LMU LUDWIG-MAXIMILIANS-UNIVERSITY MUNICH
DEPARTMENT INSTITUTE FOR INFORMATICS
DATABASE SYSTEMS GROUP

Algorithmen und Datenstrukturen
Kapitel 4: Suchverfahren

Skript zur Vorlesung

Algorithmen und Datenstrukturen

Sommersemester 2015

Ludwig-Maximilians-Universität München

basierend auf dem Skript von Prof. Dr. Martin Ester, Prof. Dr. Daniel A. Keim, Dr. Michael Schiwietz und Prof. Dr. Thomas Seidl

Suchverfahren

- **Gegeben:** Menge von Objekten, die durch (eindeutige) Schlüssel charakterisiert sind.
- **Aufgabe von Suchverfahren:** Suche bestimmter Objekte anhand ihres Schlüssels.

Bisher: Zwei Methoden der Speicherung solcher Objekte:

- Sequentielle Speicherung
  → sortiertes Array (binäre Suche)
- Verkettete Speicherung
  → lineare Liste

Motivation

Zeitaufwand im schlechtesten Fall für eine Menge von n Elementen:

Operation	Sequenziell gespeichert (sortiertes Array)	Verkettet gespeichert (lineare Liste)
Suche Objekt mit gegebenen Schlüssel	$O(\log n)$	$O(n)$
Einfügen an bekannter Stelle	$O(n)$	$O(1)$
Entfernen an bekannter Stelle	$O(n)$	$O(1)$

Im Folgenden:

Behandlung von Verfahren, die sowohl die Suche als auch das Einfügen und Entfernen „effizient“ unterstützen
→ bessere Laufzeitkomplexität als $O(n)$

Binäre Suchbäume: Motivation

Ziel:

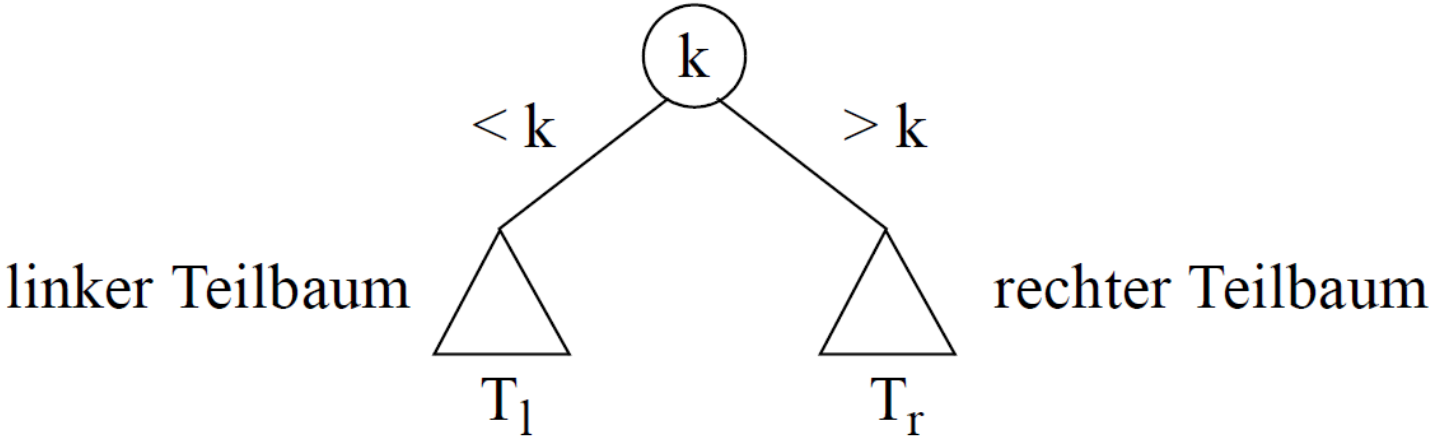
Die Operationen *Suchen*, *Einfügen* und *Entfernen* sollen alle in $O(\log n)$ Zeit durchgeführt werden.

Ansatz:

Organisation der Objektmenge als Knoten eines binären Baumes

Binäre Suchbäume: Definition I

Definition:
Ein binärer Baum heißt **binärer Suchbaum**, wenn für jeden seiner Knoten die **Suchbaumeigenschaft** gilt, d.h. alle Schlüssel im linken Teilbaum sind kleiner und alle Schlüssel im rechten Teilbaum sind größer als der Schlüssel im Knoten.



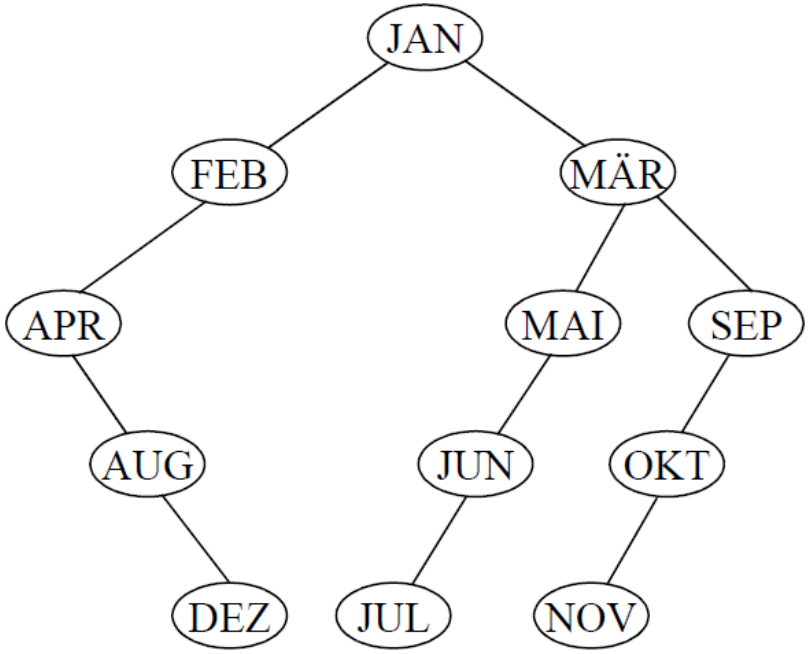
Binäre Suchbäume: Definition II

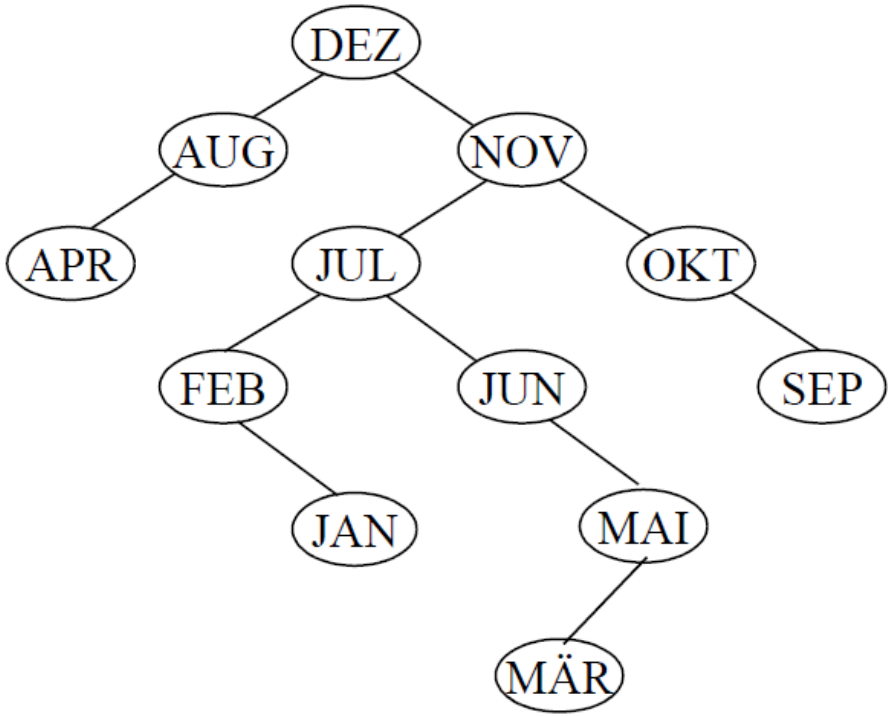
Anmerkungen:

- Zur Organisation einer gegebenen Menge von n Schlüsseln gibt es eine große Anzahl unterschiedlicher binärer Suchbäume
- Der *inorder*-Durchlauf eines binären Suchbaumes generiert die (aufsteigend) sortierte Folge der gespeicherten Schlüssel

Binäre Suchbäume: Beispiele

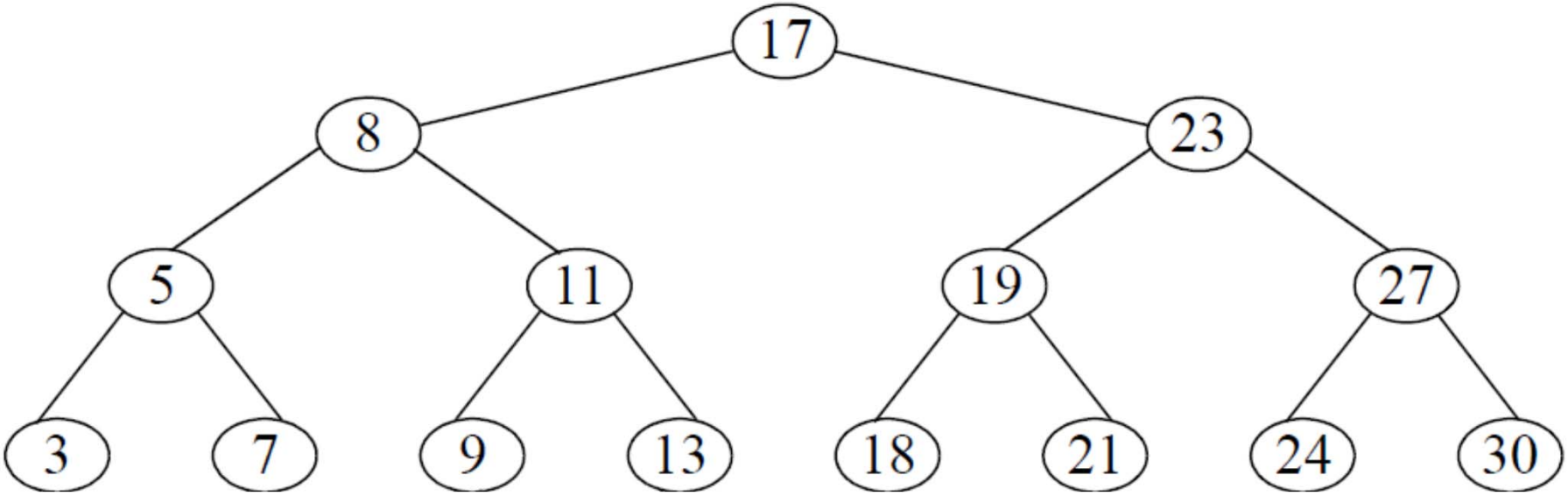
Beispiele: Zwei verschiedene binäre Suchbäume über den Monatsnamen





Binäre Suchbäume: Entscheidungsbaum

Entscheidungsbaum zur binären Suche: binärer Suchbaum



Allgemeine binäre Suchbäume: Umsetzung

```
class BinaryNode {
   int key;
   BinaryNode left;
   BinaryNode right;

   BinaryNode(int k) { key = k; left = null; right = null;}
}

public class BinarySearchTree {
   BinaryNode root;

   public BinarySearchTree () { root = null; }

   // ... Implementierung der Methoden insert, find, delete,...
}
```

Allgemeine binäre Suchbäume: Methode `insert` I

Methode **`insert(x)`** der Klasse `BinarySearchTree` fügt einen gegebenen Schlüssel x ein, falls dieser noch nicht im Baum enthalten ist.

```
public void insert(int x) throws Exception {
   root = insert(x, root);
}
```

Allgemeine binäre Suchbäume: Methode insert II

Die überladene Methode **insert(x, t)** liefert eine Referenz auf die Wurzel des Teilbaums zurück, in den x eingefügt wurde.

```
protected BinaryNode insert(int x, BinaryNode t)
      throws Exception {
   if (t == null)
      t = new BinaryNode(x);
   else if (x < t.key)
      t.left = insert(x, t.left);
   else if (x > t.key)
      t.right = insert(x, t.right);
   else
      throw new Exception("Inserting twice");
   return t;
}
```

Allgemeine binäre Suchbäume: Methode find

Methode **find(x)** sucht in einem binären Suchbaum den Schlüssel x und liefert diesen zurück, falls er gefunden wurde. Andernfalls wird eine Ausnahmebehandlung durchgeführt.

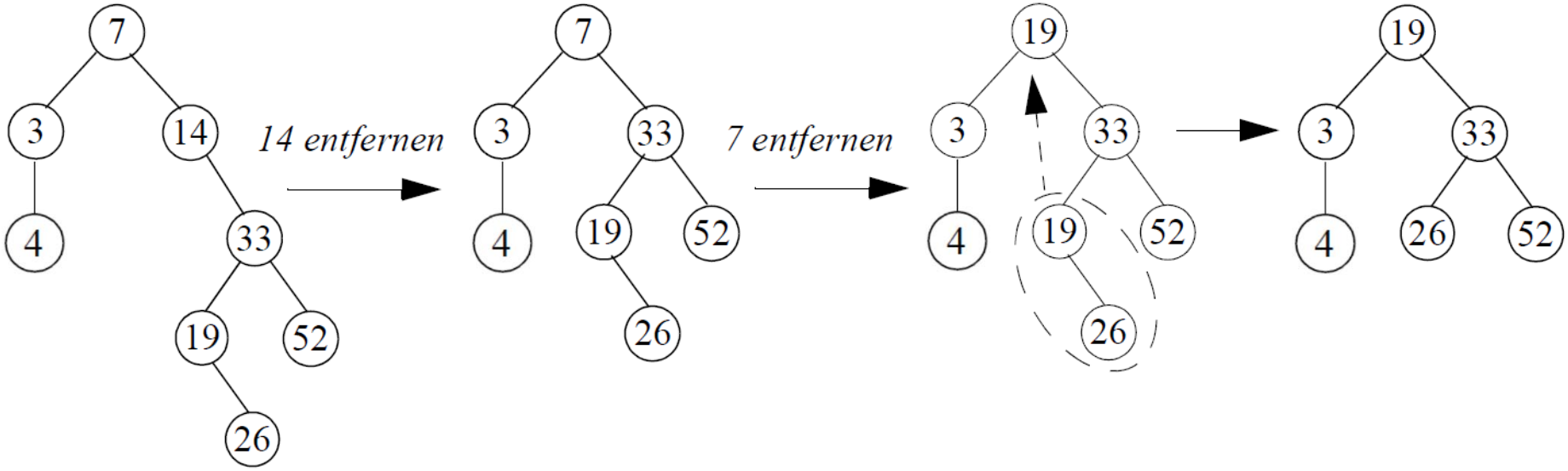
```
public int find(int x) throws Exception {
   return find(x, root).key;
}

protected BinaryNode find(int x, BinaryNode t) throws Exception {
   while (t != null) {
      if (x < t.key) t = t.left;
      else if (x > t.key) t = t.right;
      else return t;
   }
   throw new Exception("key not found");
}
```

Allgemeine binäre Suchbäume: Entfernen von Schlüsseln I

- Entfernen eines Schlüssels ist komplexer.
- Schlüssel in inneren Knoten des Baumen können auch betroffen sein.
- Suchbaumstruktur muss aufrecht erhalten bleiben.

Beispiel:

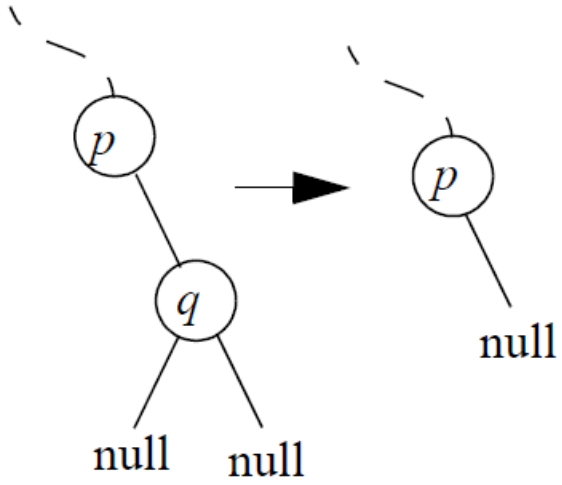


Allgemeine binäre Suchbäume: Entfernen von Schlüsseln II

Allgemein: Zwei grundsätzlich verschiedene Situationen:

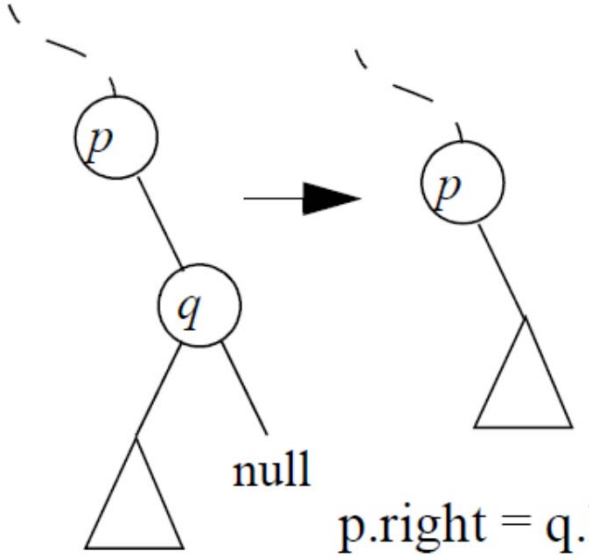
Fall 1: der Knoten q mit dem zu entfernenden Schlüssel besitzt höchstens einen Sohn (Blatt oder Halbblatt)

1a) Knoten q besitzt keinen Sohn



p.right = **null**;

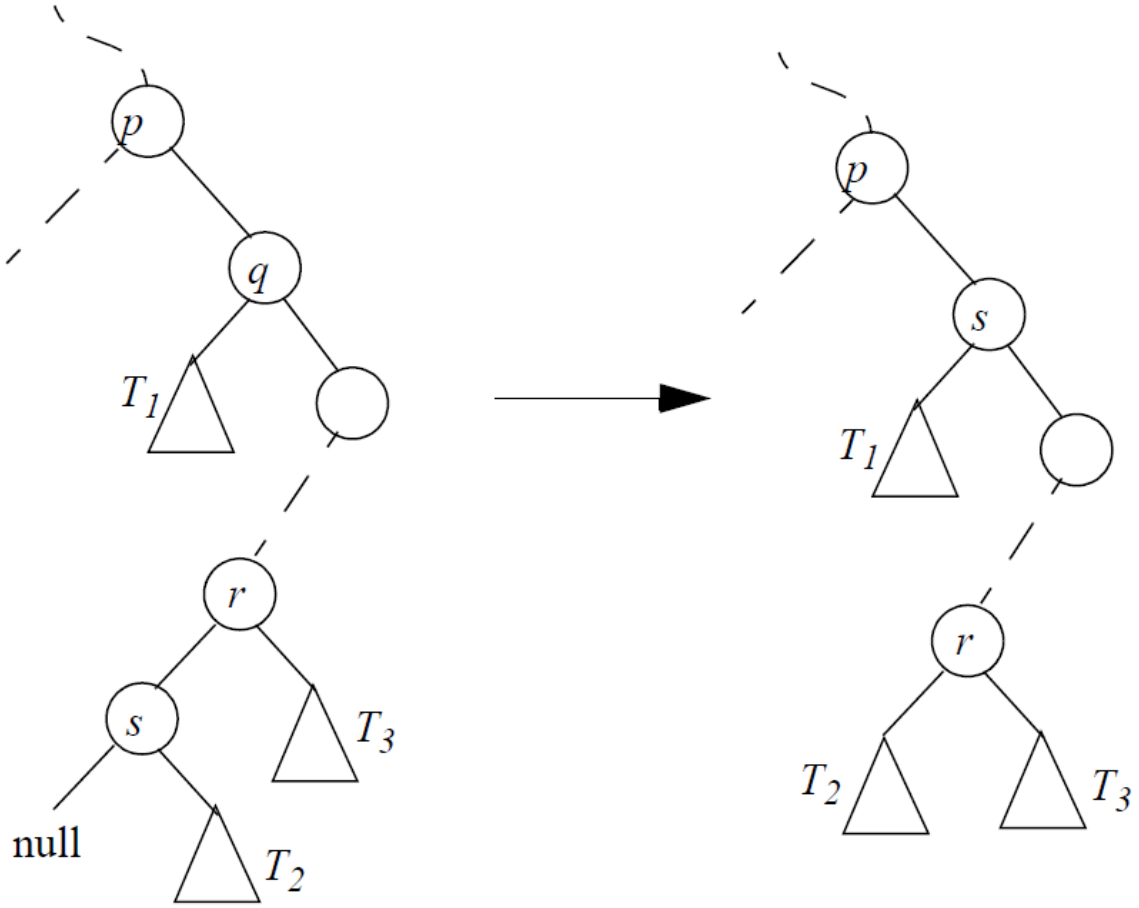
1b) Knoten q besitzt einen linken Sohn (rechts → symmetrisch



p.right = q.left;

Allgemeine binäre Suchbäume: Entfernen von Schlüsseln III

Fall 2: der Knoten q mit dem zu entfernenden Schlüssel besitzt zwei Söhne Sohn (innerer Knoten)



Allgemeine binäre Suchbäume: Methode delete I

- Methode **`delete(x)`** der Klasse `BinarySearchTree` entfernt einen Schlüssel x aus einem binären Suchbaum.
- Die Hilfsmethode **`findMin`** bestimmt den Knoten des kleinsten Schlüssels des rechten Teilbaumes (eines inneren Knotens)
- Die Hilfsmethode **`deleteMin`** entfernt diesen Knoten.

```
public void delete(int x) throws Exception {
   root = delete(x, root);
}
```

Allgemeine binäre Suchbäume: Methode delete II

```
protected BinaryNode delete(int x, BinaryNode t)
                                                throws Exception {
   if (t == null)
      throw new Exception("x does not exist (delete)");
   if (x < t.key)
      t.left = delete(x, t.left);
   else if (x > t.key)
      t.right = delete(x, t.right);
   else if (t.left != null && t.right != null) {
      // x is in inner node t
      t.key = findMin(t.right).key;
      t.right = deleteMin(t.right);
   }
   else // x is in leaf or in semi-leaf node, reroot t
      t = (t.left != null) ? t.left : t.right;
   return t;
}
```

Allgemeine binäre Suchbäume: Methode delete III


```
protected BinaryNode findMin(BinaryNode t) throws Exception {
   if ( t == null)
      throw new Exception (“key not found (findMin)”);
   while ( t.left != null) t = t.left;
   return t;
}

protected BinaryNode deleteMin(BinaryNode t) throws Exception {
   if (t == null)
      throw new Exception (“key not found (deleteMin)”);
   if (t.left != null)
      t.left = deleteMin (t.left);
   else
      t = t.right;
   return t;
}
```

Allgemeine binäre Suchbäume: Methode delete IV

Zum Verständnis:

- Der Fall des Entfernens aus einem inneren Knoten (`t.right != null && t.left != null`) wird auf eine ‘Blatt-’ (`t.right == null && t.left == null`) oder eine ‘Halbblatt-Situation’ (`t.right != null || t.left != null`) zurückgeführt, indem der zu löschende Schlüssel durch den kleinsten der größeren Schlüssel ersetzt wird.
- Dieser Schlüssel befindet sich stets in einem Blatt oder einem Halbblatt, das daraufhin aus dem Baum entfernt wird.

Allgemeine binäre Suchbäume: Laufzeitanalyse I

Alle drei Methoden sind auf einen einzigen, bei der Wurzel beginnenden Pfad des Suchbaumes, beschränkt.
→ **maximaler Aufwand** ist $O(h)$, mit h = Höhe des Baumes

Für die Höhe binärer Bäume mit n Knoten gilt:

- **Maximale Höhe** eines binären Baumes mit n Knoten ist n.
  → lineare Liste
- **Minimale Höhe** ist $\lceil \log_2(n+1) \rceil$.

Allgemeine binäre Suchbäume: Laufzeitanalyse II

Begründung für minimale Höhe:

- Für eine gegebene Anzahl n von Knoten haben die sogenannten vollständig ausgeglichenen binären Bäume minimale Höhe.
- In einem vollständig ausgeglichenen binären Baum müssen alle Levels bis auf das unterste vollständig besetzt sein.
- Die maximale Anzahl n von Knoten in einem vollständig ausgeglichenen binären Baum der Höhe h ist:

$$n = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

- Daraus folgt: $h_{min} = \lceil \log_2(n+1) \rceil$

Bemerkung: Es gilt: $\lceil \log_2(n+1) \rceil = \lfloor \log_2(n) \rfloor + 1 \quad \forall n \in \mathbb{N}$

Allgemeine binäre Suchbäume: Durchschnittsanalyse & Kritikpunkt

Durchschnittsanalyse

- Unter den Annahmen, dass
  - der Baum nur durch Einfügungen entstanden ist und
  - alle möglichen Permutationen der Eingabereihenfolge gleichwahrscheinlich sind

  ergibt eine aufwendige Durchschnittsanalyse einen mittleren Wert $h_{\emptyset} = 2 \cdot \ln 2 \cdot \log n \approx 1{,}386 \cdot \log n$.
- Der durchschnittliche Zeitbedarf für Suchen, Einfügen und Entfernen ist damit $O(\log n)$

Kritikpunkt

- Im Worst-Case ist der Aufwand aller drei Operationen $O(n)$.

Vollständig ausgeglichene binäre Bäume I

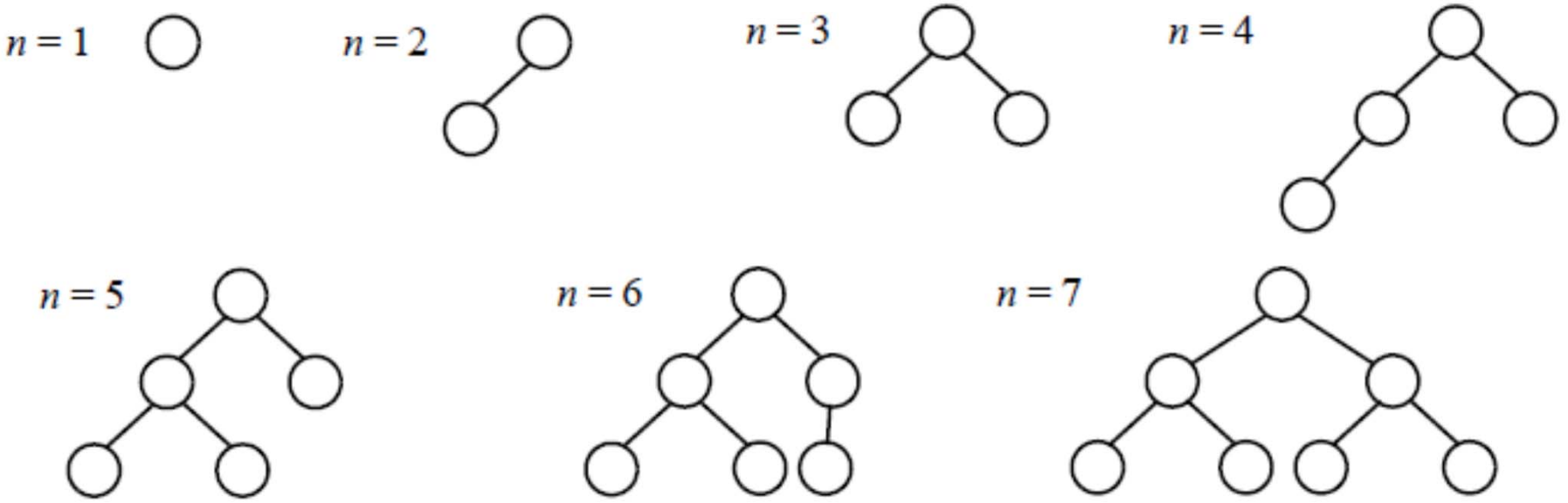
Definition:

- Binäre Suchbäume, bei denen alle Levels bis auf das unterste vollständig besetzt sind, nennt man **vollständig ausgeglichene binäre Suchbäume**.
- Solche Bäume besitzen die minimale Höhe $\lceil \log_2(n+1) \rceil$

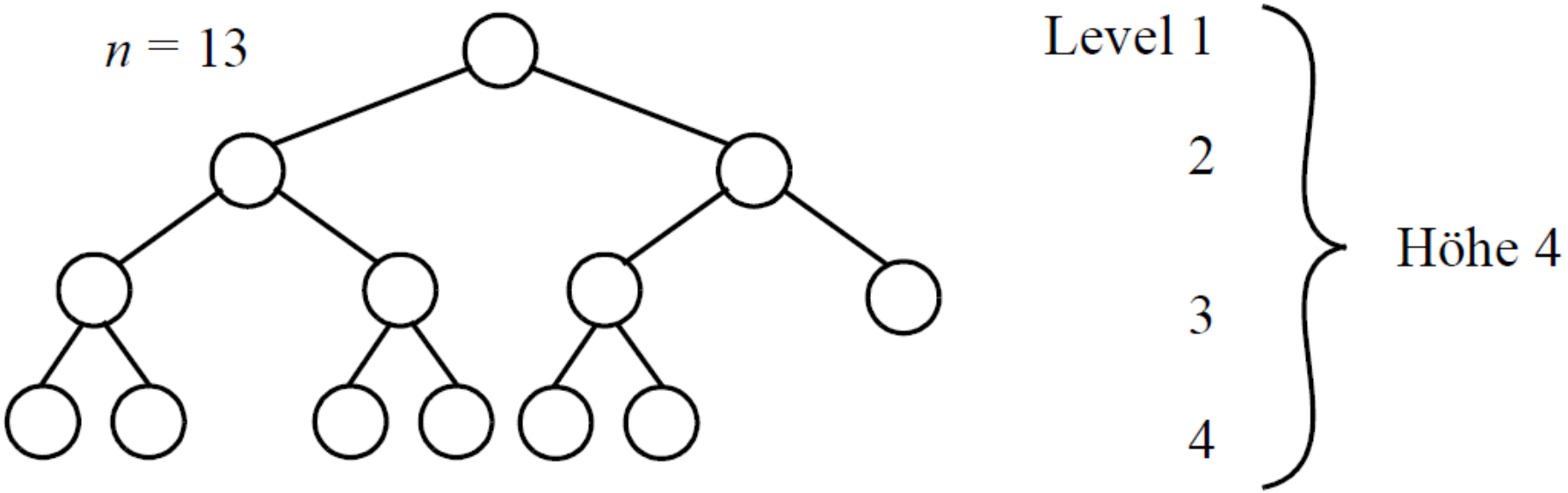
→ **optimaler Zeitaufwand für Suchoperationen**

Vollständig ausgeglichene binäre Bäume II

Vollständig ausgeglichene binäre Bäume für verschiedene Knotenzahlen n:



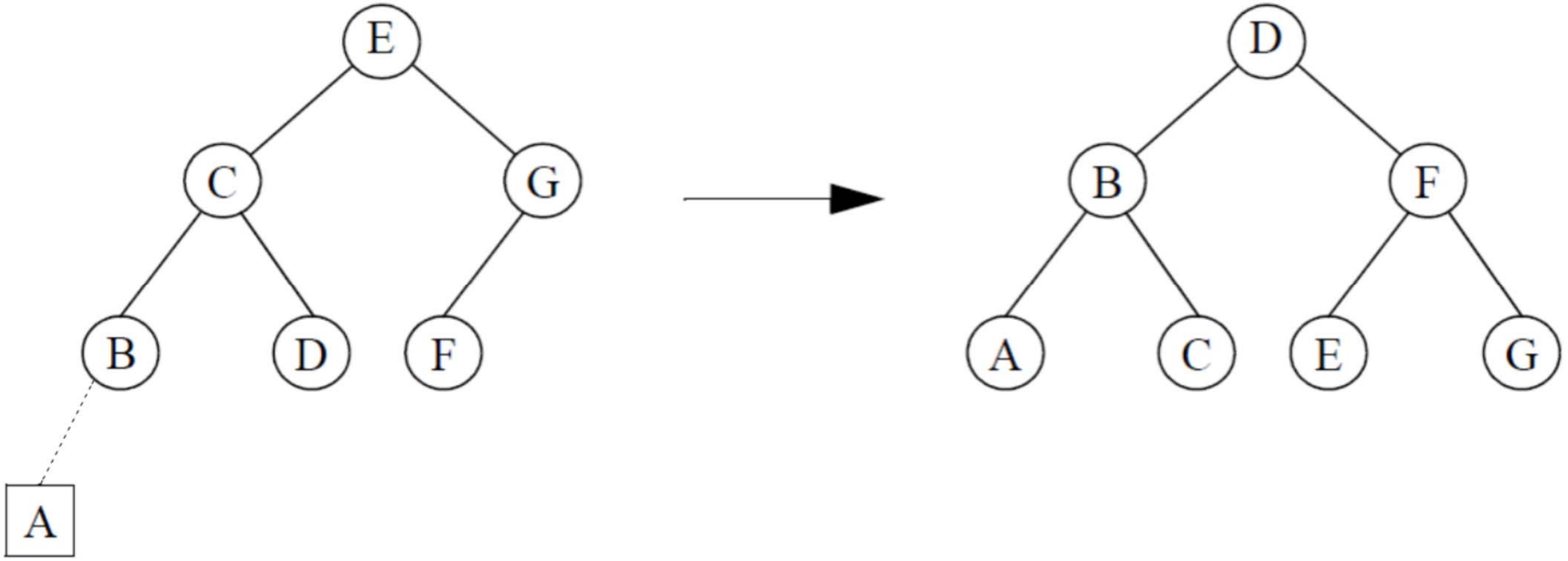
Vollständig ausgeglichene binäre Bäume III



Vollständig ausgeglichene binäre Bäume IV

Beispiel:

Einfügen von Schlüssel „A“ in einen bestehenden Baum:



Vollständig ausgeglichene binäre Bäume V

Problem:
Der Baum muss beim Einfügen von Schlüssel „B“ **vollständig reorganisiert** werden.

→ **Einfügezeit im schlechtesten Fall:** $O(n)$.

Auswahl einer Kompromisslösung mit den Eigenschaften:

- Die Höhe des Baumes ist im schlechtesten Fall $O(\log n)$.
- Reorganisationen bleiben auf den Suchpfad zum einzufügenden, bzw. zu entfernenden Schlüssel beschränkt → im schlechtesten Fall in $O(\log n)$ ausführbar.

Balancierte Suchbäume

Definition:
Eine Klasse von Suchbäumen heißt **balanciert**, falls:

- $h_{max} = O(\log n)$
- Die Operationen Suchen, Einfügen und Entfernen sind auf einem Pfad von der Wurzel zu einem Blatt beschränkt und benötigen damit im schlechtesten Fall $O(\log n)$ Zeit.

AVL-Bäume I

Definition: *(Adelson-Velskij und Landis 1962)*
Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume T_r und T_l der Wurzel gilt:

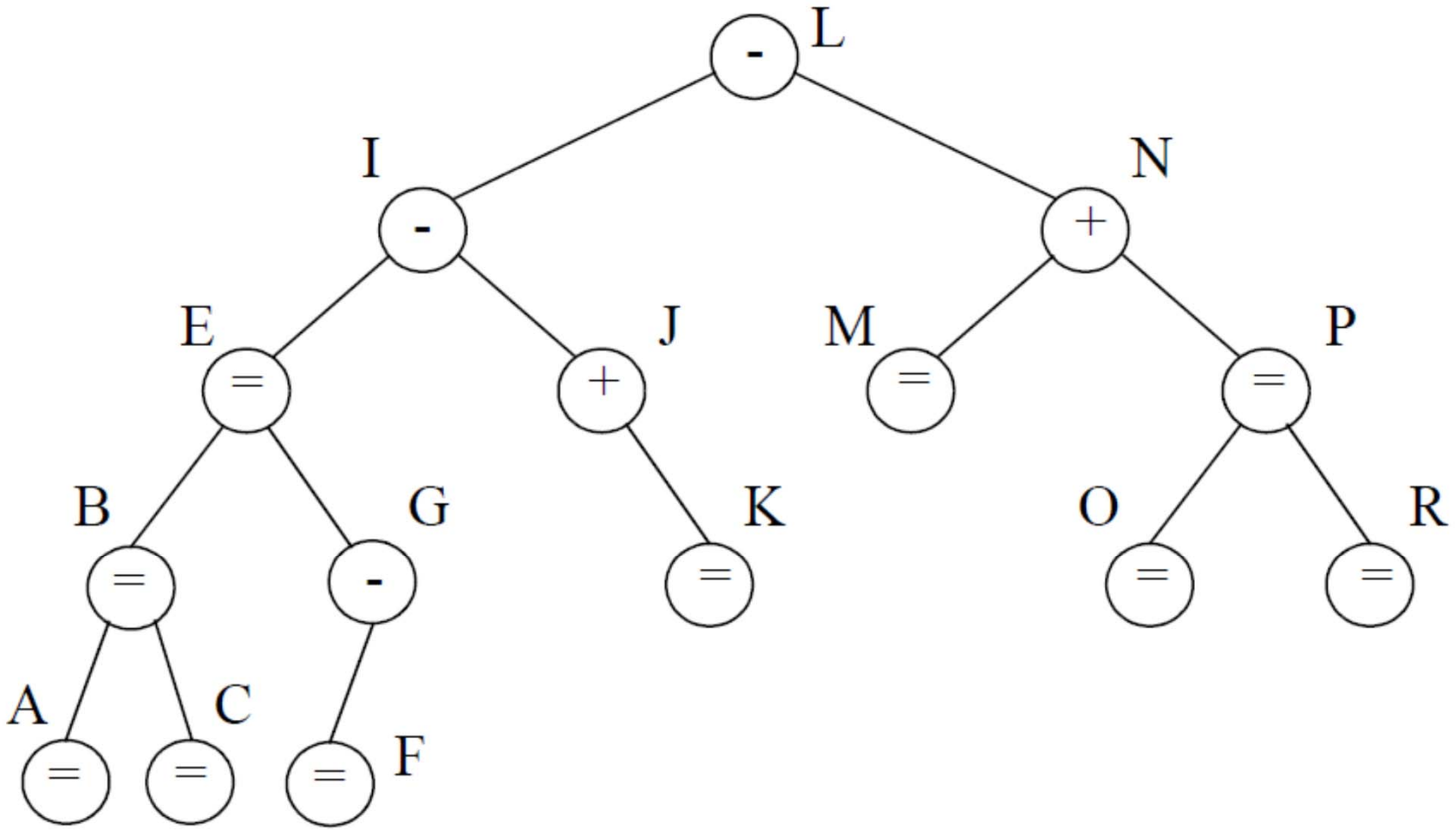
- $|h(T_r) - h(t_l)| \leq 1$
- T_r und T_l sind ihrerseits AVL-Bäume. (rekursive Definition)

Eigenschaften:

- Der Wert $h(T_r) - h(T_l)$ wird als **Balancefaktor** (BF) eines Knotens bezeichnet. Er kann in einem AVL-Baum nur die Werte -1, 0 oder 1 annehmen. (Darstellung: $-$, $=$ und $+$)
- Stukturverletzungen durch Einfügungen bzw. Entfernungen von Schlüssel erfordern Rebalancierungsoperationen.

AVL-Bäume II

Beispiel für einen AVL-Baum



AVL-Bäume: Minimale und maximale Höhe I

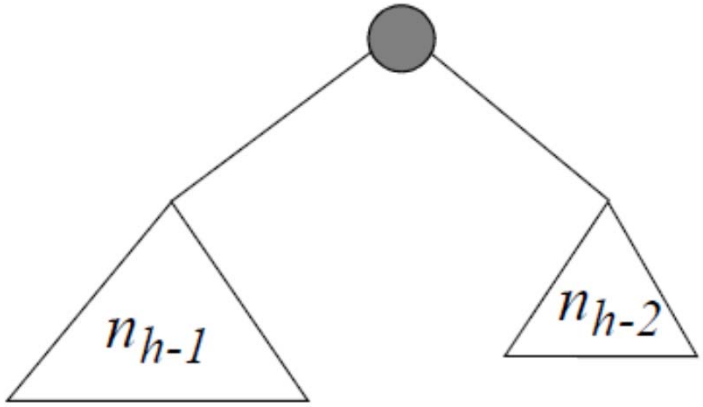
Behauptungen:

- Die minimale Höhe $h_{min}(n)$ eines AVL-Baumes mit n Schlüsseln ist $\log_2(n+1)$. Dies folgt aus der Tatsache, dass ein AVL-Baum minimaler Höhe einem vollständig ausgeglichenen binären Suchbaum entspricht.
- Die maximale Höhe $h_{max}(n)$ eines AVL-Baumes mit n Schlüsseln ist $O(\log n)$.

AVL-Bäume: Minimale und maximale Höhe II

Beweis:

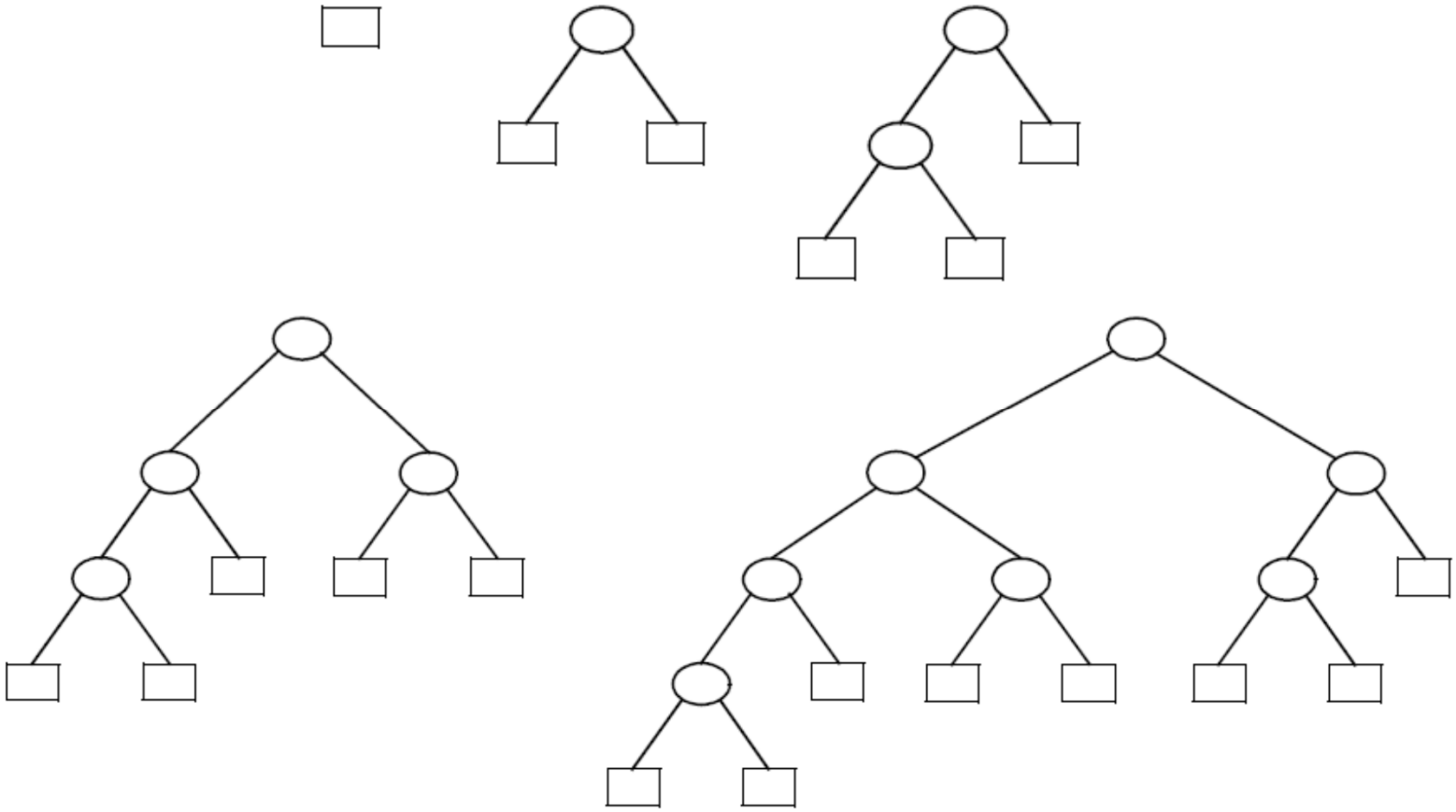
- Maximale Höhe wird von **minimalen AVL-Bäumen** realisiert.
  = AVL-Bäume, die für eine gegebene Höhe h die minimale Anzahl von Schlüsseln abspeichert.
- Gestalt von minimalen AVL-Bäume (bis auf Symmetrie):



- Sei n_h die minimale Anzahl von Schlüsseln in einem AVL-Baum der Höhe h. Dann gilt:
  $n_h = n_{h-1} + n_{h-2} + 1 \quad$ für $\quad h \geq 2 \;$ und $\; n_0 = 0, n_1 = 1$

AVL-Bäume: Minimale und maximale Höhe III

- Die minimalen AVL-Bäume der Höhen $h = 0,1,2,3,4$ haben bis auf Symmetrie folgende Gestalt:



AVL-Bäume: Minimale und maximale Höhe IV

- Rekursionsgleichung für n_h erinnert an die Definition der *Fibonacci-Zahlen*:

$$fib(n) = \begin{cases} n & \text{für } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{für } n > 1 \end{cases}$$

h	0	1	2	3	4	5	6
n_h	0	1	2	4	7	12	20
$fib(h)$	0	1	1	2	3	5	8

- **Hypothese:** $n_h = fib(h+2) - 1$

AVL-Bäume: Minimale und maximale Höhe V

- **Beweis:** Induktion über h
  - Induktionsanfang: $n_0 = fib(2) - 1 = 1 - 1 = 0$
  - Induktionsschluss: $n_{h+1} = n_h + n_{h-1} + 1$
    $= 1 + fib(h+2) - 1 + fib(h+1) -$
    $= fib(h+3) - 1$

- Hilfssatz: (Beweis durch vollständige Induktion wird hier übergangen)

$$fib(n) = \frac{1}{\sqrt{5}} \cdot (\varphi_1^n - \varphi_2^n) \text{ mit } \varphi_1 = \frac{1+\sqrt{5}}{2}, \varphi_2 = \frac{1-\sqrt{5}}{2} \approx -0{,}618$$

- Für jede beliebige Schlüsselanzahl $n \in \mathbb{N}$ gibt es ein eindeutiges $h_{max}(n)$ mit:

$$n_{h_{max}(n)} \leq n < n_{h_{max}(n)+1}$$

- Mit obiger Hypothese folgt hieraus:

$$n + 1 \geq fib(h_{max}(n) + 2)$$

AVL-Bäume: Minimale und maximale Höhe VI

- Durch Einsetzen:

$$n+1 \geq \frac{1}{\sqrt{5}} \cdot \left(\varphi^{h_{max}(n)+2} - \underbrace{\varphi^{h_{max}(n)+2}}_{< 0{,}62 < \frac{\sqrt{5}}{2}} \right) \geq \frac{1}{\sqrt{5}} \cdot \varphi_1^{h_{max}(n)+2} - \frac{1}{2}$$

  Und damit:

$$\frac{1}{\sqrt{5}} \cdot \varphi_1^{h_{max}(n)+2} \leq n + \frac{3}{2}$$

- Durch Auflösen nach $h_{max}(n)$ ergibt sich:

$$\log_{\varphi_1}\left(\frac{1}{\sqrt{5}}\right) + h_{max}(n) + 2 < \log_{\varphi_1}\left(n + \frac{3}{2}\right)$$

AVL-Bäume: Minimale und maximale Höhe VII

- Und für $h_{max}(n)$:

$$h_{max}(n) \leq \log_{\varphi_1}\left(n + \frac{3}{2}\right) - \left(\log_{\varphi_1}\left(\frac{1}{\sqrt{5}}\right) + 2\right) \leq$$

$$\leq \log_{\varphi_1}(n) + const \overset{*)}{=} \log_{\varphi_1}(2) \cdot \log_2(n) + const$$

$$\overset{*)}{=} \frac{\ln 2}{\ln \varphi_1} \cdot \log_2(n) + const \approx 1{,}44 \cdot \log_2(n) + const$$

*) mit $\left(\log_b(a) = \frac{\log_c(a)}{\log_c(b)}\right)$

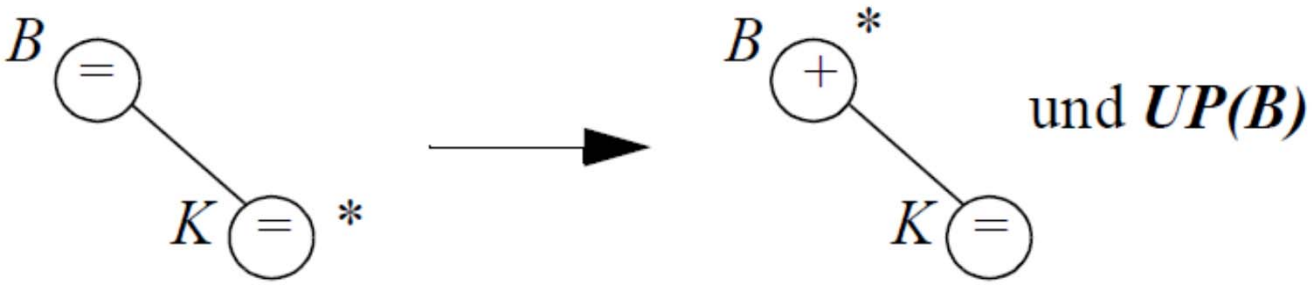
- Für große Schlüsselanzahlen ist die Höhe eines AVL-Baumes somit um maximal 44% größer als die des vollständig ausgeglichenen binären Suchbaumes.

- Also gilt die Behauptung: $h_{max}(n) = O(\log n)$.

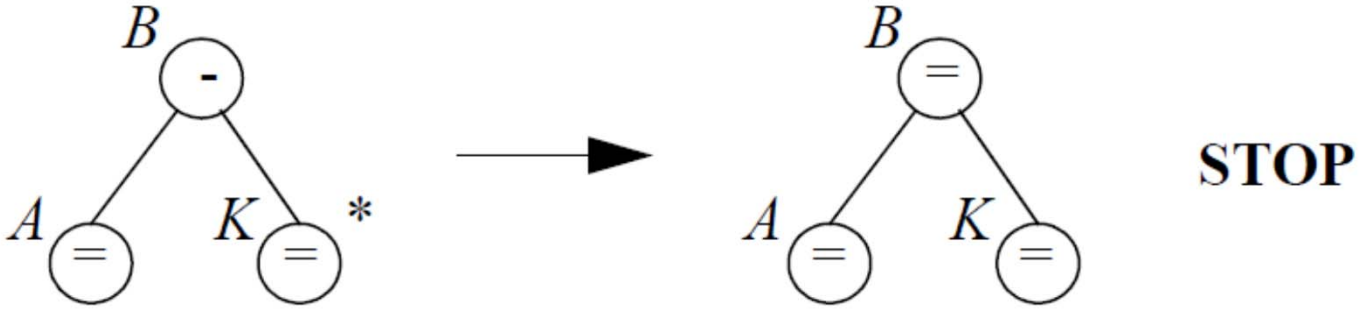
AVL-Bäume: Einfügen von Schlüsseln I

(Symmetrische Fälle sind in der folgenden Beschreibung nicht dargestellt)

- Schlüssel k wird in neuen Sohn K des Knotens B eingefügt:
  - Fall 1: B ist ein Blatt



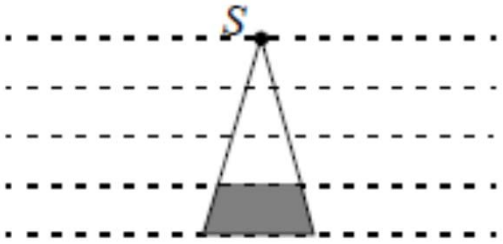
  - Fall 2: B hat einen linken Sohn



AVL-Bäume: Einfügen von Schlüsseln II

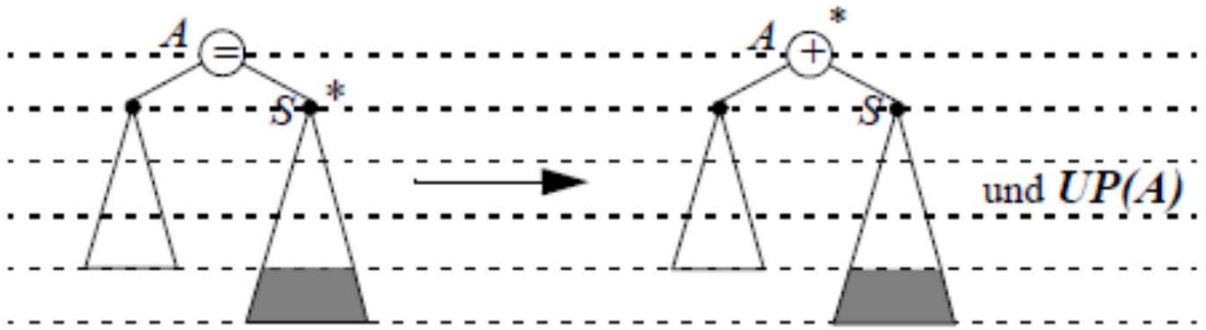
- Methode `UP(S)` wird für einen Knoten S aufgerufen, dessen Teilbaum in seiner Höhe um 1 gewachsen ist.
- S ist die Wurzel eines korrekten AVL-Baumes.

→ **Mögliche Strukturverletzung** durch zu hohen Teilbaum!



AVL-Bäume: Einfügen von Schlüsseln III

- **Fall 1:** Vater von S hat BF ‚=‘
  - **1.1:** Vater von S ist nicht die Wurzel

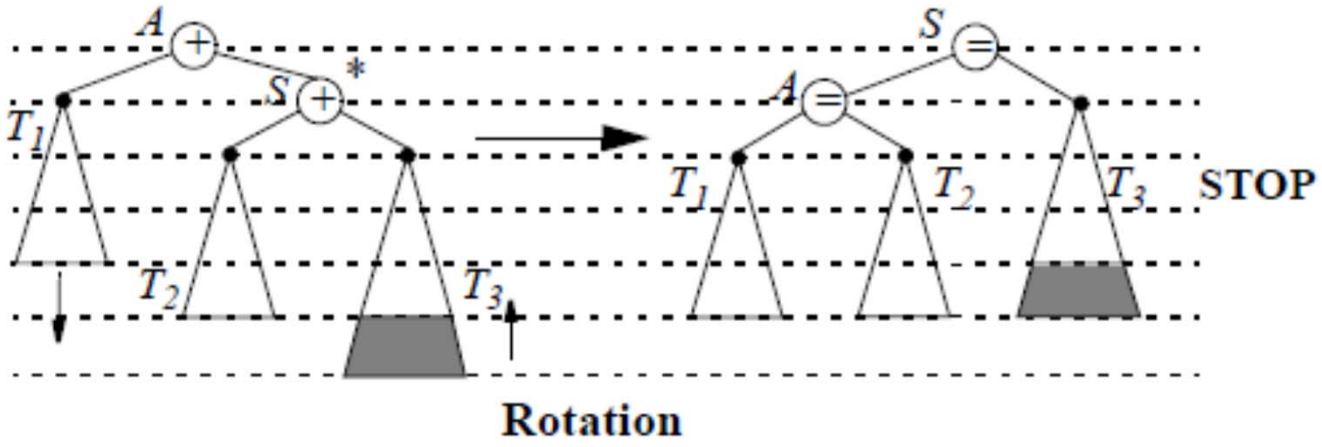


  - **1.2:** Vater von S ist die Wurzel
    → dieselbe Transformation und **STOP**.

- **Fall 2:** Vater von S hat BF ‚+‘ oder ‚–‘ und S ist die Wurzel des kürzeren Teilbaumes
  → In beiden Fällen wird der BF im Vater zu ‚=‘ und **STOP**

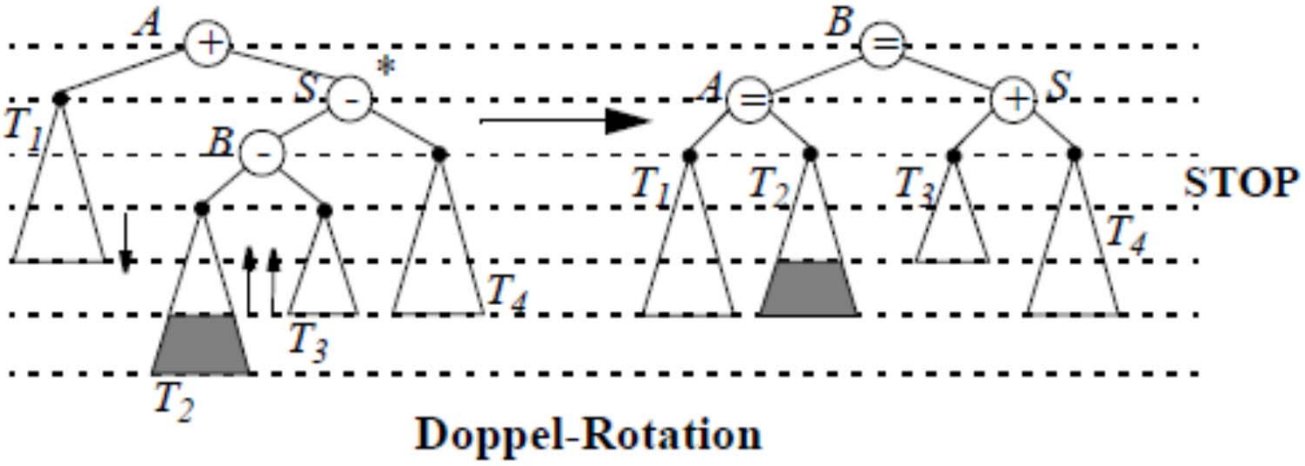
AVL-Bäume: Einfügen von Schlüsseln IV

- **Fall 3:** Vater von S hat BF ‚+' oder ‚–' und S ist die Wurzel des höheren Teilbaumes
  - **3.1:** Vater von S hat BF ‚+' und S hat BF ‚+'



AVL-Bäume: Einfügen von Schlüsseln V

- **3.2:** Vater von S hat BF ‚+' und S hat BF ‚–'
  - **3.2.1:** B hat BF ‚–'



Doppel-Rotation

  - **3.2.2:** B hat BF ‚+'
- **3.3:** Vater von S hat BF ‚–' und S hat BF ‚–' → symmetrisch zu 3.1
- **3.4:** Vater von S hat BF ‚–' und S hat BF ‚+' → symmetrisch zu 3.2

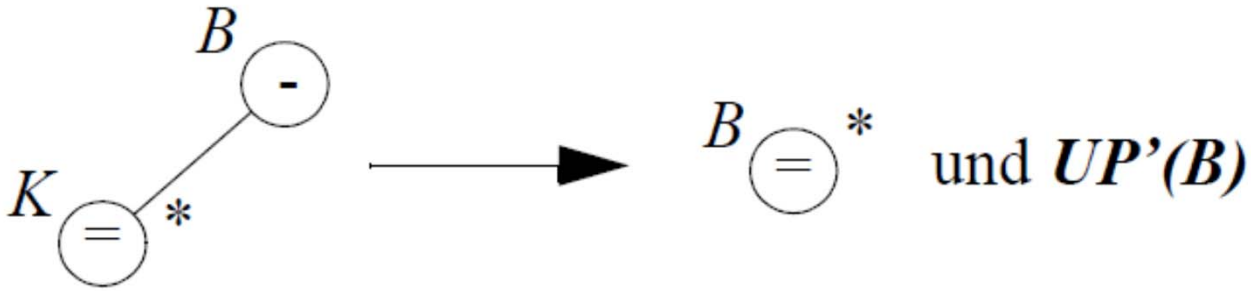
Beim Einfügen genügt eine einzige Rotation, bzw. Doppelrotation um eine Strukturverletzung zu beseitigen.

AVL-Bäume: Entfernen von Schlüsseln I

(Symmetrische Fälle sind in der folgenden Beschreibung nicht dargestellt)

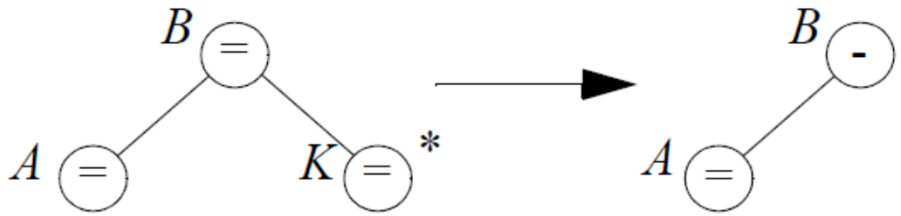
Knoten K mit Schlüssel k wird entfernt.

- **Fall 1:** K hat höchstens einen Sohn
  - **1.1:** K ist ein Blatt
    - **1.1.1:** K hat keinen Bruder



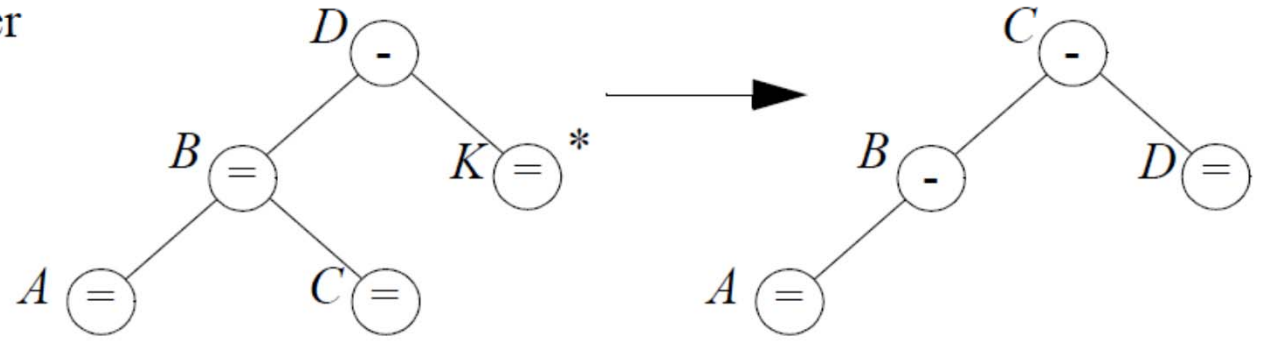
AVL-Bäume: Entfernen von Schlüsseln II

- **1.1.2:** K hat einen Bruder



STOP

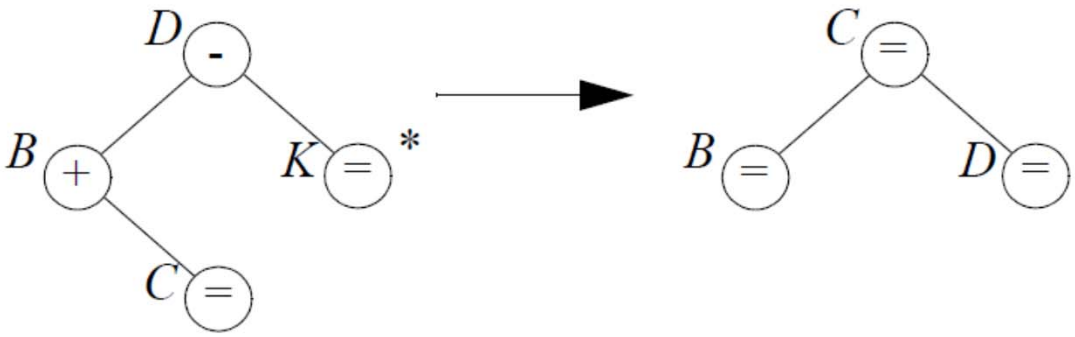
oder



STOP

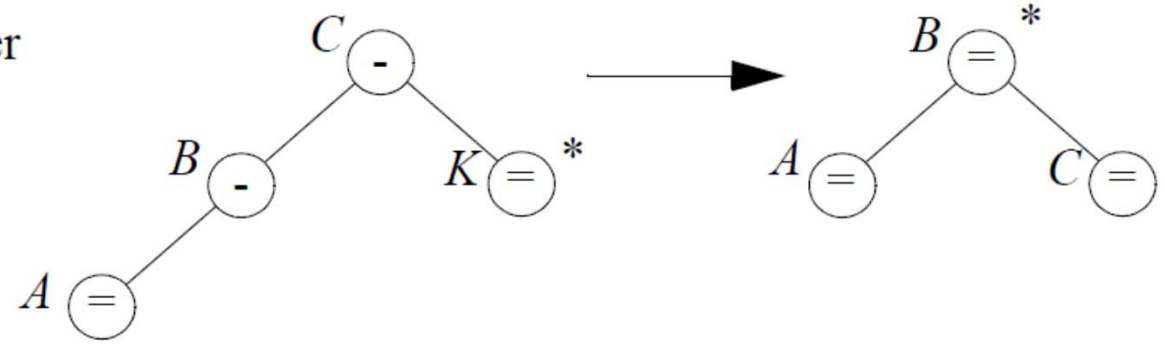
AVL-Bäume: Entfernen von Schlüsseln III

oder



und ***UP'(C)***

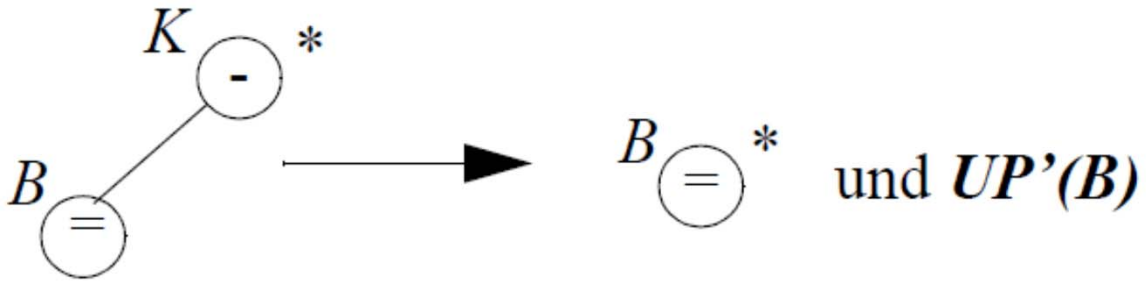
oder



und ***UP'(B)***

AVL-Bäume: Entfernen von Schlüsseln IV

- **1.2:** K hat genau einen Sohn
  - **1.2.1:** K hat einen linken Sohn



  - **1.2.2:** K hat einen rechten Sohn → symmetrisch zu 1.2.1

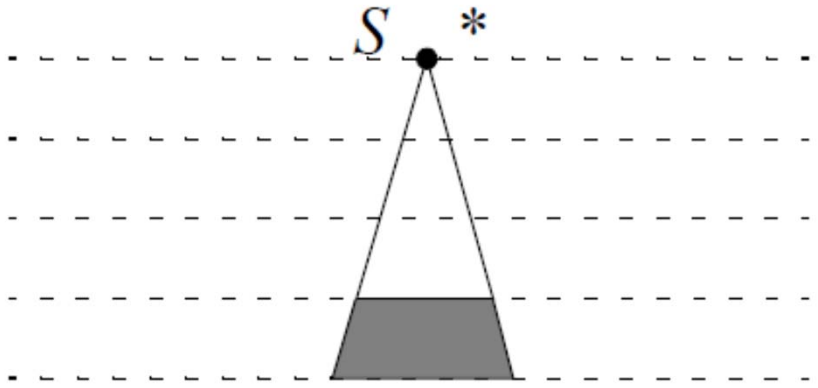
- **Fall 2:** K ist ein innerer Knoten (hat zwei Söhne)
  - Man bestimme in dem AVL-Baum den **kleinsten** Schlüssel s, der **größer als k** ist.
  - Schlüssel s ist in einem Halbblatt S.
  - Ersetze k durch s und entferne den Schlüssel s

    → somit haben wir Fall 2 auf Fall 1 zurückgeführt.

AVL-Bäume: Entfernen von Schlüsseln V

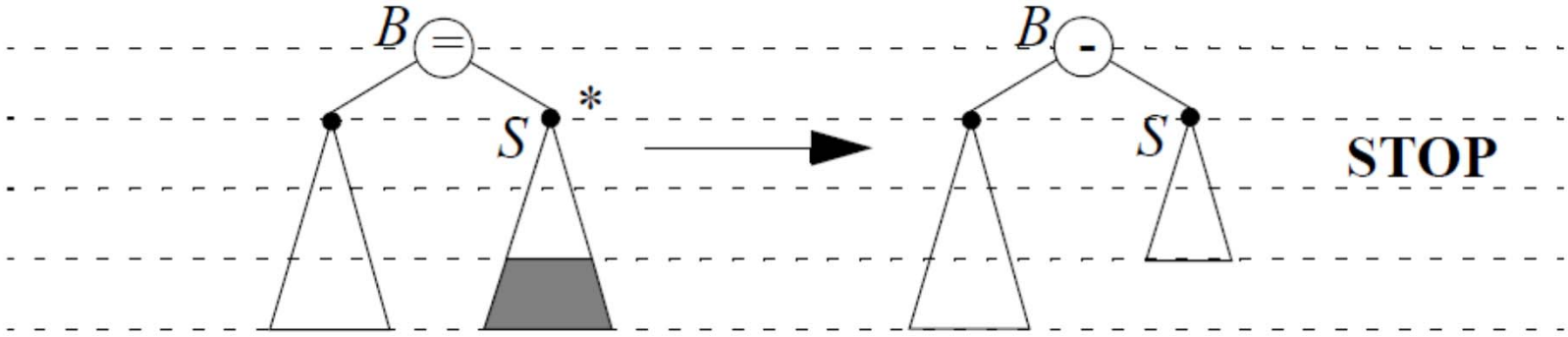
- Methode UP‘(S) wird aufgerufen für einen Knoten S, dessen Teilbaum in seiner Höhe um 1 reduziert ist.
- Der Teilbaum mit Wurzel S ist ein korrekter AVL-Baum.

→ **mögliche Strukturverletzung** durch zu niedrigen Teilbaum

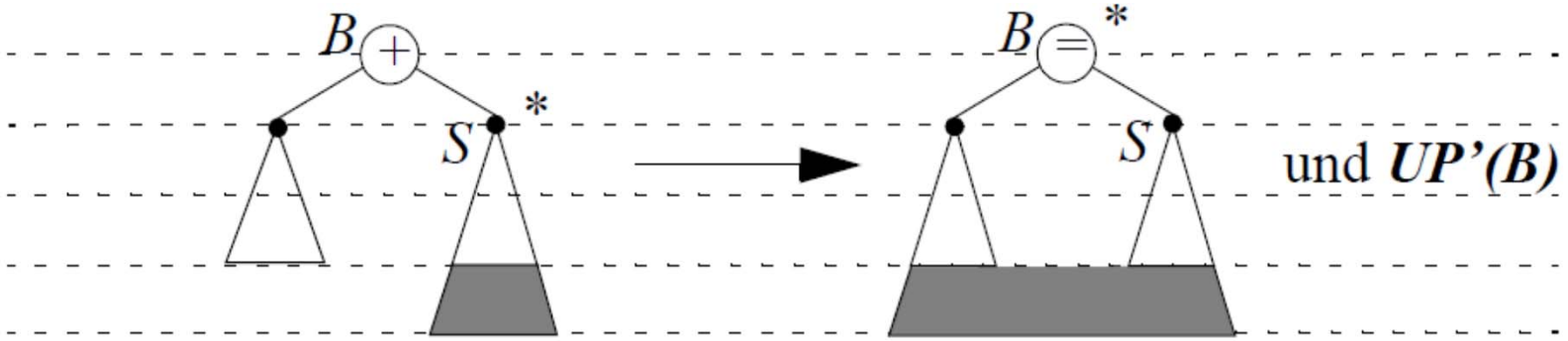


AVL-Bäume: Entfernen von Schlüsseln VI

- **Fall 1:** Vater von S hat BF ‚=‘

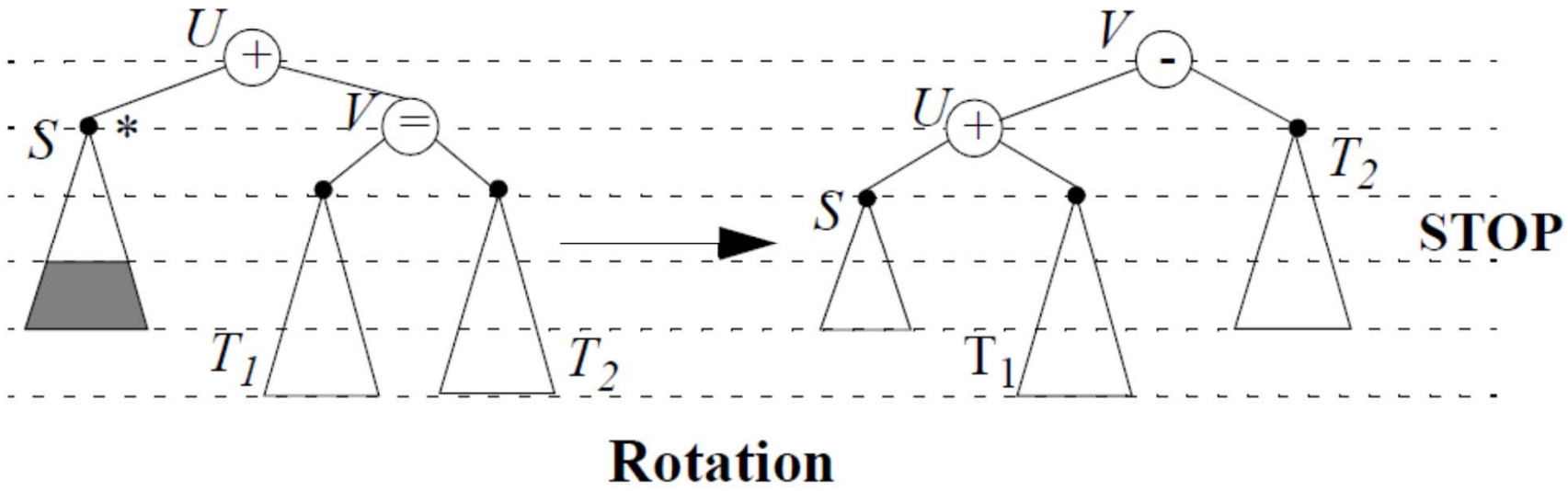


- **Fall 2:** Vater von S hat BF ‚+‘ oder ‚–‘ und S ist die Wurzel des höheren Teilbaums.



AVL-Bäume: Entfernen von Schlüsseln VII

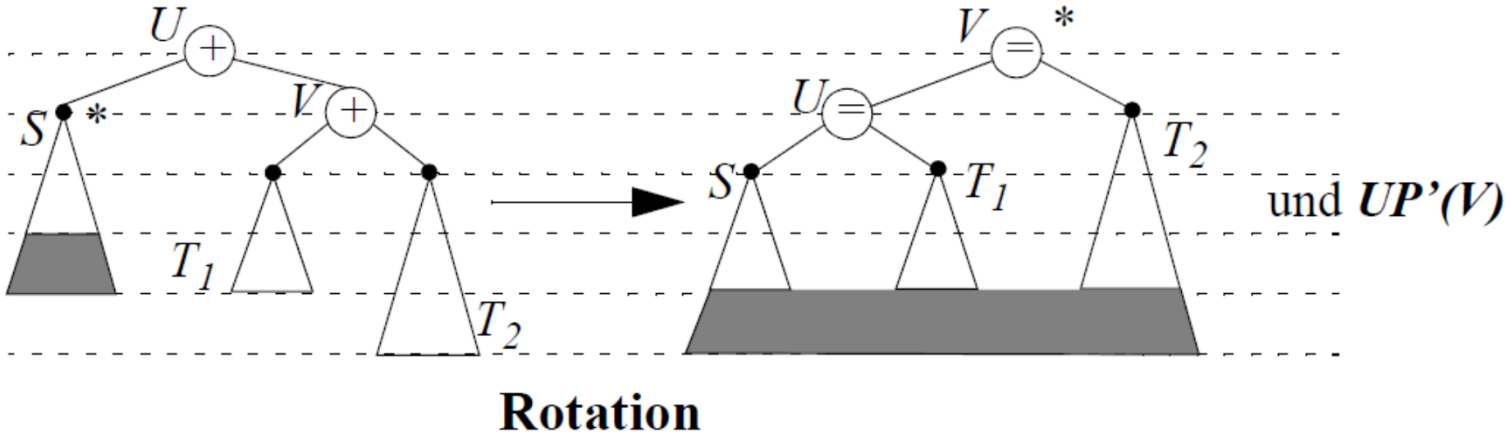
- **Fall 3:** Vater von S hat BF ‚+' oder ‚–' und S ist die Wurzel des kürzeren Teilbaums.
  - **3.1:** Vater von S hat BF ‚+'
    - **3.1.1:** Bruder von S hat BF ‚='



Rotation

AVL-Bäume: Entfernen von Schlüsseln VIII

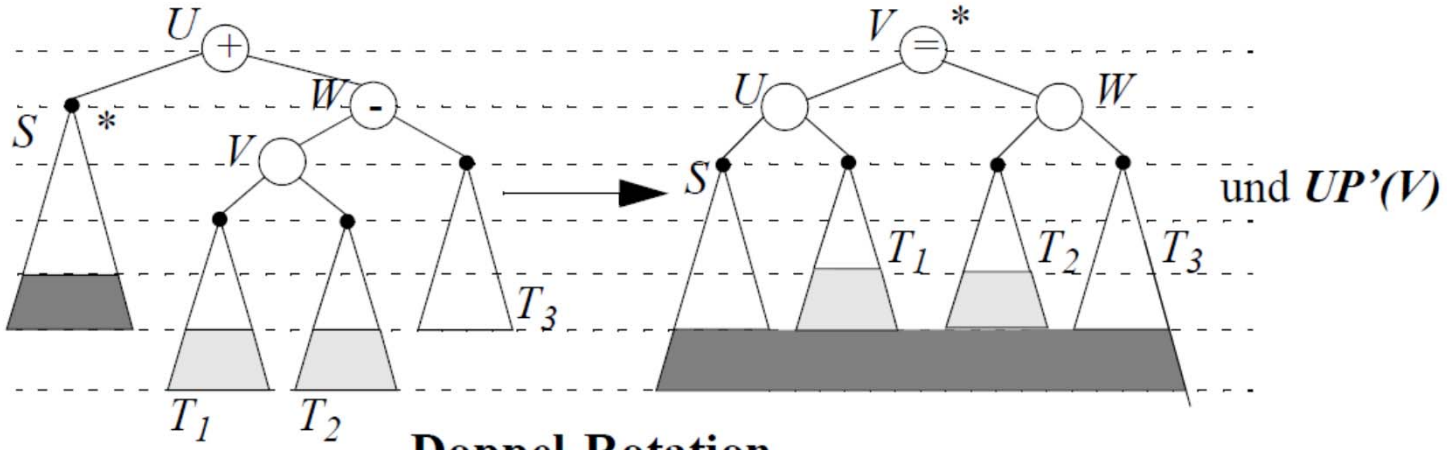
- **3.1.2:** Bruder von S hat BF ‚+'



Rotation

- **3.1.3:** Bruder von S hat BF ‚–'

Mindestens einer der beiden Bäume T_1 und T_2 hat die durch den hell schraffierten Bereich angegebene Höhe



Doppel-Rotation

AVL-Bäume: Entfernen von Schlüsseln IX

  - **3.2:** Vater von S hat BF ‚–' → symetisch zu 3.1
- **Fall 4:** S ist die Wurzel → **STOP**

- Im Falle von Entferne-Operationen wird eine mögliche Strukturverletzung nicht notwendigerweise durch eine einzige Rotation bzw. Doppelrotation beseitigt.
- Im schlechtesten Fall muss auf dem Suchpfad bottom-up vom zu entfernenden Schlüssel bis zur Wurzel auf jedem Level eine Rotation bzw. Doppelrotation durchgeführt werden.

- **Korollar:** Die **AVL-Bäume** bilden eine Klasse **balancierter Bäume.**