < \texttt{x}$.

(24) $(\texttt{T.right}, \texttt{right\_deeper}) \leftarrow$ **AVL_insert**$(\texttt{T.right}, \texttt{x}, \texttt{r})$;

(25) // Rekursives Einfügen in rechten Unterbaum

(26) $(\texttt{T}, \texttt{deeper}) \leftarrow$ **RebalanceInsRight**$(\texttt{T}, \texttt{right\_deeper})$;

(27) // Rebalancierung in der Wurzel von T,
symmetrisch zum **3. Fall**.

(28) **return** $(\texttt{T}, \texttt{deeper})$.

Zeile (18): **AVL_insert** rekursiv auf `T.left`, `x`, `r` anwenden.

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 Ebene tiefer geworden.

Invariante (*Beweis* durch Induktion über rekursive Aufrufe):

`T.left` ist AVL-Baum (mit korrekten Balancefaktoren).

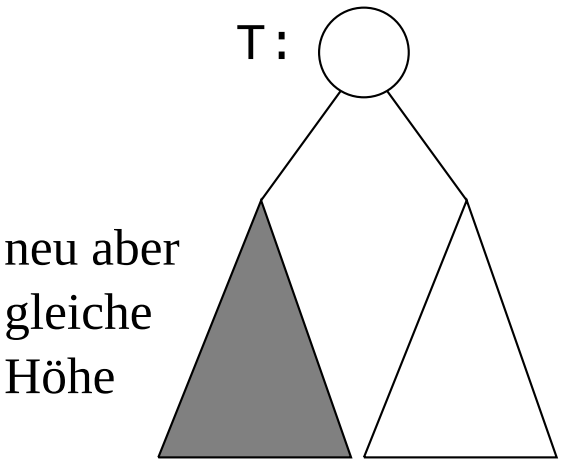
Probleme: **?** Balancebedingung in Wurzel von `T` erfüllt?
? Balancefaktor in Wurzel von `T` korrekt?

Zeile (20): **RebalanceInsLeft** prüft und korrigiert;
Ergebnis: Neuer Baum `T` und Flagbit `deeper`.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-1

`left_deeper` = *false*



Aktion:

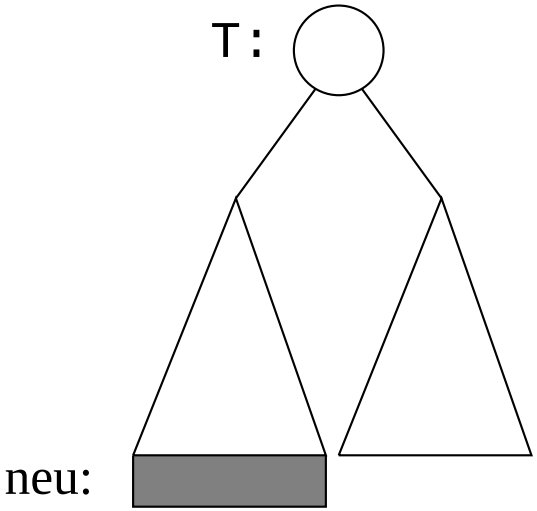
`deeper` $\leftarrow$ *false* ;
// `T.bal` stimmt noch

// Wenn dieser Fall einmal eingetreten ist,
// setzt er sich nach oben immer weiter fort.
// D. h.: Rebalancierung abgeschlossen.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-2

left_deeper $=$ *true* $\wedge$
T.bal $=$ 0 // alter Wert!



Aktion:

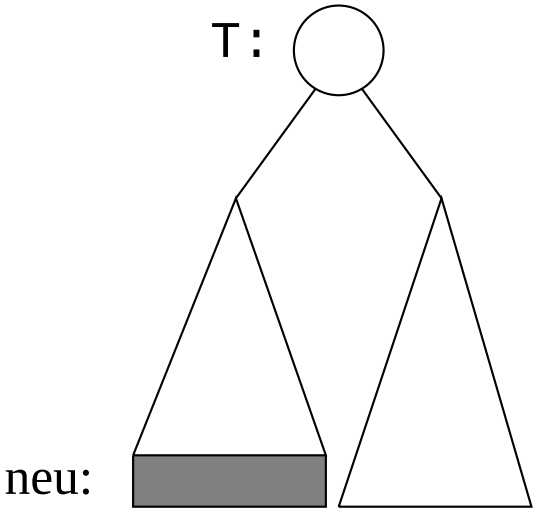
deeper $\leftarrow$ *true* ;
T.bal $\leftarrow -1$;

// Die Rebalancierungsaufgabe wird
// nach oben durchgereicht.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-3

left_deeper $=$ *true* $\wedge$
T.bal $=$ 1 // alter Wert!



Aktion:

deeper $\leftarrow$ *false* ;
T.bal $\leftarrow 0$;

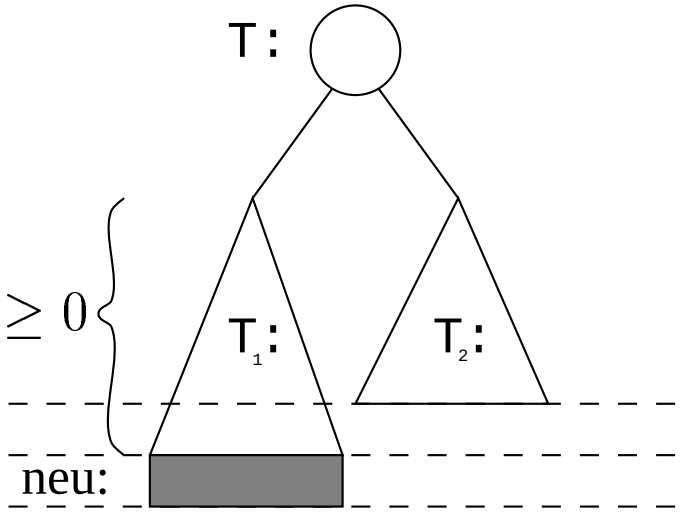
// Rebalancierung ist abgeschlossen,
// ohne Strukturänderung.

RebalanceInsLeft(T,left_deeper)

Fall RebIL-4

Aktion:

left_deeper $= \mathit{true} \wedge$
T.bal $= -1$ // alter Wert!



???

// Strukturänderung nötig.

Beobachte: Schon **vor** dem rekursiven Aufruf von **AVL_insert** war linker Unterbaum T_1 von T nicht leer.

Schon gesehen: **RebIL-1**, **RebIL-3** (und **RebIR-1**, **RebIR-3**) liefern `deeper` $=$ *false*.

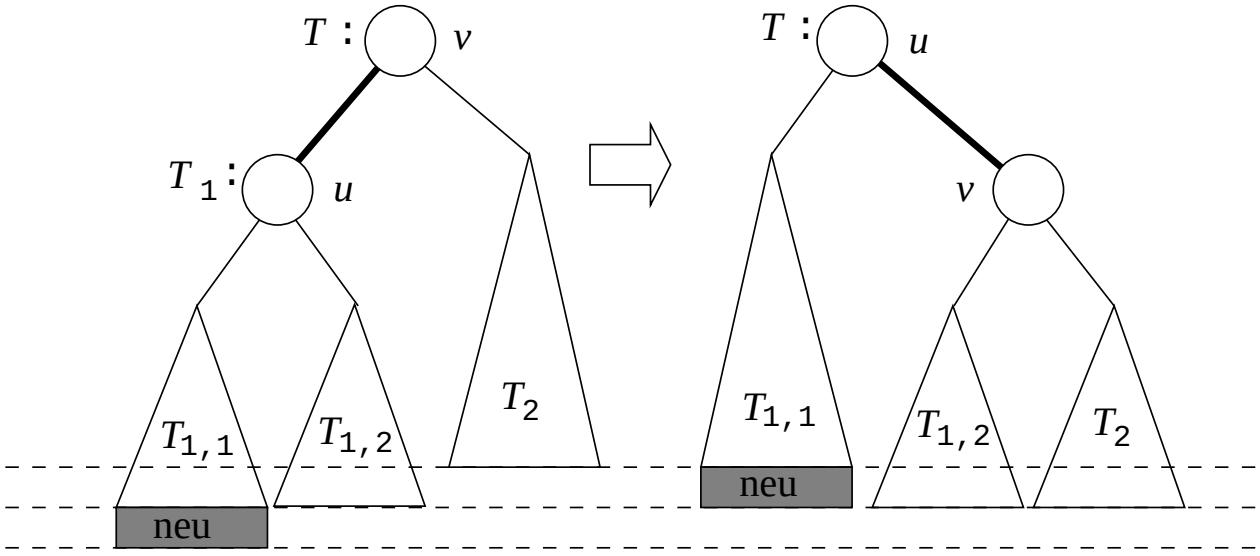
Werden sehen: **RebIL-4**/**RebIR-4** liefern `deeper` $=$ *false*.

$\Rightarrow$ beim rekursiven Aufruf der Rebalancierung für `T.left` ist Fall **RebIL-2** oder **RebIR-2** eingetreten

$\Rightarrow$ Balancefaktor `T.left.bal` in T_1 ist -1 oder 1.

Unterfall RebIL-4.1
$\texttt{T.left.bal} = -1$

Aktion: Rechtsrotation



$\texttt{T} \leftarrow \textbf{rotateR}(\texttt{T})$;

$\texttt{deeper} \leftarrow \mathit{false}$;

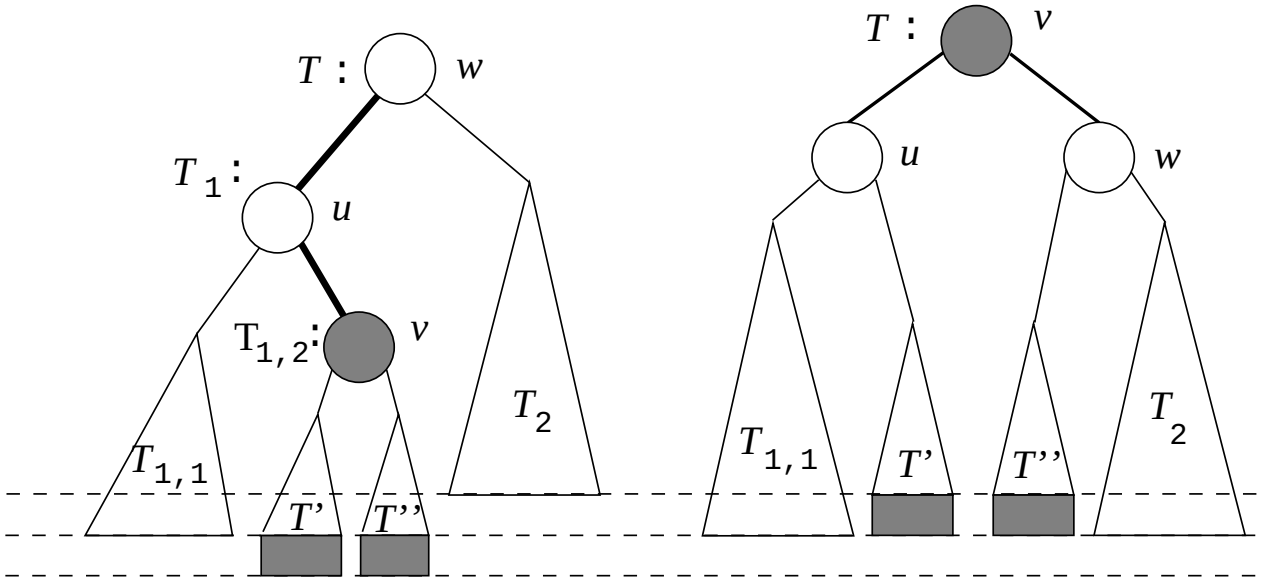
$\texttt{T.bal} \leftarrow 0$;

$\texttt{T.right.bal} \leftarrow 0$;

Unterfall RebIL-4.2

$\texttt{T.left.bal} = 1$

Aktion: Links-Rechts-Doppelrotation



$\texttt{T} \leftarrow$ **rotateLR**$(\texttt{T})$;

$\texttt{deeper} \leftarrow$ *false* ;

Neue `bal`-Werte: s. Tabelle.

Neue `bal`-Werte im Fall RebIL-4.2:

altes	neues		
`T.bal`	`T.left.bal`	`T.right.bal`	`T.bal`
-1	0	1	0
0	0	0	0
1	-1	0	0

Die Zahl (altes) `T.bal` stand **vor der LR-Rotation** in `T.left.right.bal`, gab 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 hat?
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.

Faustregel für **RebalanceInsLeft** und **RebalanceInsRight**:

Laufe von der Einfügestelle nach oben.

Wenn ein **äußerer** Teilbaum (links-links oder rechts-rechts) zu tief ist:

Eine **einfache Rotation**
hebt diesen Teilbaum ein Level höher.

Wenn ein **mittlerer** Teilbaum (links-rechts oder rechts-links) zu tief ist:

Eine **Doppelrotation**
hebt den mittleren Teilbaum ein Level höher.

Nach der ersten Rotation ist die Rebalancierung **beendet**.

Zum Ausprobieren:
`https://www.cs.usfca.edu/~galles/visualization/AVLtree.html`

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.

4.3.5 Folgerung (aus Algorithmus **AVL_insert**):

Für jedes $n \geq 0$ gibt es einen höhenbalancierten Baum mit n Knoten.

Beweis: Man fügt $1, \ldots, n$ mittels **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 $1, \ldots, n$, die genau diesen Baum erzeugt – sogar ohne jede Rotation **(Übung)**.

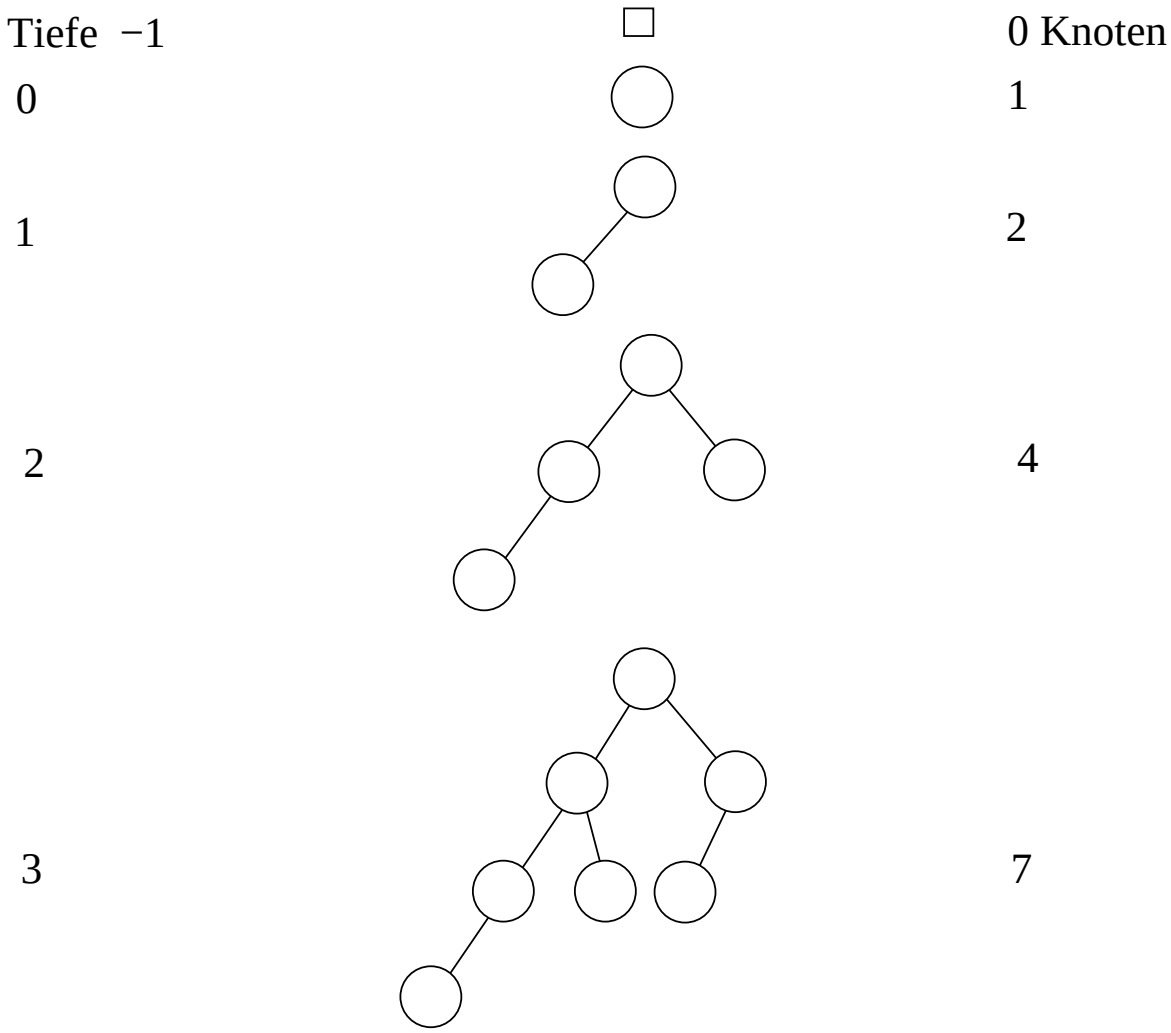
AVL_delete(T,x)

Lösche wie bei gewöhnlichem BSB: Fälle, *ExtractMin*

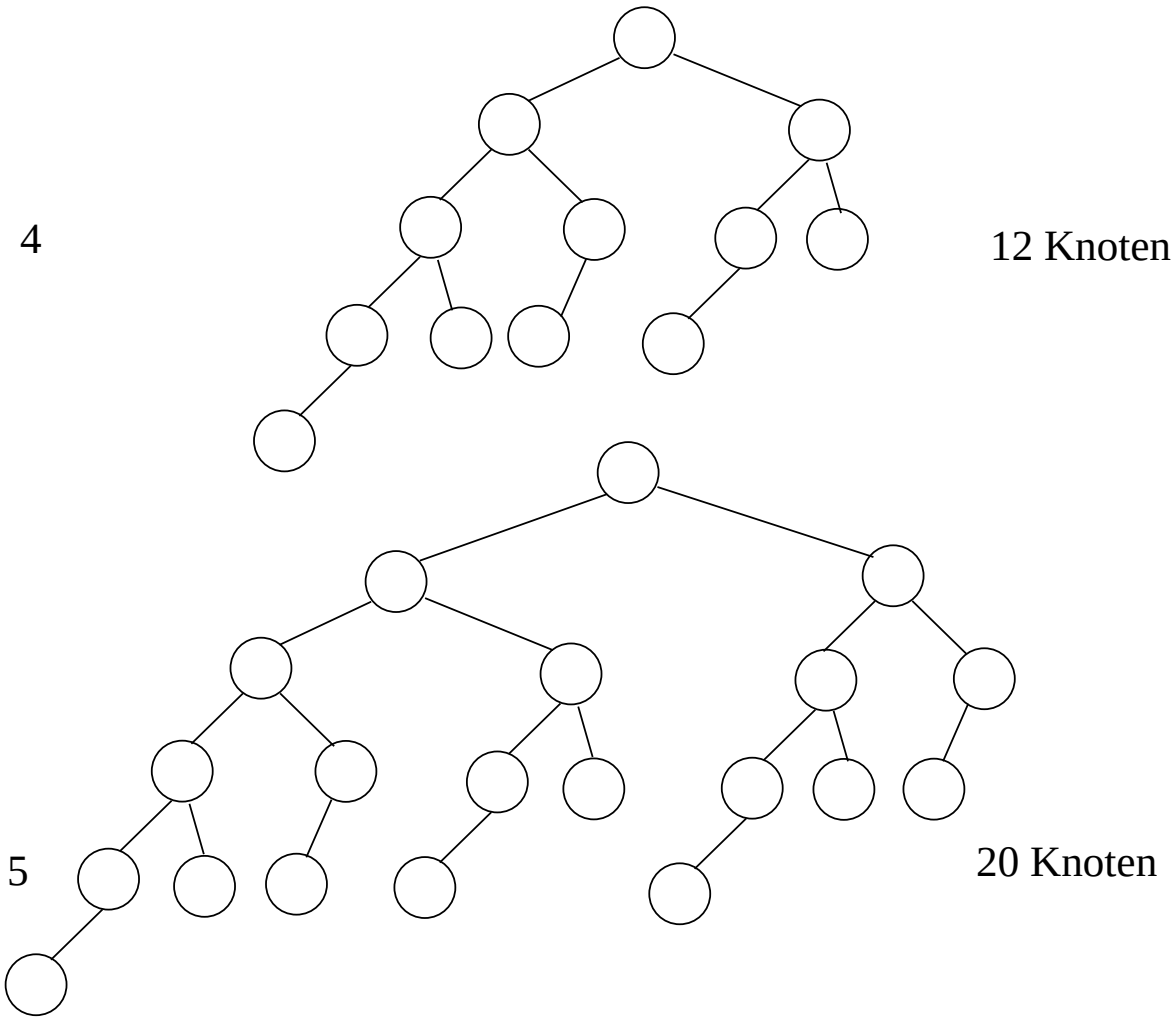
Effekt: Knoten u wird entfernt, dem mindestens ein Unterbaum fehlt. Dadurch rutscht der andere Unterbaum von u eine Ebene höher.

Laufe den Weg von der Stelle, wo u gelöscht wurde, zur Wurzel, teste an jedem Knoten die Balancebedingung, führe gegebenenfalls eine Einfach- oder eine Doppelrotation aus: **RebalanceDelLeft**, **RebalanceDelRight**. Steuerung einer Fallunterscheidung durch `shallower`-Flagbit, das angibt, ob der betroffene Unterbaum „flacher“ geworden ist.

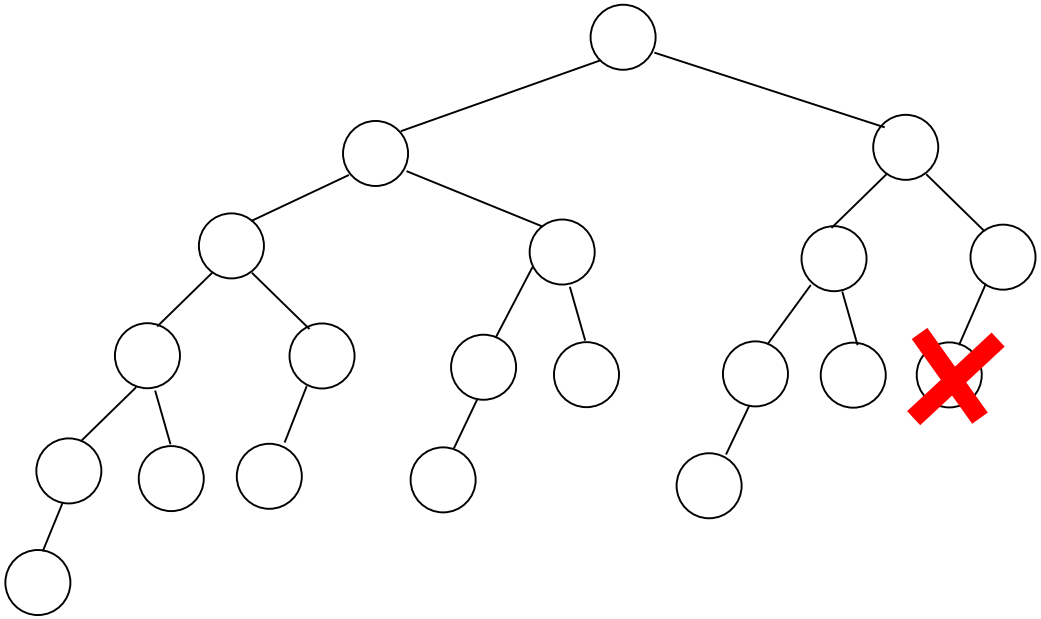
Achtung: Möglicherweise auf mehreren Levels nacheinander Rotation nötig. *Beispiel*: „Fibonacci-Bäume“ . . .

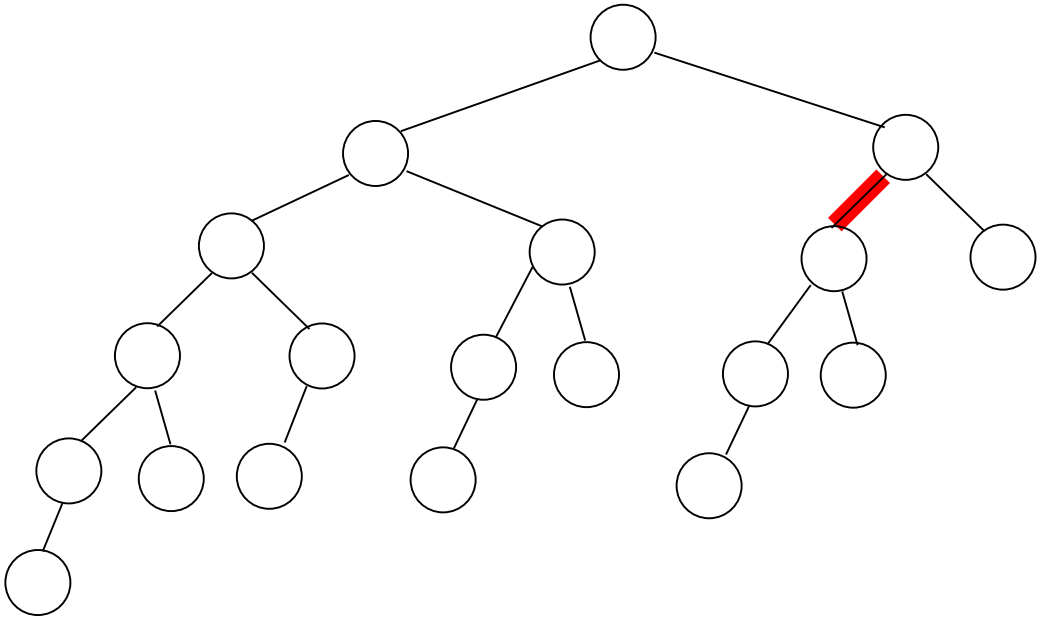


Fibonacci-Bäume der Tiefe $-1, 0, 1, 2, 3$

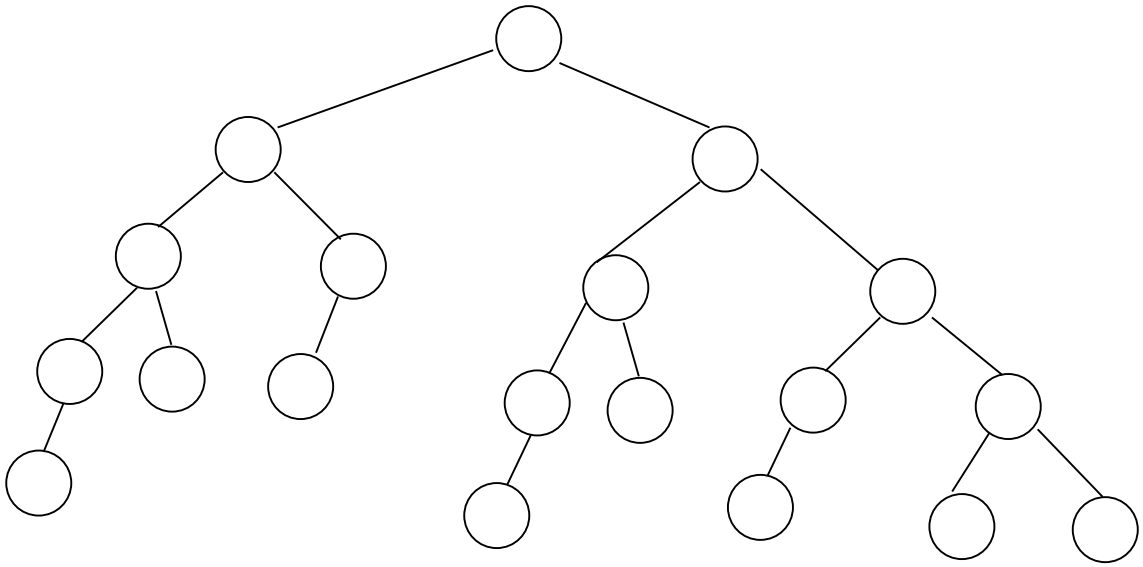


Fibonacci-Bäume der Tiefe $4, 5$





Rechtsrotation!



Fertig!

Programmierung von **AVL_delete**: siehe Druckfolien.
Nicht prüfungsrelevant!

In Übung: Ausführen der Löschung und Rebalancierung, analog zur Einfügung.

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.key < x`.

(13) $(\texttt{T.right}, \texttt{right\_shallower}) \leftarrow$ **AVL_delete**$(\texttt{T.right}, \texttt{x})$;

(14) // Rekursives Löschen im rechten Unterbaum

(15) $(\texttt{T}, \texttt{shallower}) \leftarrow$
RebalanceDelRight`(T,right_shallower)` ;

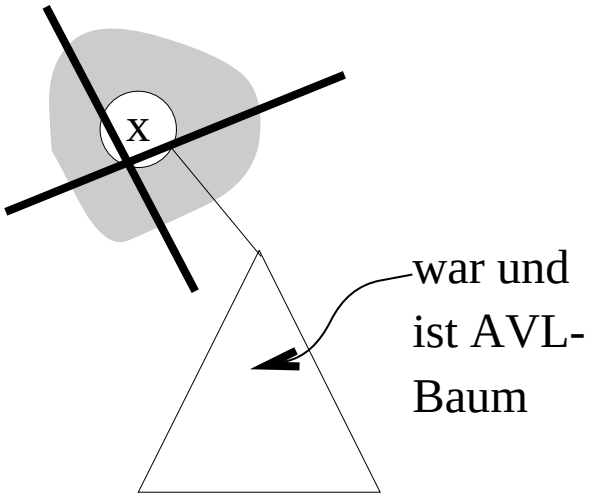
(16) // Rebalancierung in der Wurzel von `T`,
symmetrisch zu 2. Fall

(17) **return** $(\texttt{T}, \texttt{shallower})$.

(18) **4. Fall:** `T` $\neq \square$ **and** `T.key = x`.

(19) // Entferne Wurzelknoten!

(20) **Fall 4a:** `T.left` $= \square$



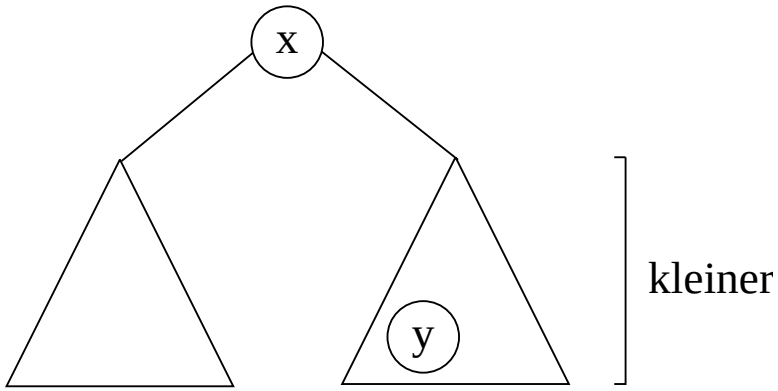
(21) **return** $(\texttt{T.right}, \textit{true})$.

(22) // Baum ist flacher als vorher.

(23) **Fall 4b:** `T.right` $= \square$

(24) **return** $(\texttt{T.left}, \textit{true})$.

(25) **Fall 4c:** $\mathtt{T} \neq \square$ **and** beide Unterbäume nicht leer



(26) $(\mathtt{T.right}, \mathtt{v}, \mathtt{right\_shallower}) \leftarrow$
AVL_extractMin$(\mathtt{T.right})$;

(27) $\mathtt{v.left} \leftarrow \mathtt{T.left}$; $\mathtt{v.right} \leftarrow \mathtt{T.right}$;

(28) $\mathtt{v.bal} \leftarrow \mathtt{T.bal}$;

(29) $\mathtt{T} \leftarrow \mathtt{v}$; // v ersetzt die Wurzel von T

(30) $(\mathtt{T}, \mathtt{shallower}) \leftarrow$
RebalanceDelRight$(\mathtt{T}, \mathtt{right\_shallower})$;

(31) **return** $(\mathtt{T}, \mathtt{shallower})$.

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 ausgeklinkt

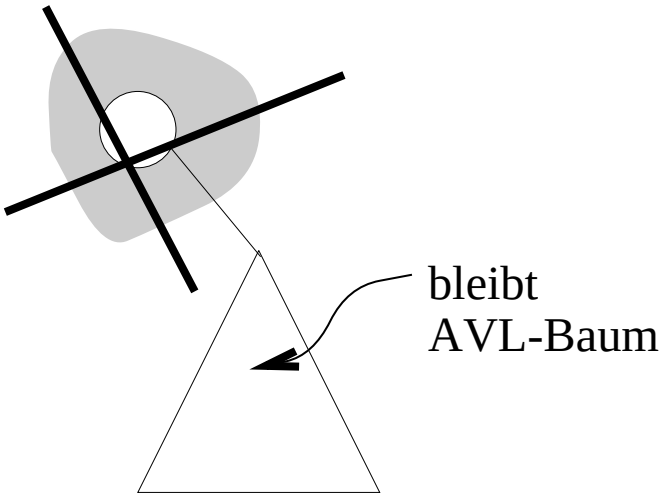
(4) // shallower = *true*, falls T flacher geworden ist

...

In **AVL_extractMin**(T):

Fall „Schluss“:

$\texttt{T.left} = \square$



Wurzel abhängen:

$\texttt{v} \leftarrow \texttt{T}$;

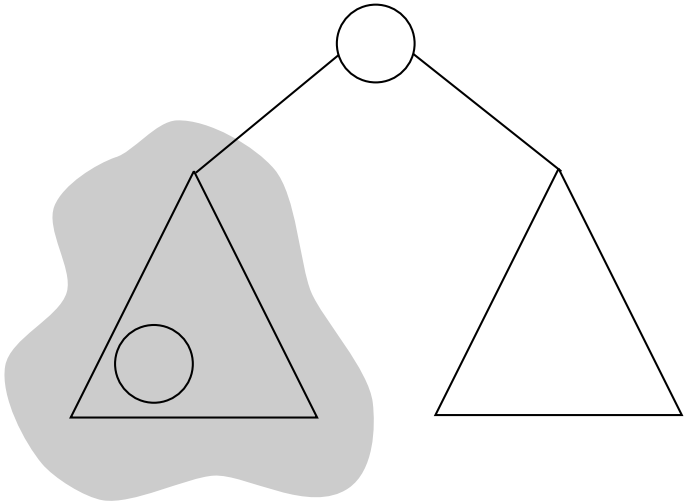
$\texttt{T} \leftarrow \texttt{T.right}$

return $(\texttt{T}, \texttt{v}, \mathit{true})$.

In **Prozedur AVL_extractMin**(T):

Fall „Rekursion“:

T.left $\neq \square$



(T, v, left_shallower) $\leftarrow$ **AVL_extractMin**(T.left);

(T, shallower) $\leftarrow$ **RebalanceDelLeft**(T,left_shallower);

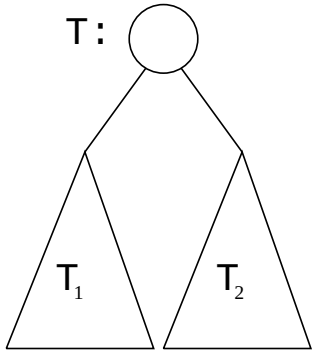
return(T, v, shallower).

RebalanceDelLeft(`T,left_shallower`)

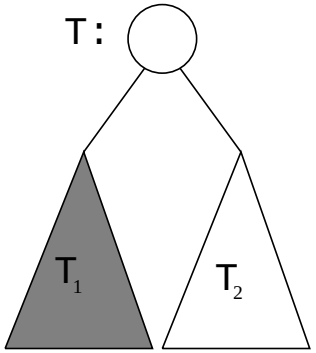
// prüft Balancebedingung in der Wurzel von `T`, korrigiert

// (**RebalanceDelRight**: symmetrisch)

Situation:

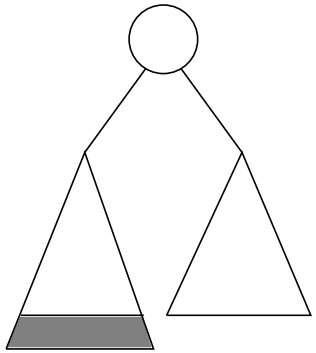


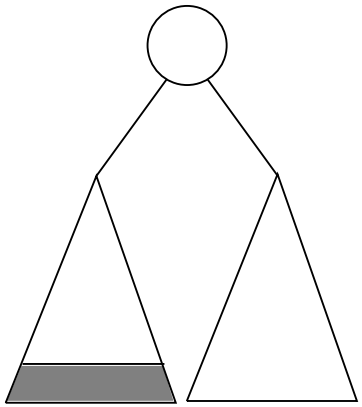
durch rekursives Löschen in T_1 umgebaut zu



$T_1 =$ `T.left` verändert; in „`left_shallower`“ wird die Information geliefert, ob T_1 flacher geworden ist.

`T.bal`: bislang unverändert.

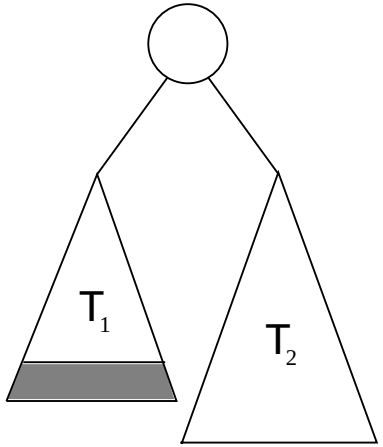
Fall	Aktion
Fall RebDL-1:	
$\texttt{left\_shallower} = \mathit{false}$	$\texttt{shallower} \leftarrow \mathit{false};$
Fall RebDL-2:	
$\texttt{left\_shallower} = \mathit{true} \wedge$ $\texttt{T.bal} = -1$	
	$\texttt{shallower} \leftarrow \mathit{true};$ $\texttt{T.bal} \leftarrow 0;$
geschrumpft	

Fall	Aktion
Fall RebDL-3: $\texttt{left\_shallower} = \mathit{true} \wedge$ $\texttt{T.bal} = 0$	
	$\texttt{shallower} \leftarrow \mathit{false};$ $\texttt{T.bal} \leftarrow 1;$
geschrumpft	

Fall RebDL-4:

$\texttt{left\_shallower} = \textit{true} \wedge$

$\texttt{T.bal} = 1$

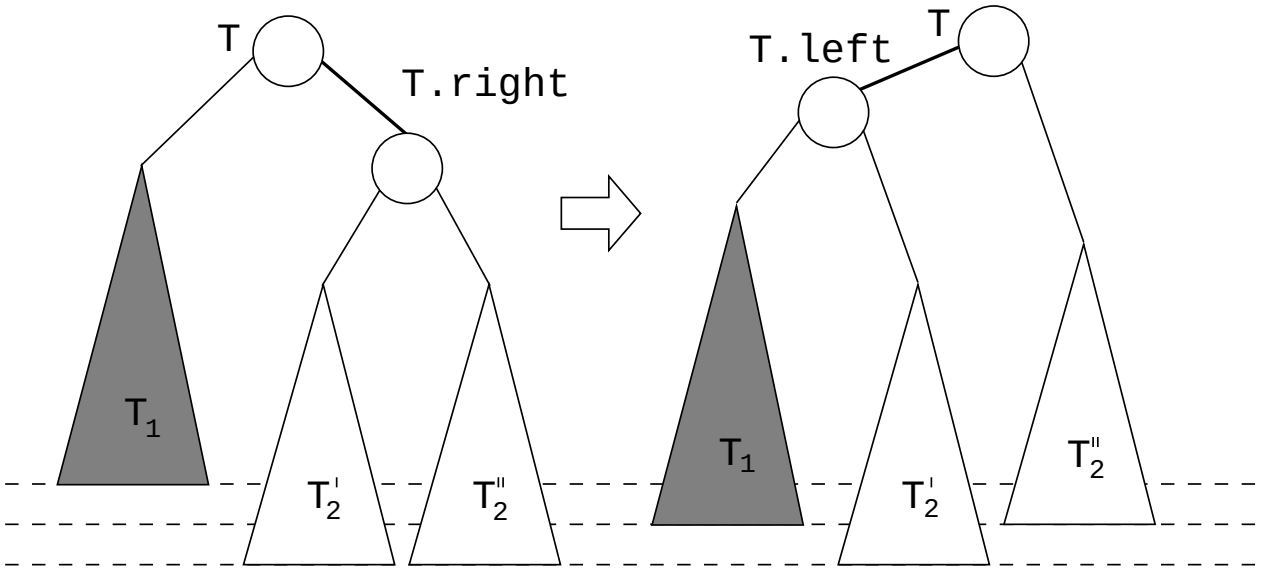


geschrumpft

Unterfälle je nach Aussehen von T_2.

T_2 hat Tiefe mindestens 1, da aus T_1 ein Knoten entfernt werden konnte, also T_1 Tiefe mindestens 0 hatte.

Fall RebDL-4.1: `T.right.bal` $= 0$;



Linksrotation! `T` $\leftarrow$ **rotateL**(`T`);

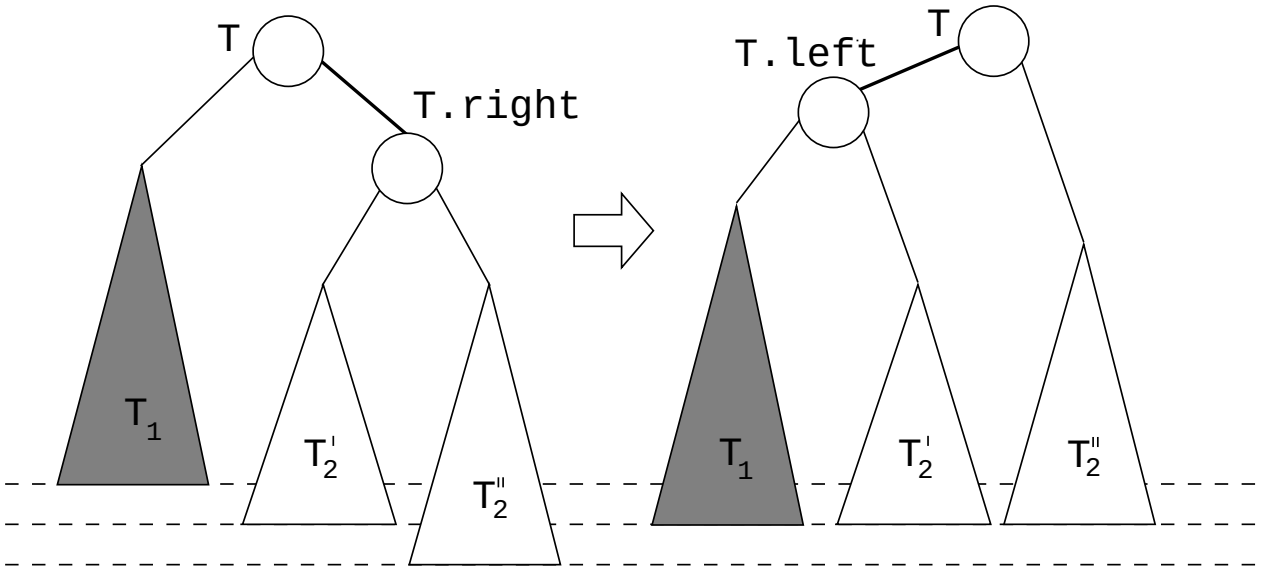
`T.left.bal` $\leftarrow 1$;

`T.bal` $\leftarrow -1$;

`shallower` $\leftarrow$ *false*;

// Rebalancierung beendet

Fall RebDL-4.2: `T.right.bal` $= 1$;



Linksrotation!

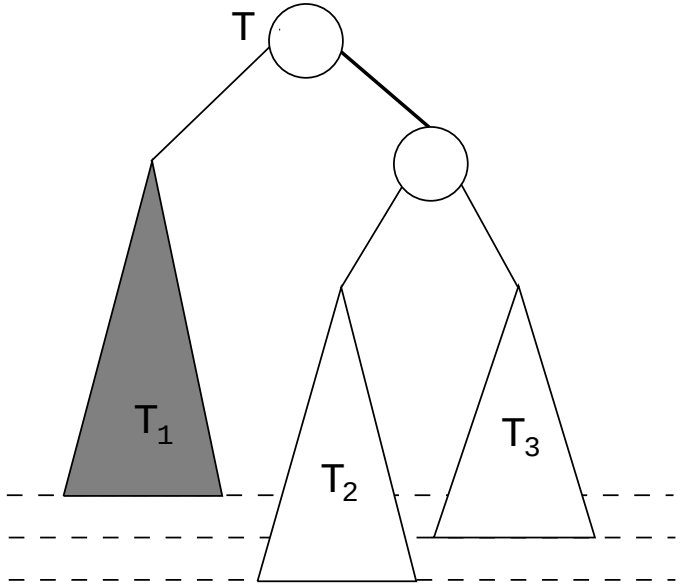
$\texttt{T} \leftarrow$ **rotateL**$(\texttt{T})$;

$\texttt{T.left.bal} \leftarrow 0$;

$\texttt{T.bal} \leftarrow 0$;

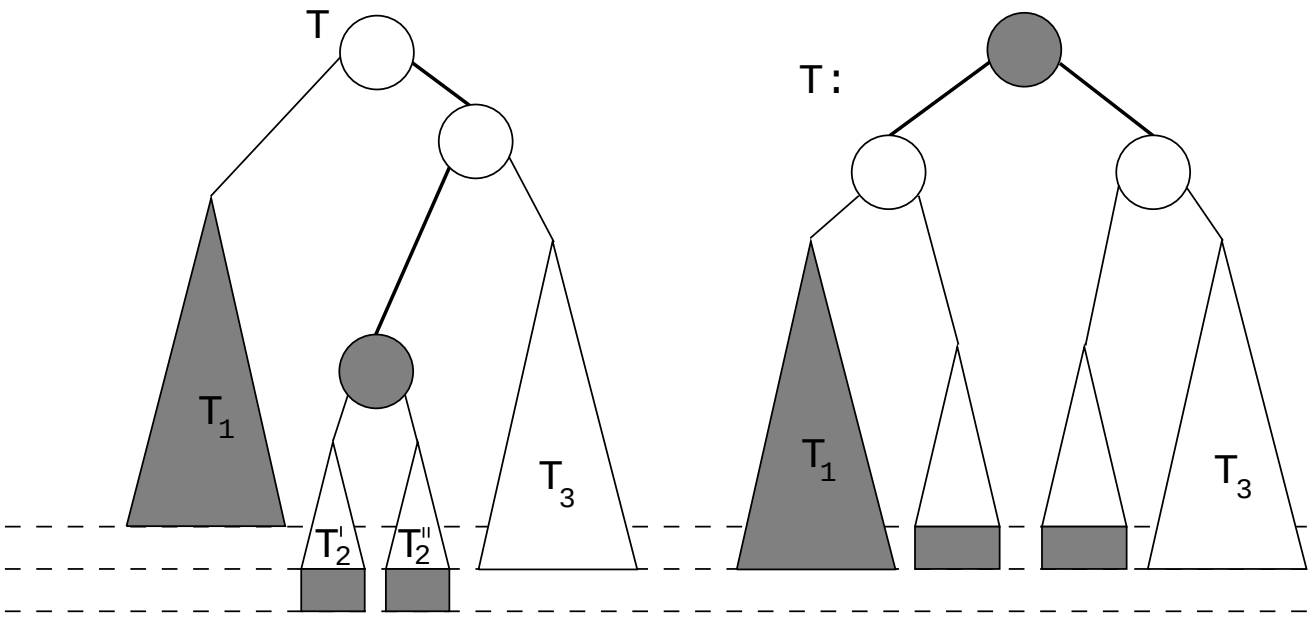
$\texttt{shallower} \leftarrow$ *true*;

Fall RebDL-4.3: `T.right.bal` $= -1;$



Fall RebDL-4.3: (Forts.)

Betrachte Teilbäume von $T_2 = \texttt{T.right.left}$.



Doppelrotation! $\texttt{T} \leftarrow$ **rotateRL**$(\texttt{T})$;

$\texttt{shallower} \leftarrow$ *true*;

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

(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.

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 *delete*(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

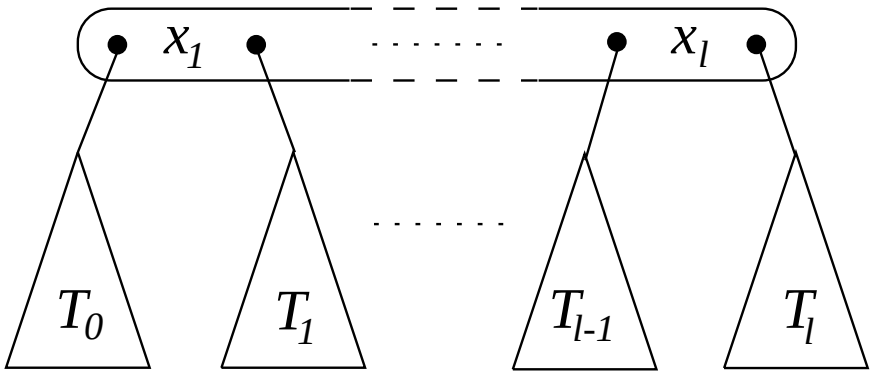
$$\boldsymbol{O(\log n)},$$

wenn n die Anzahl der Wörterbucheinträge ist.

4.4 Mehrweg-Such-Bäume

Bäume aus Knoten mit variablem Ausgrad.

Knoten mit mehreren Schlüsseln und mehr als zwei Unterbäumen:

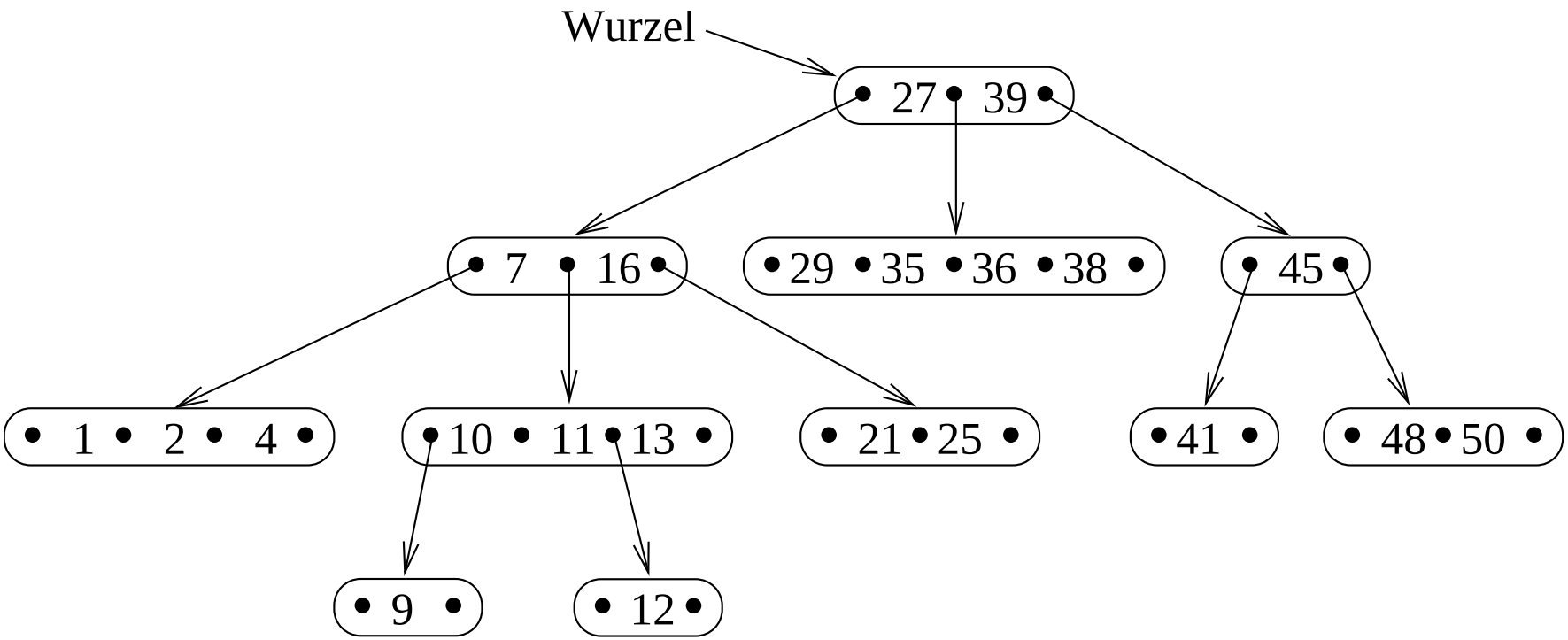


$x_1 < \cdots < x_l$.

Für y_i in T_i, $0 \leq i \leq l$:

$y_0 < x_1 < y_1 < x_2 < \cdots < y_{l-1} < x_l < y_l$

Beispiel eines $\mathbb{N}$-MwSB:



- ohne Pfeil steht für einen Zeiger auf einen leeren Baum $\square$.

Definition 4.4.1

Mehrweg-Suchbäume (MwSBe) über dem (angeordneten) Universum U sind wie folgt **induktiv definiert**:

(0) Der leere Baum ist ein U-MwSB.

(1) Ist $l \geq 1$ und sind $x_1 < x_2 < \cdots < x_l$ Schlüssel in U und sind $T_0, T_1, \ldots, T_l$ U-MwSBe mit:

Für y_i in T_i, $0 \leq i \leq l$:

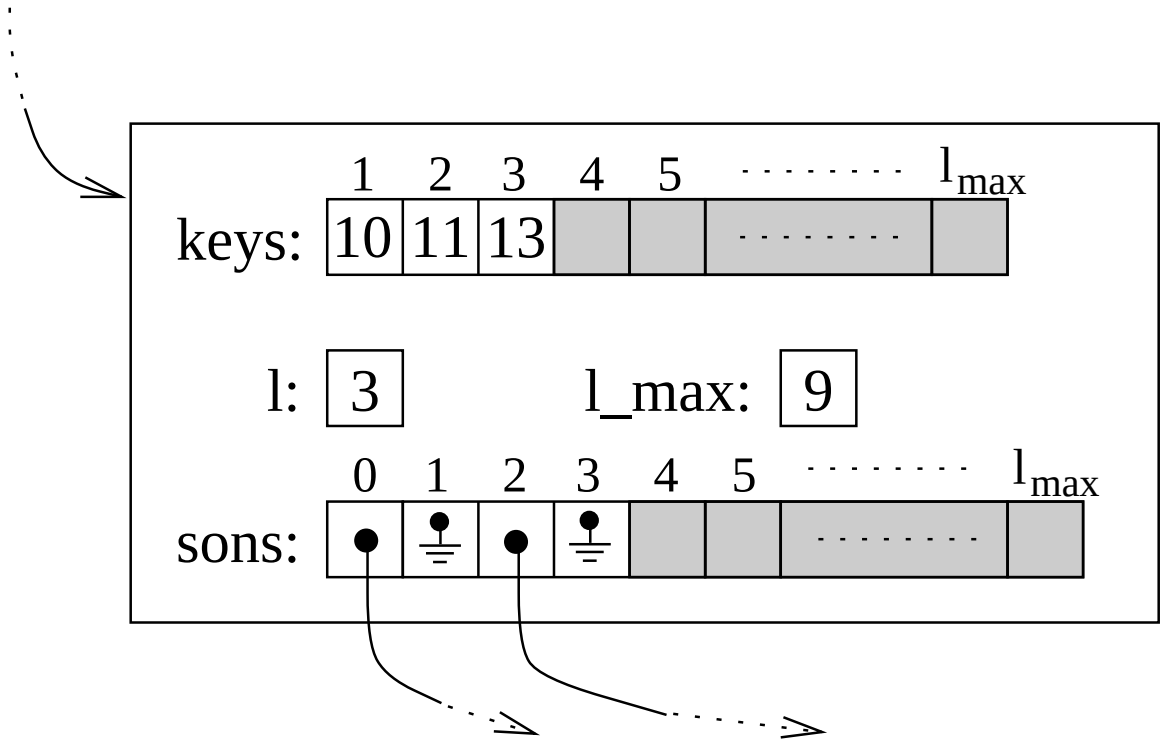
$$y_0 < x_1 < y_1 < x_2 < \cdots < y_{l-1} < x_l < y_l$$

dann ist auch $(T_0, x_1, T_1, x_2, \ldots, x_{l-1}, T_{l-1}, x_l, T_l)$ ein U-Mehrweg-Suchbaum.

Analog: (U, R)-Mehrweg-Suchbäume.

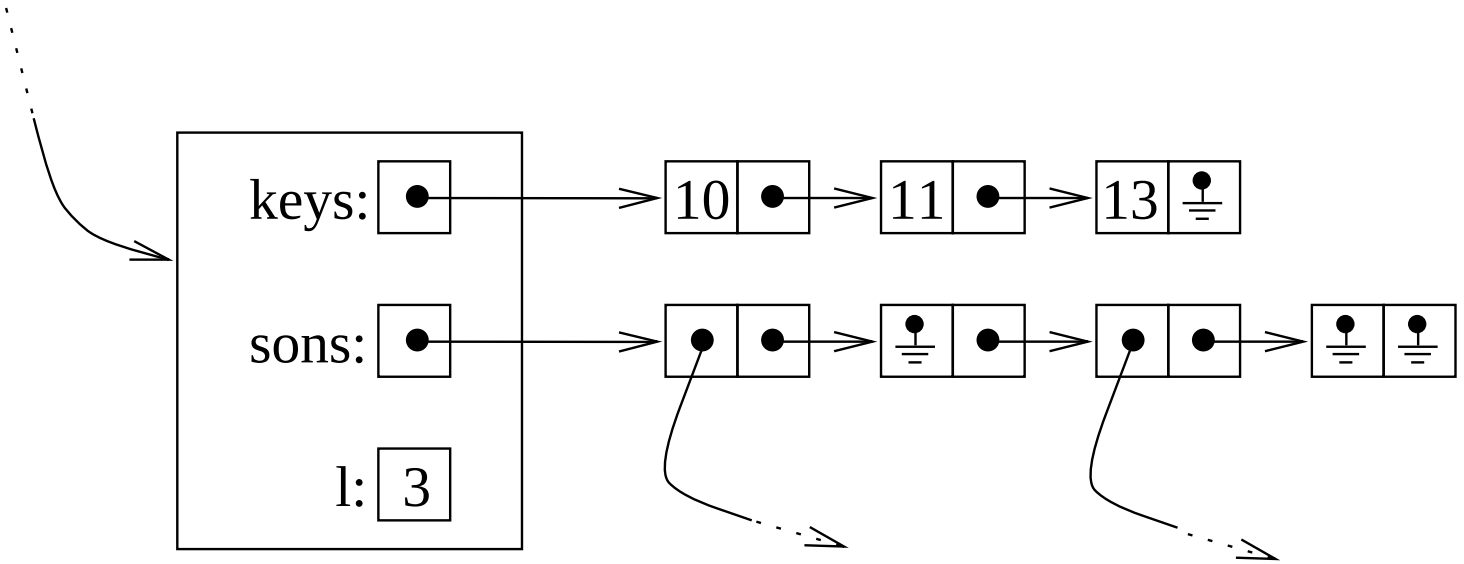
Implementierung eines Knotens in Mehrwegsuchbaum:

Variante 1: Jeder Knoten enthält 2 Arrays.



Implementierung eines Knotens in Mehrwegsuchbaum:

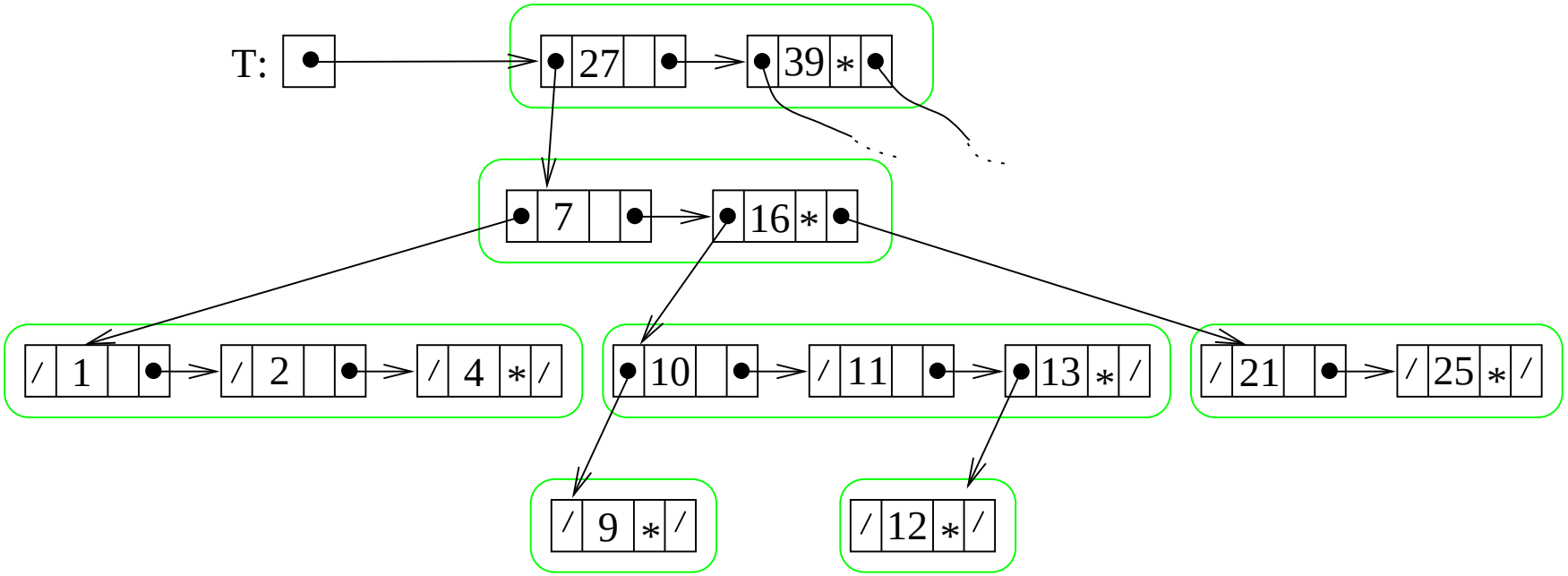
Variante 2: Schlüssel und Kinder als Listen.



Implementierung eines Knotens in 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: „*“.)
Dieser Knoten hat 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:

(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**;

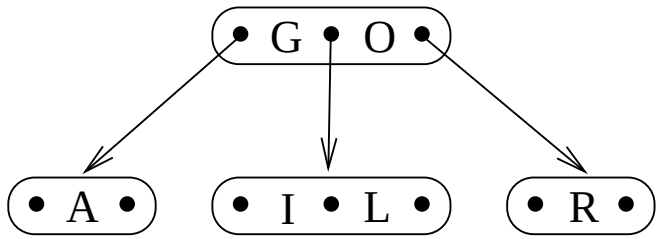
(c) Alle Blätter von T haben dieselbe Tiefe.

Beispiele:

Leerer 2-3-Baum:

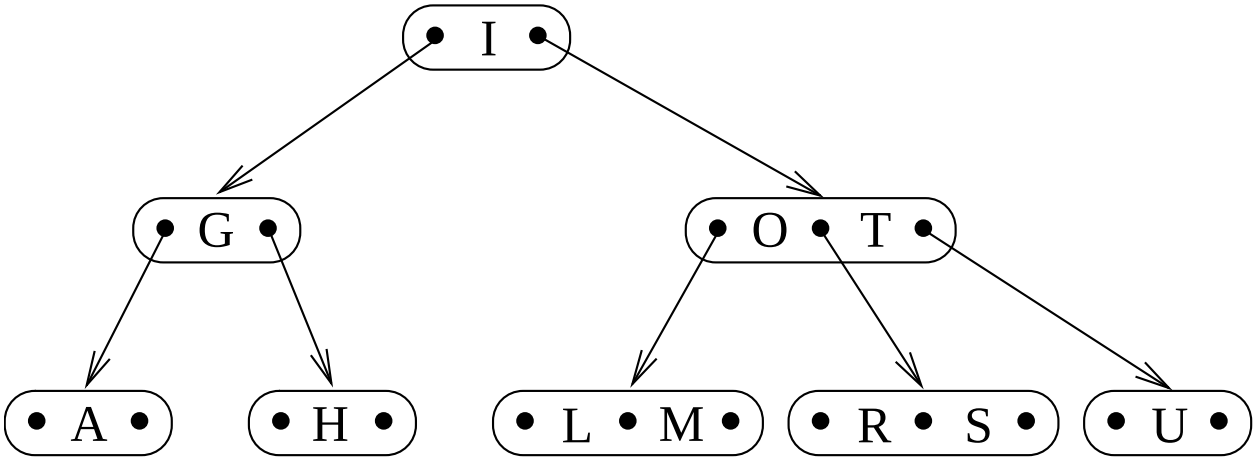
□

2-3-Baum mit 2 Ebenen:



NB: Die Ebenen werden von den Blättern her nummeriert.

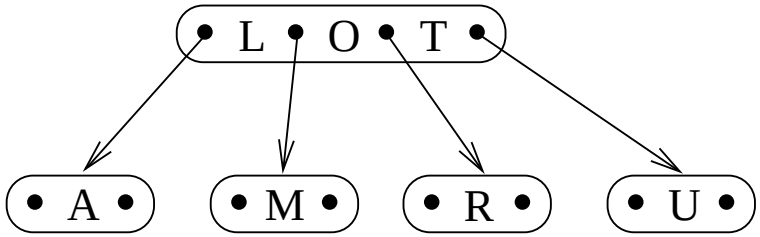
Beispiele:



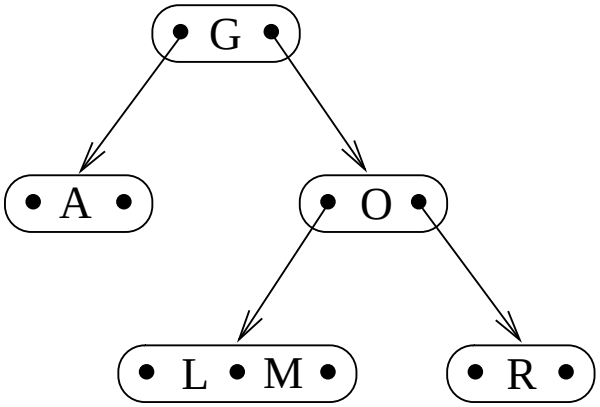
Level 2

Level 1

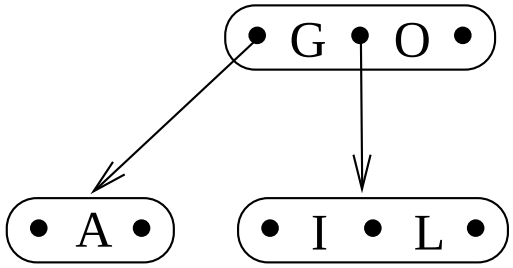
Level 0 = Blattebene



Kein 2-3-Baum (zu großer Grad).



Kein 2-3-Baum (Blätter in verschiedenen Tiefen).

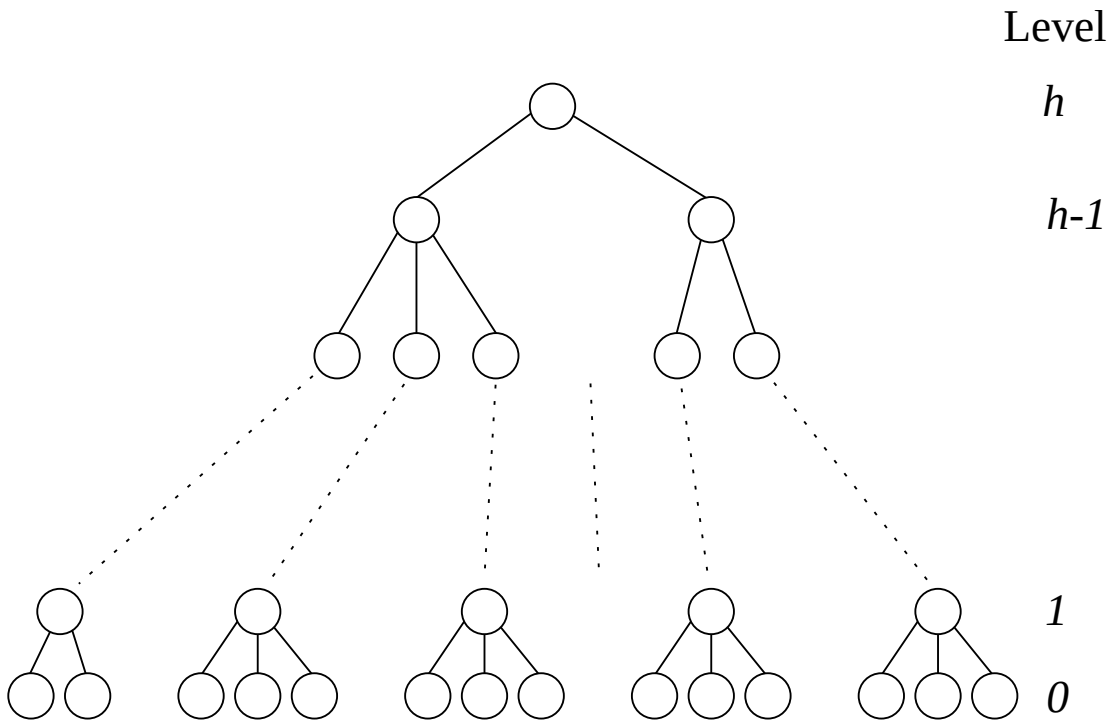


Kein 2-3-Baum (Leerer Unterbaum in Nicht-Blatt).

Proposition 4.4.3

Die Tiefe eines 2-3-Baums mit n Einträgen ist mindestens $\lceil \log_3(n+1) \rceil - 1$ und höchstens $\lfloor \log(n+1) \rfloor - 1$.

$[\log_3 x \approx 0.63 \log_2 x]$ *Beweis*: Übung.



Suche in 2-3-Bäumen

lookup(T, x), für 2-3-Baum T, $x \in U$.

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

2. Fall: Die Wurzel von T enthält Schlüssel x_1
(und eventuell $x_2 > x_1$).

2a: $x < x_1$:
Rufe *lookup* für ersten Unterbaum T_0 auf.

2b: $x = x_1$: „gefunden“.

2c: $x_1 < x \wedge$ (x_2 existiert nicht $\vee$ $x < x_2$):
Rufe *lookup* für zweiten Unterbaum T_1 auf.

2d: x_2 existiert $\wedge$ $x = x_2$: „gefunden“.

2e: x_2 existiert $\wedge$ $x_2 < x$:
Rufe *lookup* für dritten Unterbaum T_2 auf.

Klar:

Zeitaufwand: Wenn x im Knoten v_x sitzt:
Für jeden Knoten v auf dem Weg von der Wurzel zu v_x entstehen Kosten $O(1)$.

$\Rightarrow$

Zeit für Suche in 2-3-Bäumen ist $O(\log n)$.

Mitteilung 4.4.4

Einfügungen und Löschungen lassen sich in 2-3-Bäumen *ähnlich wie in AVL-Bäumen* in Zeit $O(\log n)$ ausführen.
Die nötigen Strukturänderungen werden dabei entlang des Wegs von der geänderten Stelle aus nach oben zur Wurzel gehend durchgeführt.

2-3-4-Bäume

Ein 2-3-4-Baum ist ein **Mehrweg-Suchbaum** mit:

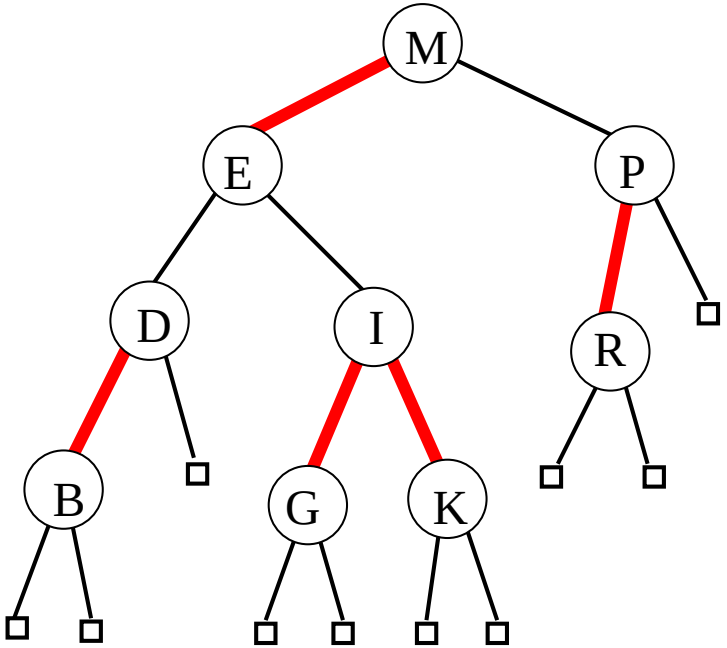
(a) Jeder Knoten hat 2, 3 oder 4 Kinder (1, 2 oder 3 Schlüssel)

(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.

Rot-Schwarz-Bäume



Rot-Schwarz-Bäume

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**.

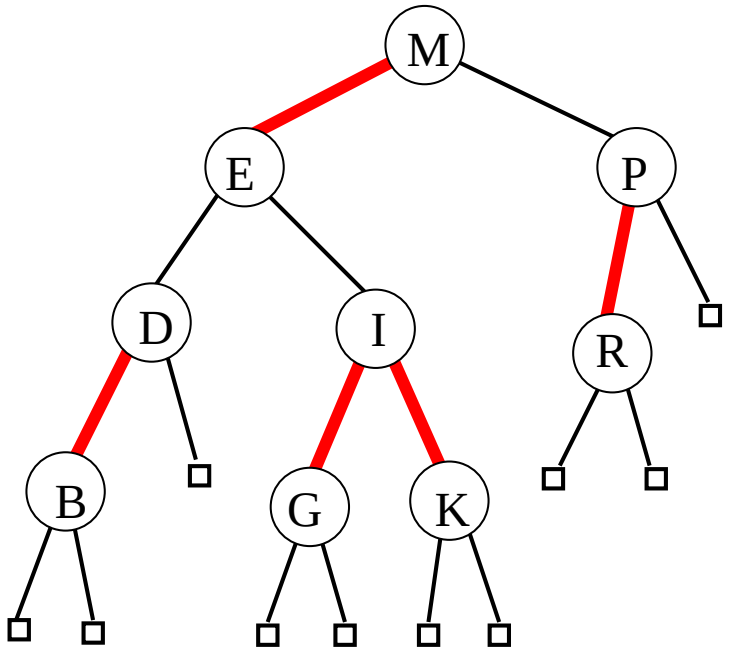
(c) Auf keinem Weg folgen zwei rote Kanten aufeinander.

(d) Wenn ein Knoten nur eine rote Ausgangskante hat, so führt diese zum linken Kind.

(e) 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** $b(T)$ des Rot-Schwarz-Baums T ist die Schwarz-Tiefe des Wurzelknotens von T.



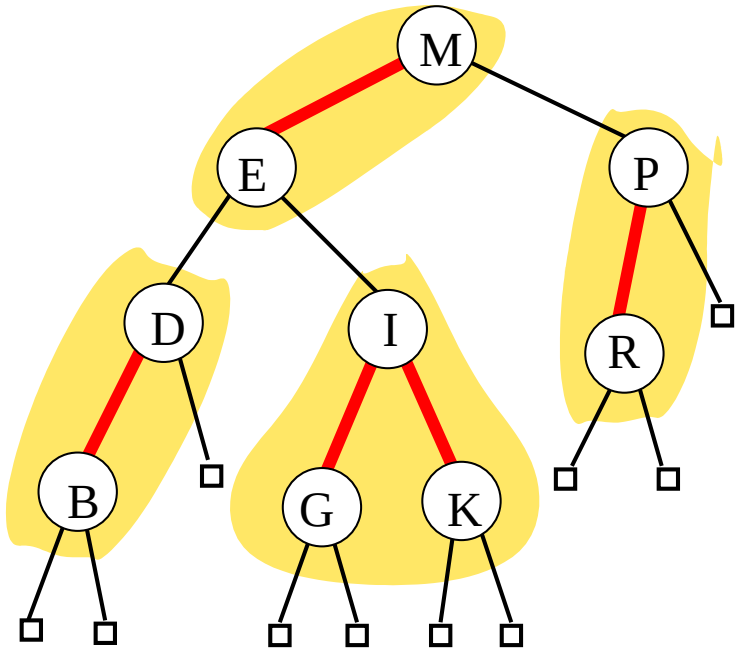
Schwarz-Tiefe: 2.

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].

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 **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 (≥ 1 Schlüssel)

(c) Für jeden Knoten v in T gilt: v hat **einen** leeren Unterbaum
$\Rightarrow$ **alle** Unterbäume unter v sind leer, d. h. 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: $t = 32$. Tiefe: $\leq (\log n)/5$.

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.