Kapitel 2 Suchverfahren

Gegeben sei eine Menge von Objekten, die durch (eindeutige) Schlüssel charakterisiert sind. Aufgabe von Suchverfahren ist die Suche bestimmter Objekte anhand ihres Schlüssels.

Bisher: zwei Methoden der Speicherung solcher Objekte

- sequentielle Speicherung ➠ sortiertes Array (binäre Suche)
- verkettete Speicherung ➠ lineare Liste

Zeitaufwand im schlechtesten Fall für eine Menge von *n* Elementen;

Operation	sequentiell gespeichert (sortiertes ARRAY)	verkettet gespeichert (lineare Liste)
Suche Objekt mit gegebenem Schlüssel	$O(\log n)$	$O(n)$
Einfügen an bekannter Stelle	$O(n)$	$O(1)$
Entfernen an bekannter Stelle	$O(n)$	$O(1)$

Im folgenden werden wir Verfahren behandeln, die sowohl die Suche als auch das Einfügen und Entfernen "effizient" unterstützen (bessere Laufzeitkomplexität als O(n)).

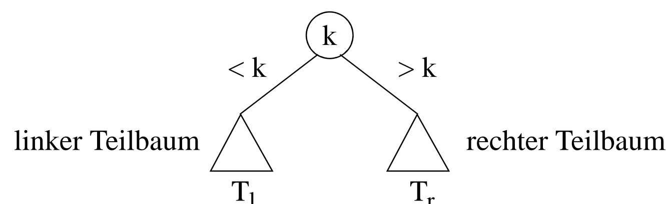
2.1 Binäre Suchbäume

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.

Definition:

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

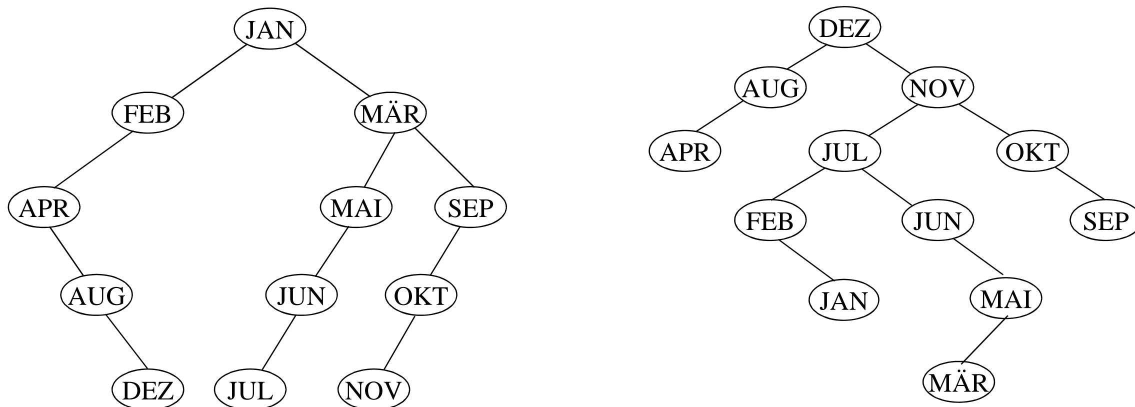


Anmerkungen:

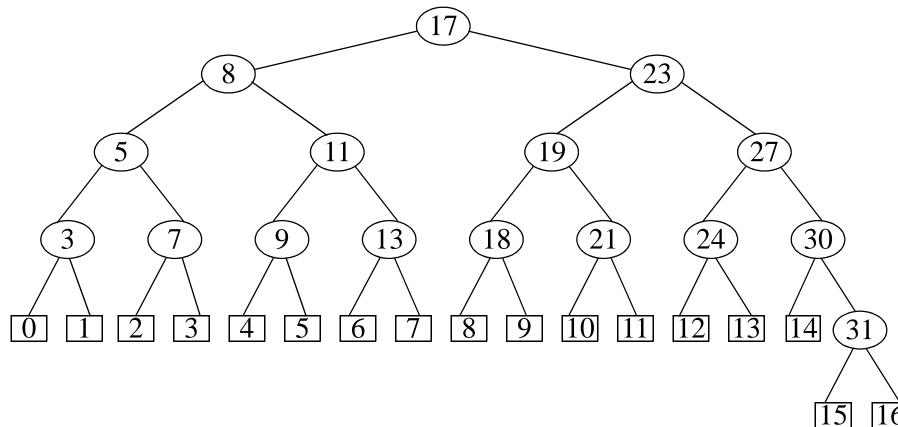
- Zur Organisation einer gegebenen Menge von *n* Schlüsseln gibt es eine große Anzahl unterschiedlicher binärer Suchbäume.
- Der *inorder*-Durchlauf eines binären Suchbaumes generiert die (aufsteigend) sortierte Folge der gespeicherten Schlüssel.

Beispiele:

Zwei verschiedene binäre Suchbäume über den Monatsnamen:



Der Entscheidungsbaum zur binären Suche ist ein binärer Suchbaum:



2.1.1 Allgemeine binäre Suchbäume

```
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,...
}
```

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

```
// Methoden der Klasse BinarySearchTree
public void insert (int x) throws Exception
{ root = insert (x, root);}

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;
}
```

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

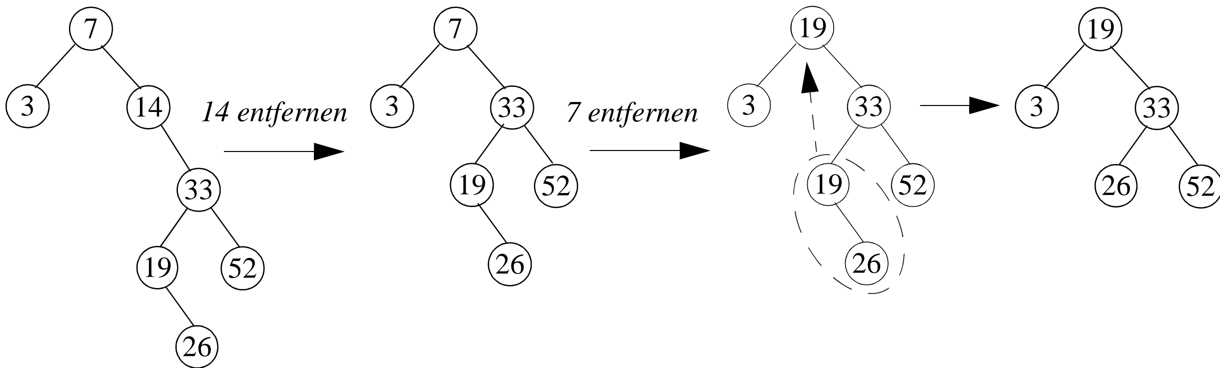
Die folgende Methode *find (x)* sucht in einem binären Suchbaum den Schlüssel *x* und liefert diesen zurück, falls er gefunden wurde. Andernfalls wird eine Ausnahmebehandlung durchgeführt.

```
// Methoden der Klasse BinarySearchTree
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");
}
```

Das Entfernen eines Schlüssels ist etwas komplexer, da auch Schlüssel in inneren Knoten des Baumes betroffen sein können und die Suchbaumstruktur aufrecht erhalten werden muß.

Hierzu zunächst ein Beispiel:



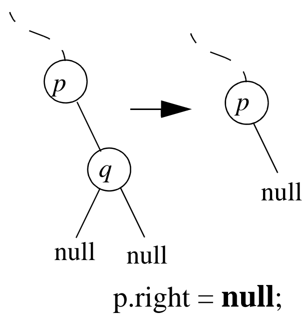
Allgemein treten zwei grundsätzlich verschiedene Situationen auf:

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

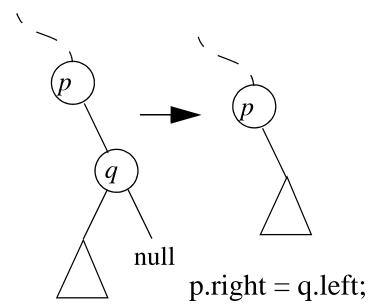
Fall 2: der Knoten q mit dem zu entfernenden Schlüssel besitzt zwei Söhne (innerer Knoten)

Fall 1:

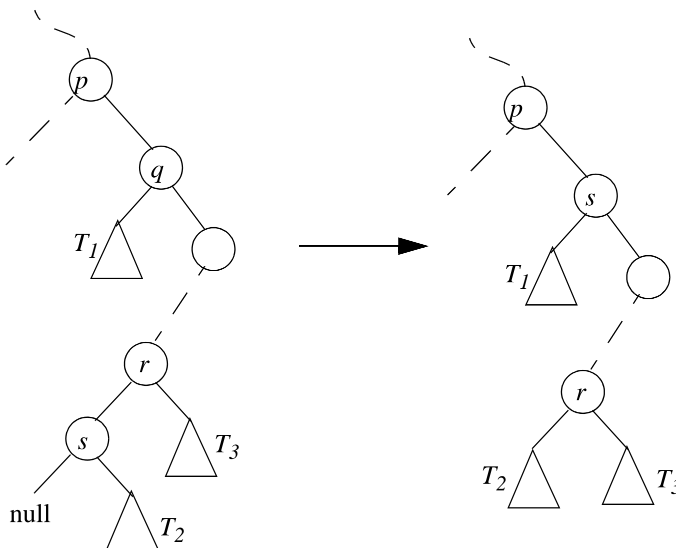
a) der Knoten q besitzt keinen Sohn



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



Fall 2:



Programmcode:
s. Methode *'deleteMin'*

Die folgende Methode *delete(x)* entfernt einen Schüssel *x* aus einem binären Suchbaum. Hierbei wird eine Hilfsmethode *findMin* verwendet, die den Knoten des kleinsten Schlüssels des rechten Teilbaumes (eines inneren Knotens) bestimmt und eine weitere Hilfsmethode *deleteMin*, die diesen Knoten entfernt:

```
// Methoden der Klasse BinarySearchTree
public void delete(int x) throws Exception
{ root = delete (x, root);}

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;
}

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;
}
```

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.

Laufzeitanalyse der Algorithmen *insert*, *find* und *delete*

Alle drei Methoden sind auf einen einzigen, bei der Wurzel beginnenden Pfad des Suchbaumes beschränkt.

➠ der **maximale Aufwand** ist damit $O(h)$, wobei h die Höhe des Baumes ist.

Für die Höhe binärer Bäume mit n Knoten gilt:

- Die **maximale Höhe** eines binären Baumes mit n Knoten ist n. (➠ Lineare Liste)
- Seine **minimale Höhe** ist $\lceil \log_2(n+1) \rceil$

Begründung: 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:

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

➠ $h_{min} = \lceil \log_2(n+1) \rceil$.

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

Eine aufwendige Durchschnittsanalyse ergibt unter den beiden Annahmen:

- der Baum ist nur durch Einfügungen entstanden und
- alle möglichen Permutationen der Eingabereihenfolge sind gleichwahrscheinlich

einen mittleren Wert $h_{\varnothing} = 2 \cdot \ln 2 \cdot \log n \approx 1{,}386 \cdot \log n$.

Der **durchschnittliche Zeitbedarf** für Suchen, Einfügen und Entfernen ist damit $O(\log n)$.

Kritikpunkt am naiven Algorithmus zum Aufbau binärer Suchbäume und damit an der Klasse so erzeugter binärer Suchbäume:

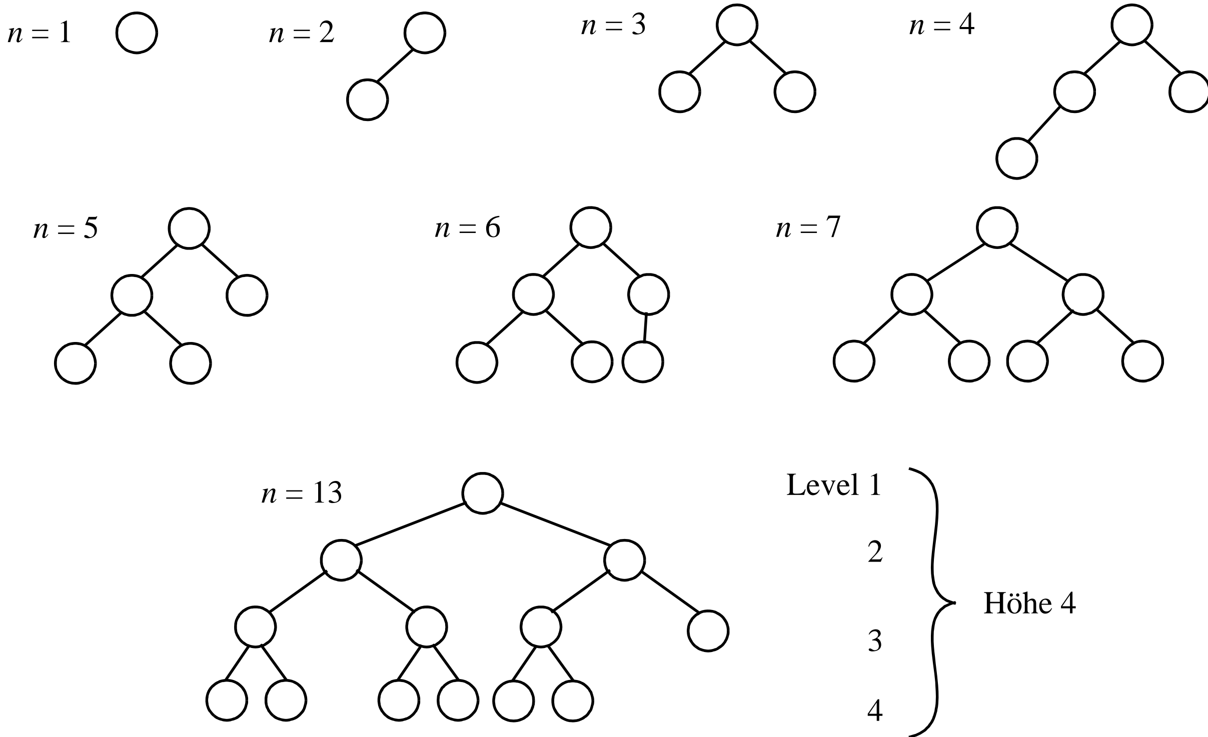
➠ im **worst-case** ist der Aufwand aller drei Operationen $O(n)$.

2.1.2 Vollständig ausgeglichene binäre Suchbäume

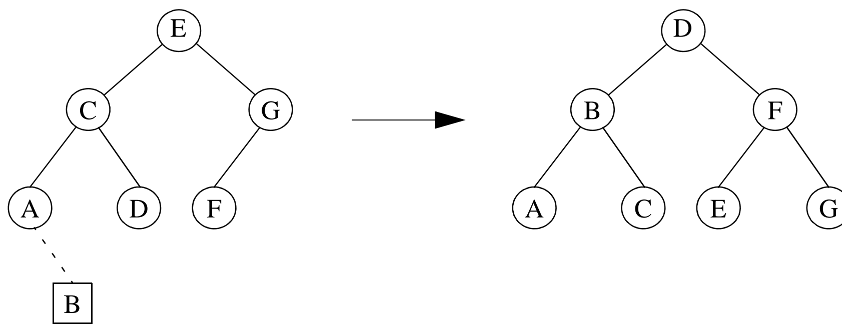
Die minimale Höhe $\lceil \log_2(n+1) \rceil$ unter allen binären Suchbäumen besitzen die **vollständig ausgeglichenen binären Suchbäume**, d.h. binäre Suchbäume, bei denen alle Levels bis auf das unterste vollständig besetzt sind.

➠ **optimaler Zeitaufwand für Suchoperationen**

Vollständig ausgeglichene binäre Bäume für verschiedene Knotenzahlen n:



Beispiel: Einfügen von Schlüssel 'B' in einen bestehenden Baum



Problem: Der Baum muß beim Einfügen von B **vollständig reorganisiert** werden.

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

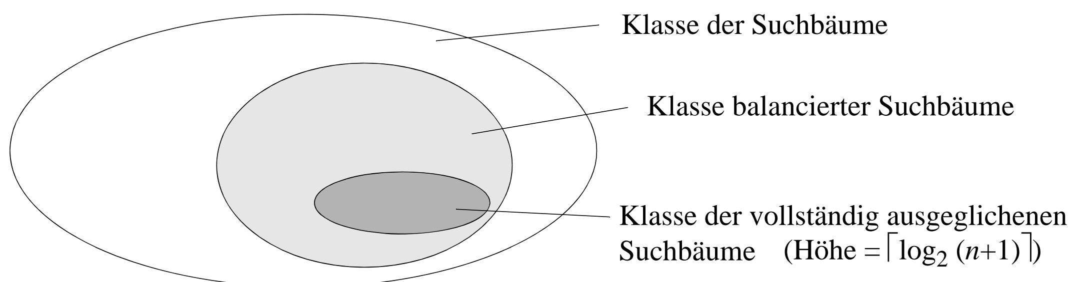
➡ Auswahl einer Kompromißlösung mit den Eigenschaften:

- die Höhe des Baumes ist im schlechtesten Fall $O(\log n)$.
- Reorganisationen bleiben auf den Suchpfad zum einzufügenden bzw.. zu entfernenden Schlüssel beschränkt und sind damit im schlechtesten Fall in $O(\log n)$ Zeit ausführbar.

Definition:

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

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



2.1.3 AVL-Bäume

(Adelson-Velskij und Landis (1962))

AVL-Bäume sind ein Beispiel für eine Klasse balancierter Suchbäume.

Definition:

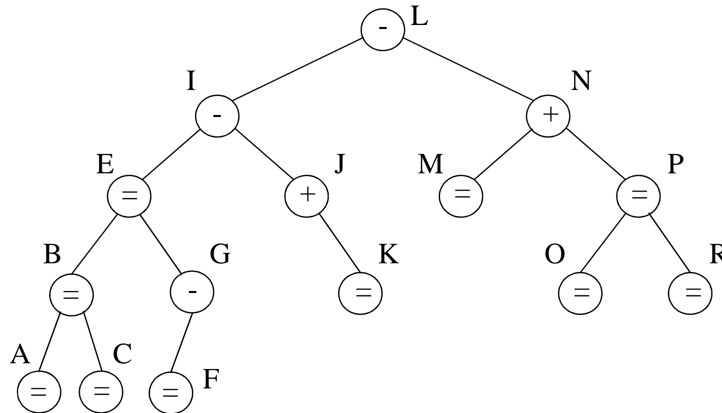
Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume T_r und T_l der Wurzel gilt:

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

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 (dargestellt durch -, = und +) annehmen.

Mögliche Strukturverletzungen durch Einfügungen bzw. Entfernungen von Schlüsseln erfordern **Rebalancierungsoperationen.**

Beispiel für einen AVL-Baum:

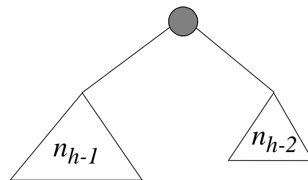


Behauptung: Die minimale Höhe $h_{min}(n)$ eines AVL-Baumes mit n Schlüsseln ist $\lceil \log_2(n+1) \rceil$. Dies folgt aus der Tatsache, daß ein AVL-Baum minimaler Höhe einem vollständig ausgeglichenen binären Suchbaum entspricht.

Behauptung: Die maximale Höhe $h_{max}(n)$ eines AVL-Baumes mit n Schlüsseln ist $O(\log n)$.

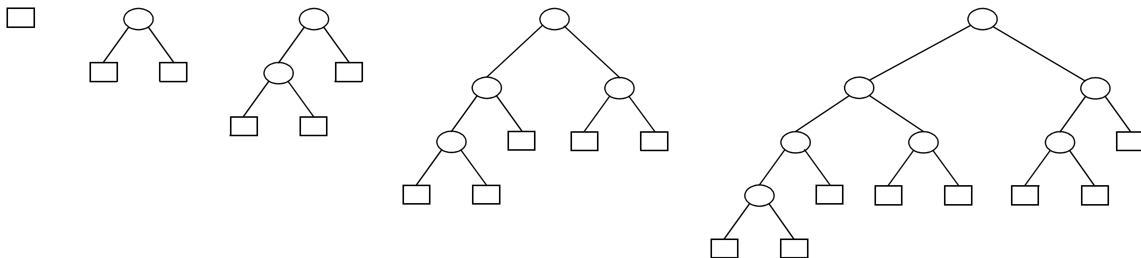
Beweis: Die maximale Höhe wird realisiert von sogenannten *minimalen AVL-Bäumen*. Dies sind AVL-Bäume, die für eine gegebene Höhe h die minimale Anzahl von Schlüsseln abspeichern.

Minimale AVL-Bäume haben bis auf Symmetrie die folgende Gestalt:



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

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



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

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

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

Beweis: Induktion über h

Induktionsanfang: $n_0 = fib(2) - 1 = 1 - 1 = 0$ ✓

Induktionsschluß: $n_{h+1} = n_h + n_{h-1} + 1$
$= 1 + fib(h+2) - 1 + fib(h+1) - 1$
$= fib(h+3) - 1$

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

Der Beweis erfolgt durch vollständige Induktion. *(wird hier übergangen)*

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

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

Mit obiger Hypothese folgt hieraus: $n + 1 \ge fib(h_{max}(n) + 2)$.

Durch Einsetzen erhalten wir:

$$n + 1 \ge \frac{1}{\sqrt{5}} \cdot \left(\varphi_1^{\,h_{max}(n)+2} - \underbrace{\varphi_2^{\,h_{max}(n)+2}}_{< 0{,}62 < \sqrt{5}/2} \right) \ge \frac{1}{\sqrt{5}} \cdot \varphi_1^{\,h_{max}(n)+2} - \frac{1}{2}$$

Und damit: $\frac{1}{\sqrt{5}} \cdot \varphi_1^{\,h_{max}(n)+2} \le n + \frac{3}{2}$.

Durch Auflösen nach $h_{max}(n)$ ergibt sich: $\log_{\varphi_1}(\frac{1}{\sqrt{5}}) + h_{max}(n) + 2 \le \log_{\varphi_1}(n + \frac{3}{2})$.

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

$$\le \log_{\varphi_1}(n) + const = \log_{\varphi_1}(2) \cdot \log_2(n) + const$$

$$\left(\log_b(a) = \frac{\log_c(a)}{\log_c(b)}\right) \quad = \frac{\ln 2}{\ln \varphi_1} \cdot \log_2(n) + const \approx 1{,}44 \cdot \log_2(n) + const$$

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

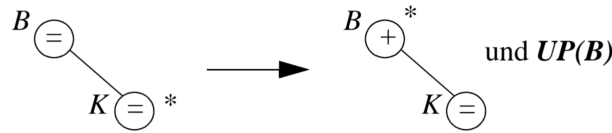
Also gilt die Behauptung: $h_{max}(n) = O(\log n)$.

Einfügen von Schlüsseln: ***Insert(k)***

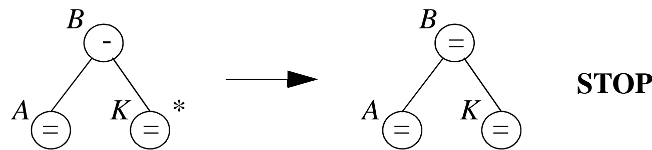
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

Der Schlüssel *k* wird in einen neuen Sohn *K* des Knotens *B* eingefügt.

Fall 1: *B* ist ein Blatt

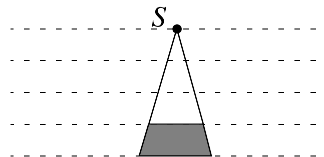


Fall 2: *B* hat einen linken Sohn



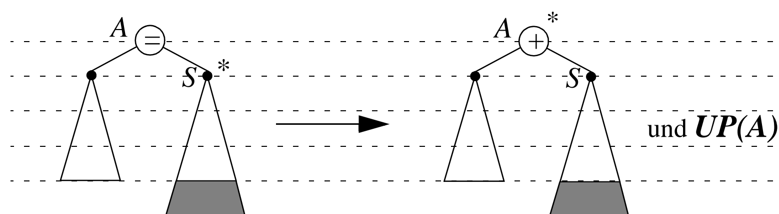
Die **Methode** ***UP(S)*** wird aufgerufen für einen Knoten *S*, dessen Teilbaum in seiner Höhe um 1 gewachsen ist. *S* ist die Wurzel eines korrekten AVL-Baumes.

➠ **mögliche Strukturverletzung** durch einen zu hohen Teilbaum!



Fall 1: der Vater von *S* hat BF '= '

1.1 der Vater von *S* ist nicht die Wurzel



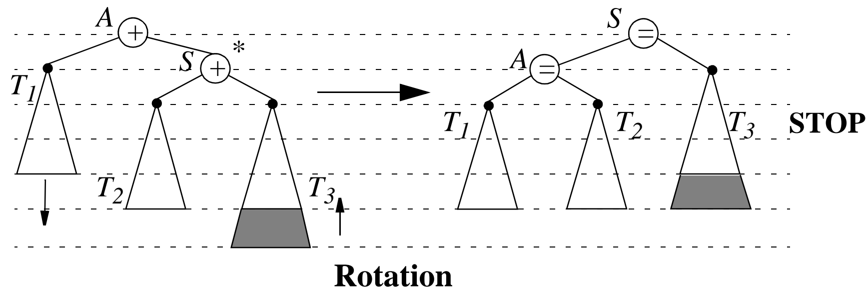
1.2 der Vater von *S* ist die Wurzel: ➙ dieselbe Transformation und **STOP**.

Fall 2: der 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**.

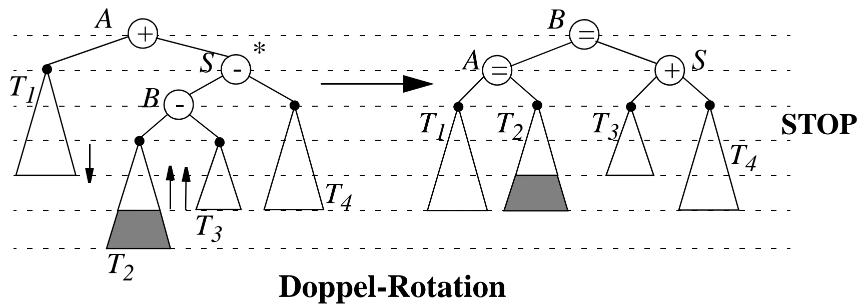
Fall 3: der Vater von *S* hat BF '+' oder '-' und *S* ist die Wurzel des höheren Teilbaumes.

3.1 der Vater von *S* hat BF '+' und *S* hat BF '+':



3.2 der Vater von *S* hat BF '+' und *S* hat BF '-'.

z.B.: *B* hat BF '-'.



der Fall *B* hat BF '+' wird analog gehandhabt.

3.3 der Vater von *S* hat BF '-' und *S* hat BF '-': ➞ symmetrisch zu 3.1

3.4 der 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. Wir werden sehen, daß dies beim Entfernen nicht genügt.

Entfernen von Schlüsseln: *Delete(k)*

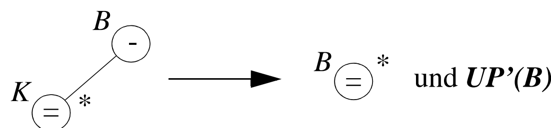
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

Der Knoten *K* mit Schlüssel *k* wird entfernt.

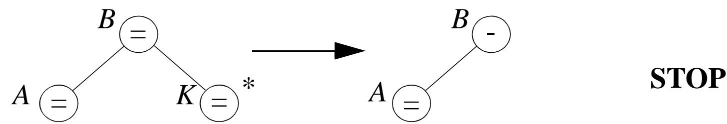
Fall 1: *K* hat höchstens einen Sohn

1.1 *K* ist ein Blatt

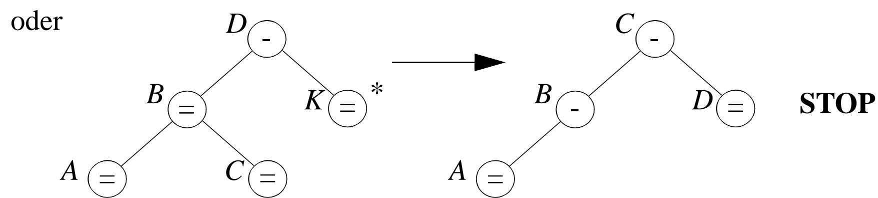
1.1.1 *K* hat keinen Bruder



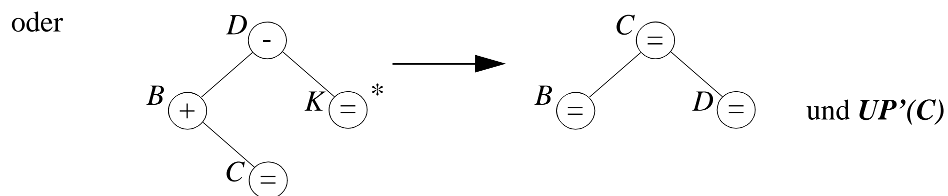
1.1.2 *K* hat einen Bruder



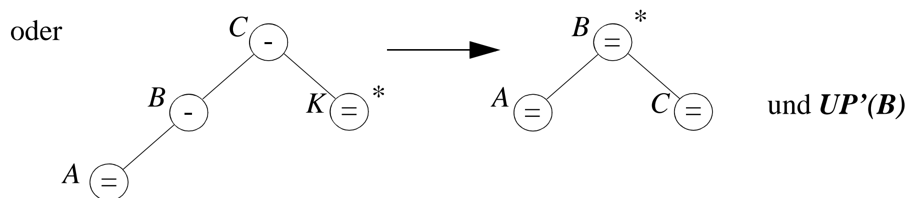
oder



oder

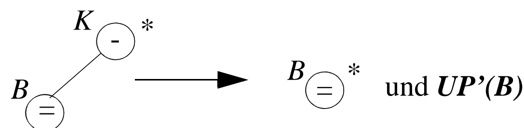


oder



1.2 *K* hat genau einen Sohn

1.2.1 *K* hat einen linken Sohn



1.2.2 *K* hat einen rechten Sohn: ➙ symmetrisch

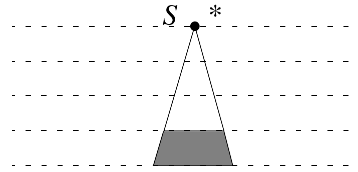
Fall 2: *K* ist ein innerer Knoten (hat zwei Söhne)

Man bestimme in dem AVL-Baum den **kleinsten** Schlüssel *s*, der **größer als *k*** ist. *s* ist in einem Halbblatt *S*. Ersetze *k* durch *s* und entferne den Schlüssel *s*.

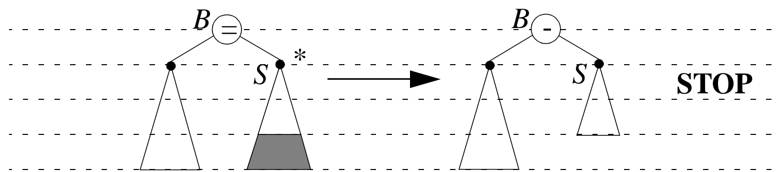
➠ damit haben wir Fall 2 auf Fall 1 zurückgeführt.

Die **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.

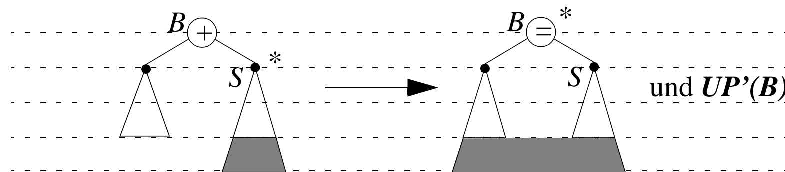
→ **mögliche Strukturverletzung** durch einen zu niedrigen Teilbaum!



Fall 1: der Vater von *S* hat BF '='.



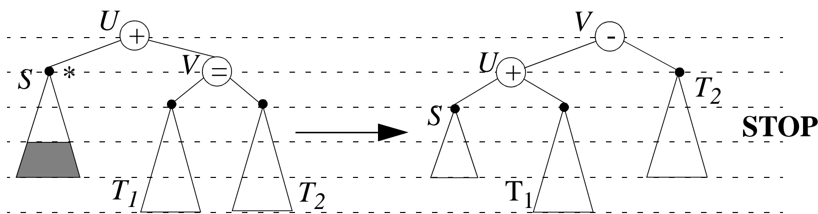
Fall 2: der Vater von *S* hat BF '+' oder '-' und *S* ist die Wurzel des höheren Teilbaums.



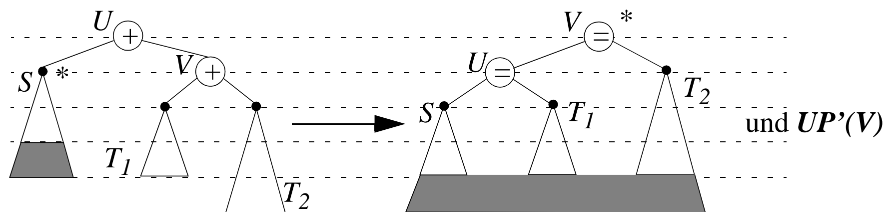
Fall 3: der Vater von *S* hat BF '+' oder '-' und *S* ist die Wurzel des kürzeren Teilbaums.

3.1 der Vater von *S* hat BF '+'

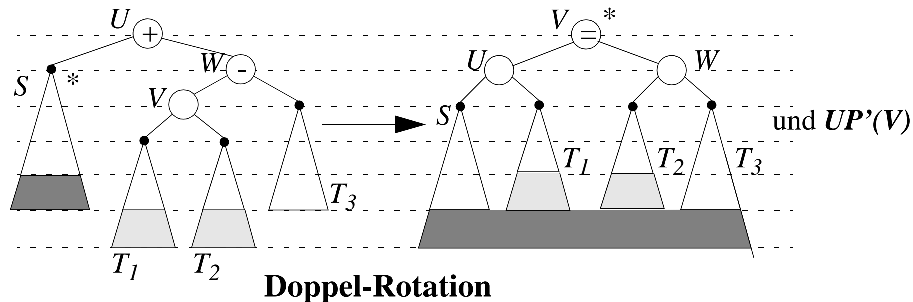
3.1.1 der Bruder von *S* hat BF '='



3.1.2 der Bruder von *S* hat BF'+'.



3.1.3 der Bruder von *S* hat BF'-'.



Doppel-Rotation

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

3.2 der Vater von *S* hat BF '-': ➠ symmetrisch zu 3.1.

Fall 4: *S* ist die Wurzel. ➠ **STOP**

Im Falle von Entferne-Operationen wird eine mögliche Strukturverletzung also 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.**

2.2 B-Bäume

Sei n die Anzahl der Objekte und damit der Datensätze. Wir nehmen nun an, daß das Datenvolumen zu groß ist, um im Hauptspeicher gehalten zu werden, z.B. $n = 10^6$.

➠ Datensätze auf externen Speicher auslagern, z.B. Plattenspeicher.

Beobachtung: Der Plattenspeicher wird als Menge von **Blöcken** (mit einer Größe im Bereich von 1 - 4 KByte) betrachtet, wobei ein Block durch das Betriebssystem jeweils komplett in den Hauptspeicher übertragen wird. Diese Übertragungseinheiten werden auch als **Seiten** des Plattenspeichers bezeichnet.

Idee: Konstruiere eine Baumstruktur mit der Eigenschaft:

1 Knoten des Baumes $\triangleq$ 1 Seite (bzw. mehreren Seiten) des Plattenspeichers

Für binäre Baumstrukturen bedeutet das:

left oder right-Zeiger folgen $\triangleq$ 1 Plattenspeicherzugriff