Kapitel 8 Suchverfahren

8.1 Laufzeitanalyse von Algorithmen und O-Notation

Die **Effizienz** ist ein wichtiges Kriterium zum Vergleich verschiedener Algorithmen zur Lösung ein und desselben Problems. Sie wird bestimmt durch den benötigten Aufwand des Algorithmus (seine **Komplexität**) in Abhängigkeit von einer speziellen Eingabesituation.

Wesentliche Effizienzkriterien sind

- die Laufzeit des Algorithmus
- der benötigte Speicherplatz

Häufig: 'trade-off' bei der Optimierung eines dieser beiden Kriterien dahingehend, daß das andere Kriterium verschlechtert wird.

In der Regel ist das wichtigere Kriterium die Laufzeit. Wir werden uns daher im folgenden auf die **Laufzeitanalyse** beschränken.

Laufzeitanalyse

1. Ansatz: Direktes **Messen der Laufzeit**, z.B. in Millisekunden.

➔ abhängig von vielen Parametern, wie Rechnerkonfiguration, Rechnerlast, Compiler, Betriebssystem, Programmiertricks, u.a., und damit sehr unzuverlässig und ungenau.

2. Ansatz: Zählen der benötigten **Elementaroperationen** des Algorithmus in Abhängigkeit von der Größe der Eingabe.

➔ das algorithmische Verhalten wird als Funktion der benötigten Elementaroperationen dargestellt. Die Charakterisierung dieser elementaren Operationen ist abhängig von der jeweiligen Problemstellung und dem zugrundeliegenden Algorithmus. Beispiele für Elementaroperationen sind Zuweisungen, Vergleiche, arithmetische Operationen, Zeigerdereferenzierungen oder Arrayzugriffe.

➔ das Maß für die Größe der Eingabe ist abhängig von der Problemstellung, z.B.

Problem	Größe der Eingabe
Suche eines Elementes in einer Liste	Anzahl der Elemente
Multiplikation zweier Matrizen	Dimension der Matrizen
Sortierung einer Liste von Zahlen	Anzahl der Zahlen

Betrachten wir nochmals die Laufzeit des binären Suchens. Für die Anzahl der Vergleiche im schlechtesten Fall hatten wir ermittelt: $A_{worst}(n) = \lfloor \log_2(n) \rfloor + 1$.

Durch Weglassen multiplikativer und additiver Konstanten kommen wir von der Anzahl der Schlüsselvergleiche zur Laufzeitfunktion. Damit erhält man eine von der Programmumgebung und anderen äußeren Einflußgrößen (z.B.: dem *Zeitbedarf einer Vergleichsoperation, der Effizienz der Programmierung,* ...) unabhängige Charakterisierung der (asymptotischen) Komplexität des Algorithmus.

➔**O-Notation verwenden**

Definition: *O-Notation*

Seien $f:(N \to \Re^+)$ und $g:(N \to \Re^+)$.

$f = O(g) \Leftrightarrow \exists\ n_0 \in N,\quad c \in \Re^+$ mit $\forall\ n \geq n_0$ ist $f(n) \leq c \cdot g(n)$.

Man sagt auch: f wächst höchstens so schnell wie g.

Anmerkung: Da $O(g)$ genau genommen eine (unendliche) Menge von Funktionen beschreibt, ist es genauer, zu sagen: $f \in O(g)$. In der Literatur hat sich jedoch das '=' eingebürgert. Wichtig ist es, darauf zu achten, daß insbesondere die Kommutativität des '='-Operators hier **nicht** gilt.

Mit dieser Notation gilt: $T_{worst}(n) = O(A_{worst}(n))$.

Im Beispiel des binären Suchens also:

$$T_{worst}(n) = O(\lfloor \log_2(n) \rfloor + 1) = O(\log n).$$

Funktionen, die bei der Laufzeitanalyse von Algorithmen realistischerweise auftauchen, sind meist monoton wachsend und von 0 verschieden. Zur Überprüfung obiger Eigenschaft betrachtet man dabei den Quotienten $f(n)/g(n)$.

Nach Definition gilt für $f = O(g)$: $f(n)/g(n) \leq c$ für $n \geq n_0$.

Man betrachte den Grenzwert: $\lim_{n \to \infty} \frac{f(n)}{g(n)}$. Existiert dieser (d.h. er ist $< \infty$), so ist $f = O(g)$.

Rechnen mit der O-Notation:

- Elimination von Konstanten: $2 \cdot n = O(n)$, $n/2 + 1 = O(n)$
- Bilden oberer Schranken: $2 \cdot n = O(n^2)$, $3 = O(\log n)$

Wichtige Klassen von Funktionen:

	Sprechweise	Typische Algorithmen
$O(1)$	konstant	
$O(\log n)$	logarithmisch	Suchen auf einer Menge
$O(n)$	linear	Bearbeiten jedes Elementes einer Menge
$O(n \cdot \log n)$		Gute Sortierverfahren, z.B. Heapsort
$O(n \cdot \log^2 n)$		
...		
$O(n^2)$	quadratisch	primitive Sortierverfahren
$O(n^k), k \geq 2$	polynomiell	
...		
$O(2^n)$	exponentiell	Backtracking-Algorithmen

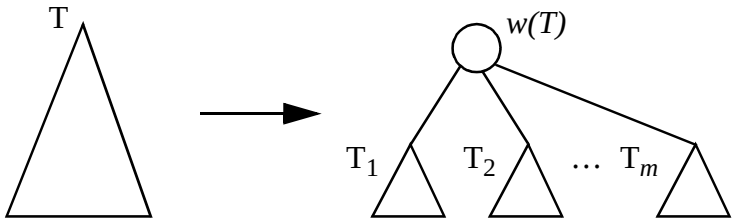
Nach diesem Exkurs in die O-Notation wollen wir uns nochmals Bäumen zuwenden, diesmal nicht beschränkt auf binäre Bäume.

8.2 Baumstrukturen

Ein **Baum** ist eine endliche Menge T von Elementen, **Knoten** genannt, mit:

(1) Es gibt einen ausgezeichneten Knoten $w(T)$, die **Wurzel** von T

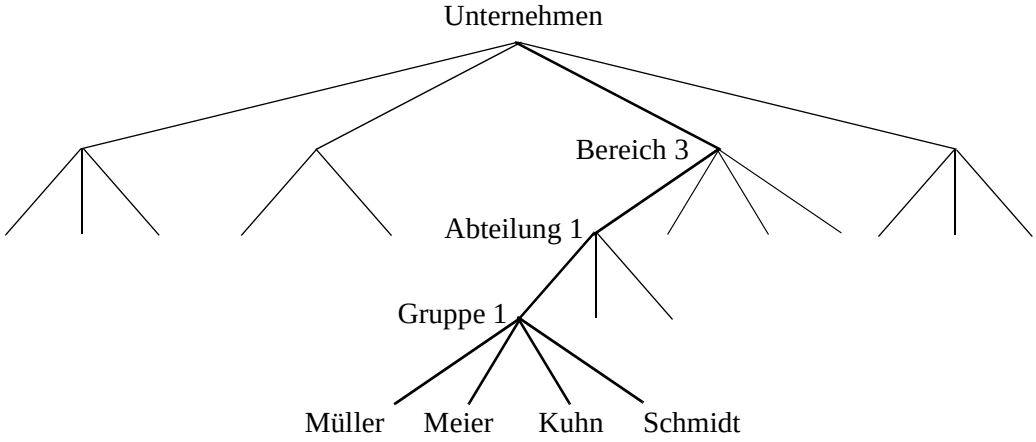
(2) Die restlichen Knoten sind in $m \geq 0$ disjunkte Mengen $T_1, \ldots, T_m$ zerlegt, die ihrerseits Bäume sind. $T_1, \ldots, T_m$ heißen **Teilbäume** der Wurzel $w(T)$. (*rekursive Definition*)



Eine **lineare Liste** ist ein (entarteter) Baum, in dem jeder Knoten höchstens einen Teilbaum besitzt.

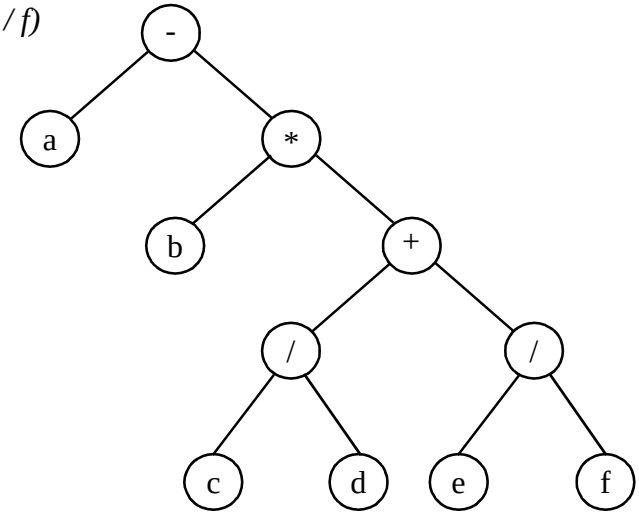
Bäume erlauben es, **hierarchische Beziehungen zwischen Objekten** darzustellen. Derartige Hierarchien treten vielfach in der realen Welt auf, z.B.:

- Die Strukturierung eines Unternehmens in Bereiche, Abteilungen, Gruppen und Angestellte



- Die Gliederung eines Buches in Kapitel, Abschnitte, Unterabschnitte
- Die Aufteilung Deutschlands in Bundesländer, Bezirke, Kreise, Gemeinden
- Darstellung (geklammerter) arithmetischer Ausdrücke als **binäre Bäume** (d.h. $m \leq 2$ für alle Knoten des Baumes). Diese Bäume heißen **Operatorbäume**.

Operatorbaum für: *a - b * (c / d + e / f)*



Der **Grad** eines Knotens *x*, ***deg(x)***, ist gleich der Anzahl der Teilbäume von *x*. Gilt $deg(x) = 0$, so nennt man *x* ein **Blatt**.

Der **Grad des Baumes** ist der maximale Grad aller seiner Knoten.

Jeder Knoten $x \neq w(T)$ hat einen eindeutigen Vorgänger *v(x)*, auch **Vater** von *x* genannt. *s(x)* bezeichnet die Menge aller **Söhne** (auch Kinder oder Nachfolger von *x* genannt) und *b(x)* die Menge aller **Brüder** von *x* (auch Geschwister von *x* genannt). Alle Brüder in *b(x)* haben denselben Vater wie *x*.

Ein **Pfad** in einem Baum ist eine Folge von Knoten $p_1, \ldots, p_n$ mit: $p_i = v(p_{i+1})$, $i = 1, \ldots, n\text{-}1$. Die **Länge des Pfades** ist *n*.

Der **Level** eines Knotens *x*, ***lev(x)*** ist: $lev(x) = \begin{cases} 1 & \text{für } x = w(T) \\ lev(v(x)) + 1 & \text{für } x \neq w(T) \end{cases}$.

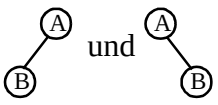
Damit ist $lev(x)$ gleich der Anzahl der Knoten auf dem Pfad von der Wurzel zum Knoten *x*.

Die **Höhe** eines Baumes ist gleich dem maximalen Level seiner Knoten. Dies entspricht der Länge des längsten von der Wurzel ausgehenden Pfades innerhalb des Baumes.

Ein **Wald** ist eine geordnete Menge von $n \geq 0$ disjunkten Bäumen. Entfernt man die Wurzel eines Baumes, so erhält man einen Wald.

Ein **binärer Baum** ist eine endliche Menge B von Knoten, die

- entweder leer ist
- oder aus einer Wurzel und zwei disjunkten binären Bäumen besteht, dem linken und dem rechten Teilbaum der Wurzel.

Damit sind: 

und zwei verschiedene binäre Bäume, aber als Bäume gleich.

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*)).

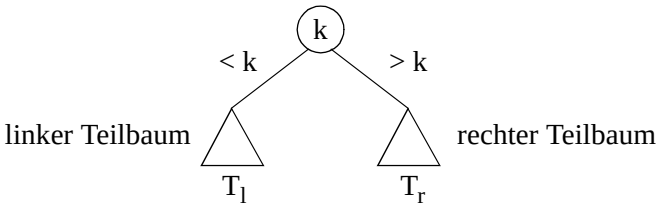
8.3 Binäre Suchbäume

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:

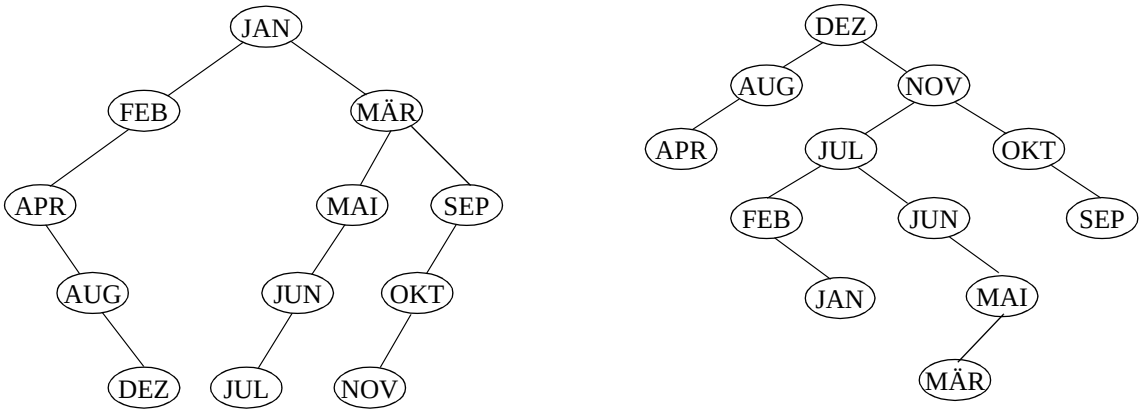


Anmerkungen:

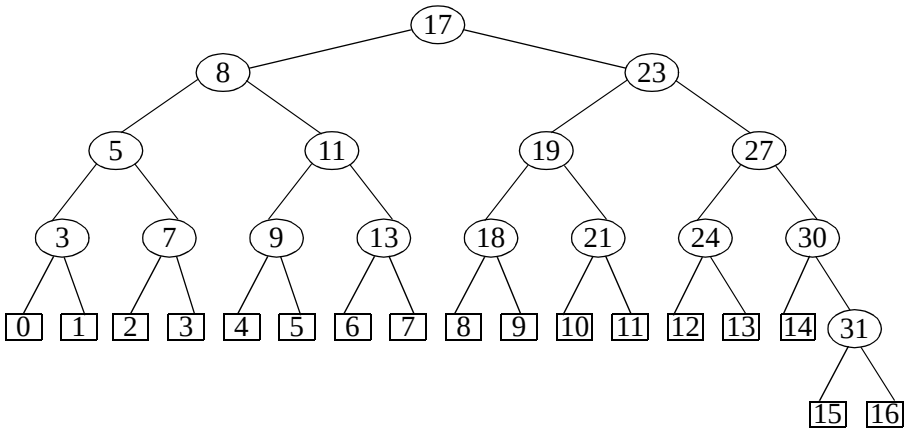
- Zur Organisation einer gegebenen Menge von *n* Schlüsseln gibt es eine große Anzahl unterschiedlicher binärer Suchbäume.
- Der *inorder*-Durchlauf eines binären Suchbaumes generiert die (aufsteigend) sortierte Folge der gespeicherten Schlüssel.

Beispiele:

Zwei verschiedene binäre Suchbäume über den Monatsnamen:



Der Entscheidungsbaum zur binären Suche ist ein binärer Suchbaum:



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

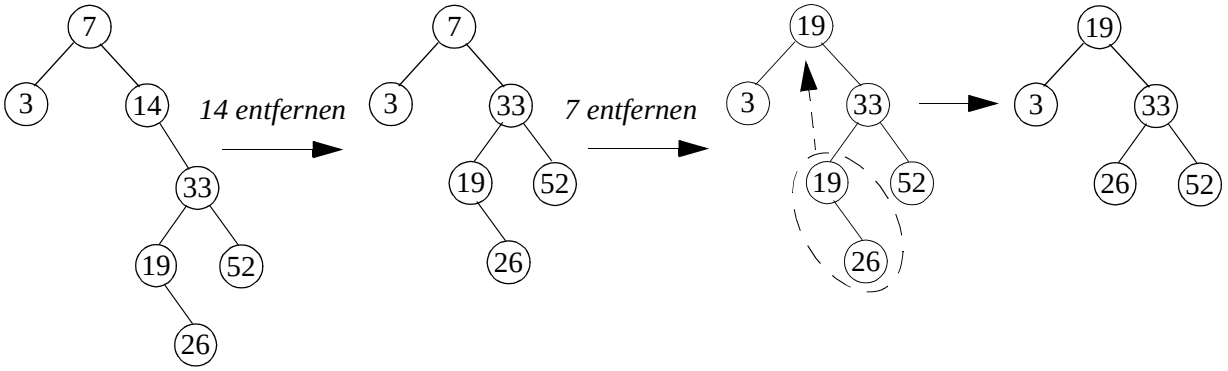
Die folgende Methode *find (x)* sucht in einem binären Suchbaum den Schlüssel *x* und liefert diesen zurück, falls er gefunden wurde. Andernfalls wird eine Ausnahmebehandlung durchgeführt.

```
// 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;       // Match
     }
 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:

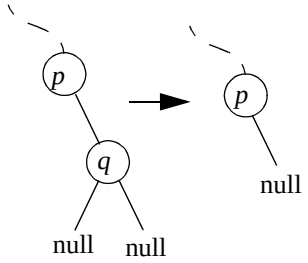


Allgemein treten zwei grundsätzlich verschiedene Situationen auf:

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

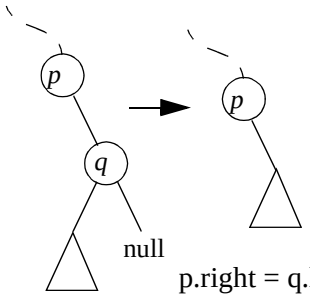
Fall 2: der Knoten *q* mit dem zu entfernenden Schlüssel besitzt zwei Söhne (innerer Knoten)

Fall 1: a) der Knoten *q* besitzt keinen Sohn



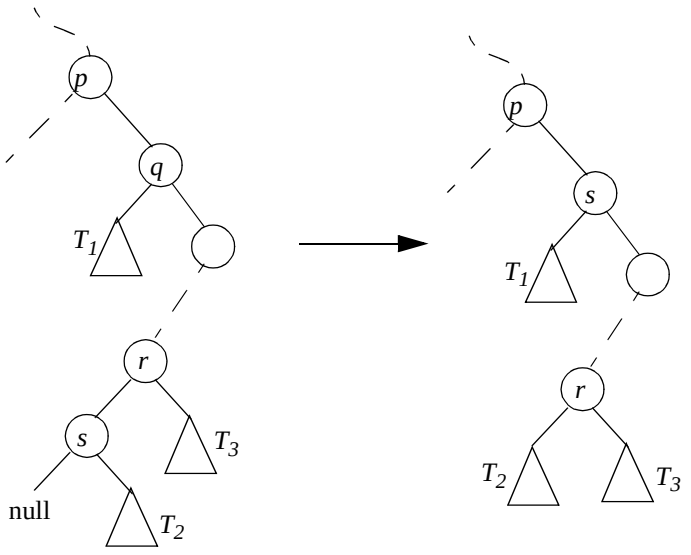
p.right = **null**;

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



p.right = q.left;

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:

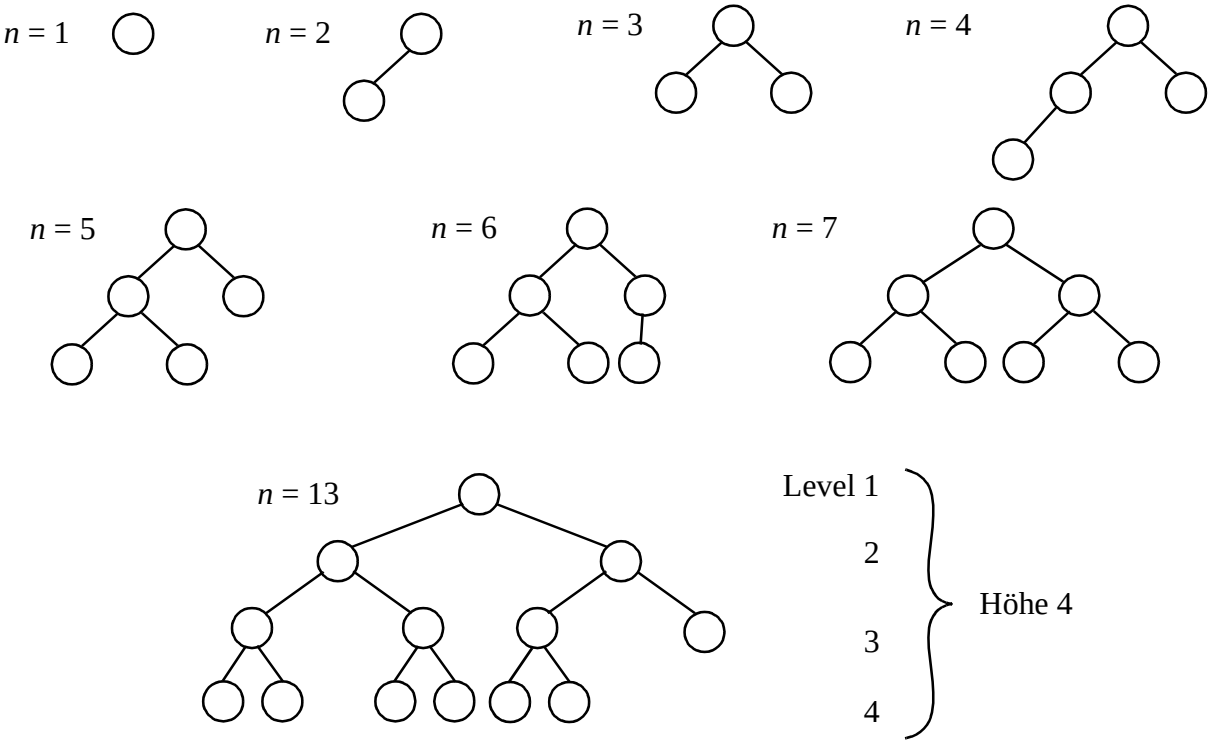
→ im **worst-case** ist der Aufwand aller drei Operationen $O(n)$.

8.3.2 Vollständig ausgeglichene binäre Suchbäume

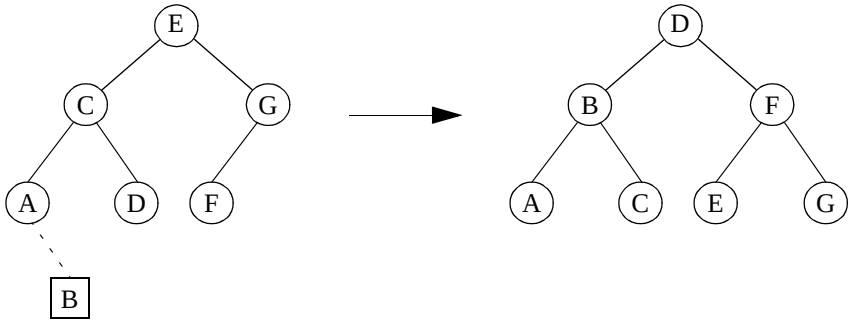
Die minimale Höhe $\lceil \log_2(n+1) \rceil$ unter allen binären Suchbäumen besitzen die **vollständig ausgeglichenen binären Suchbäume**, d.h. binäre Suchbäume, bei denen alle Levels bis auf das unterste vollständig besetzt sind.

→ **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:** $O(n)$ **.**

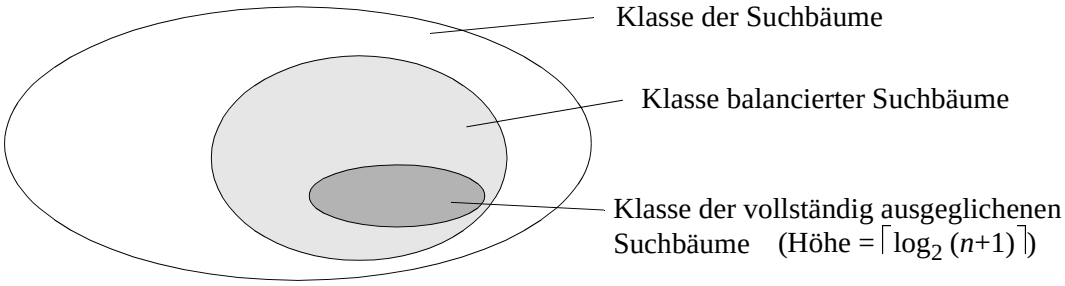
➔ Auswahl einer Kompromißlösung mit den Eigenschaften:

- 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.



8.3.3 AVL-Bäume

(Adelson-Velskij und Landis (1962))

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

Definition:

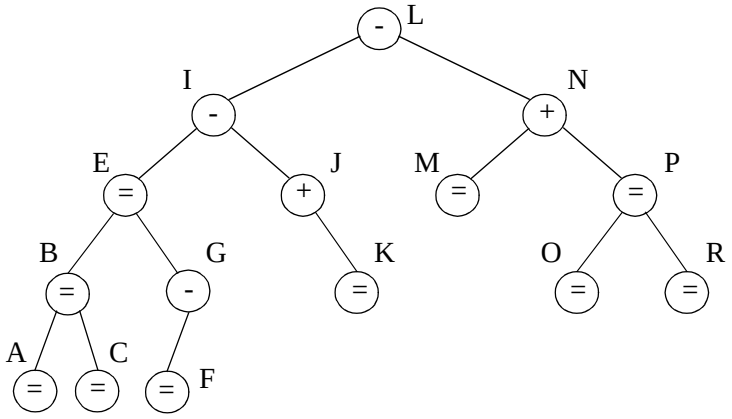
Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume T_r und T_l der Wurzel gilt:

- $|h(T_r) - h(T_l)| \leq 1$
- 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:

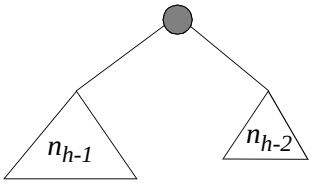


Behauptung: Die minimale Höhe $h_{min}(n)$ eines AVL-Baumes mit *n* Schlüsseln ist $\lceil \log_2(n+1) \rceil$. Dies folgt aus der Tatsache, daß ein AVL-Baum minimaler Höhe einem vollständig ausgeglichenen binären Suchbaum entspricht.

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

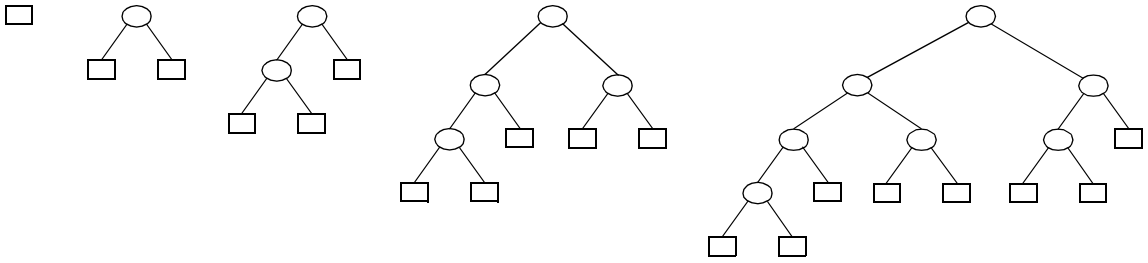
Beweis: Die maximale Höhe wird realisiert von sogenannten *minimalen AVL-Bäumen*. Dies sind AVL-Bäume, die für eine gegebene Höhe h die minimale Anzahl von Schlüsseln abspeichern.

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

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

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

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

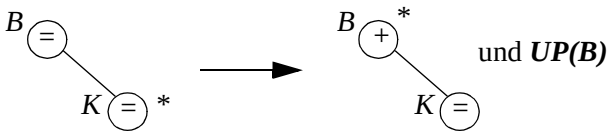
Also gilt die Behauptung: $h_{max}(n) = O(\log n)$.

Einfügen von Schlüsseln: *Insert(k)*

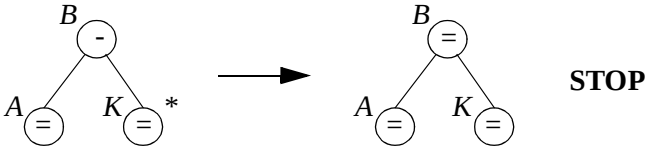
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

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

Fall 1: *B* ist ein Blatt

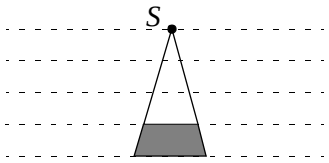


Fall 2: *B* hat einen linken Sohn



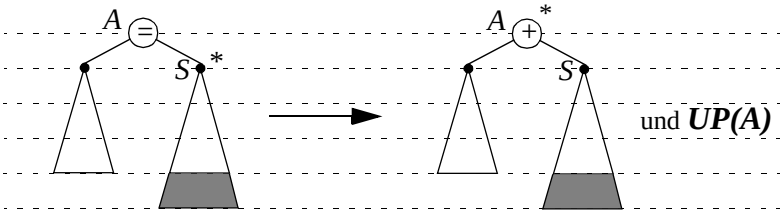
Die **Methode *UP(S)*** wird aufgerufen für einen Knoten *S*, dessen Teilbaum in seiner Höhe um 1 gewachsen ist. *S* ist die Wurzel eines korrekten AVL-Baumes.

➔**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



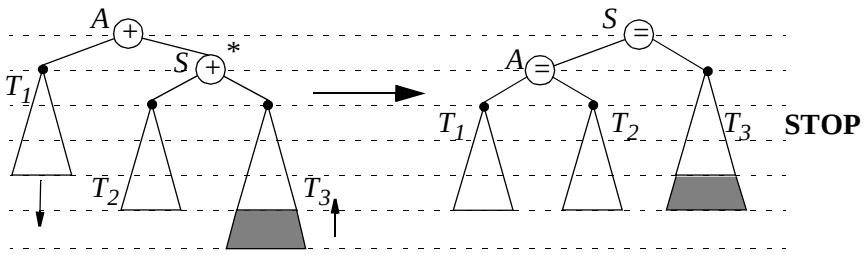
1.2 der Vater von *S* ist die Wurzel: ➔dieselbe Transformation und **STOP**.

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.

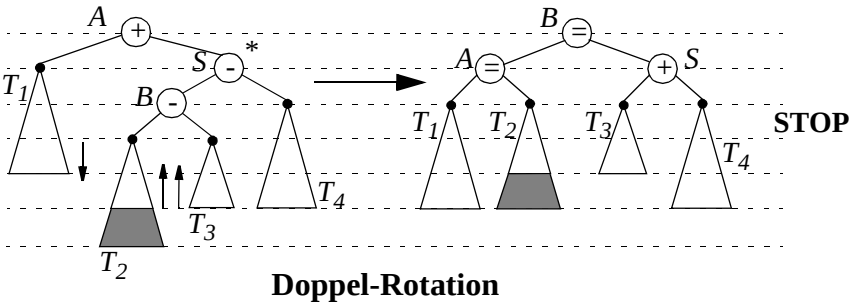
3.1 der Vater von *S* hat BF '+' und *S* hat BF '+':



Rotation

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

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



Doppel-Rotation

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)*

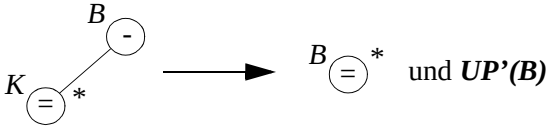
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

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

Fall 1: *K* hat höchstens einen Sohn

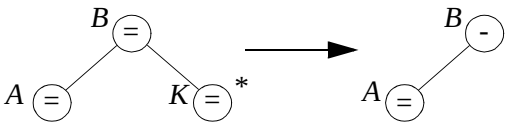
1.1 *K* ist ein Blatt

1.1.1 *K* hat keinen Bruder



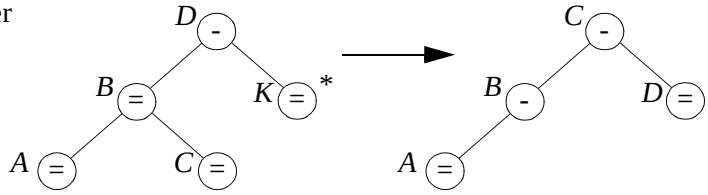
und ***UP'(B)***

1.1.2 *K* hat einen Bruder



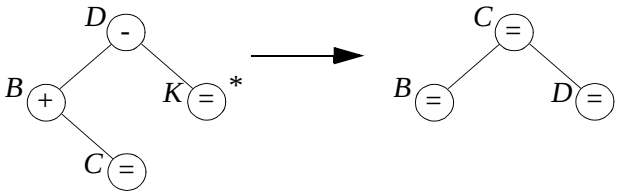
STOP

oder



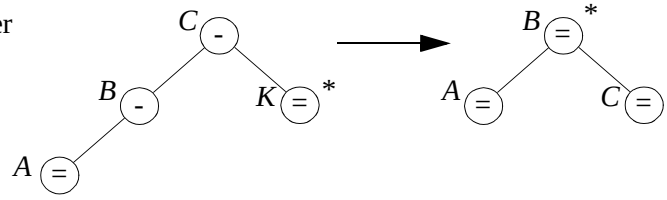
STOP

oder



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

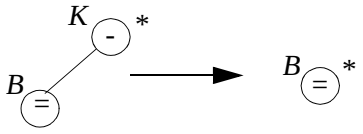
oder



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

1.2 *K* hat genau einen Sohn

1.2.1 *K* hat einen linken Sohn



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

1.2.2 *K* hat einen rechten Sohn: ➔symmetrisch

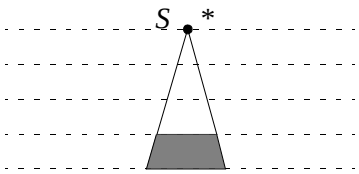
Fall 2: *K* ist ein innerer Knoten (hat zwei Söhne)

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*.

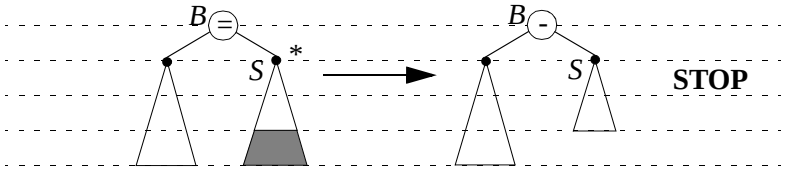
➔damit haben wir Fall 2 auf Fall 1 zurückgeführt.

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.

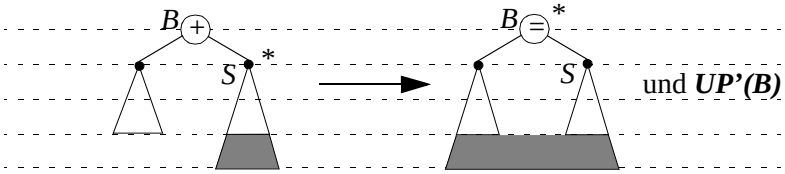
➔**mögliche Strukturverletzung** durch einen zu niedrigen Teilbaum!



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



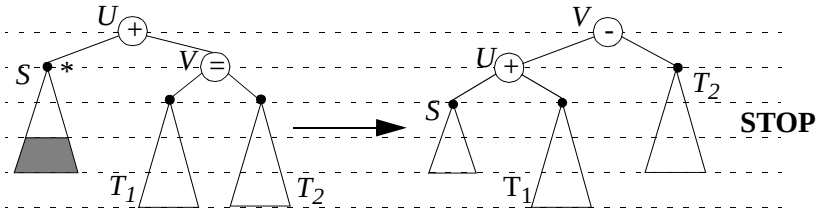
Fall 2: der Vater von *S* hat BF '+' oder '-' und *S* ist die Wurzel des höheren Teilbaums.



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

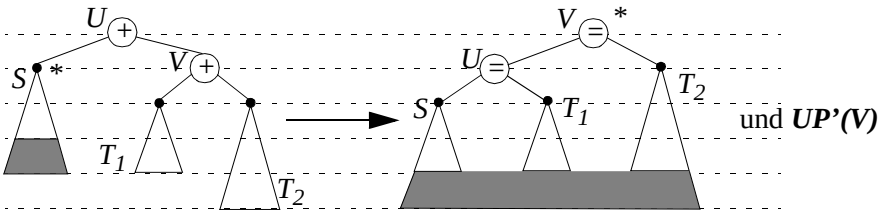
3.1 der Vater von *S* hat BF '+'

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



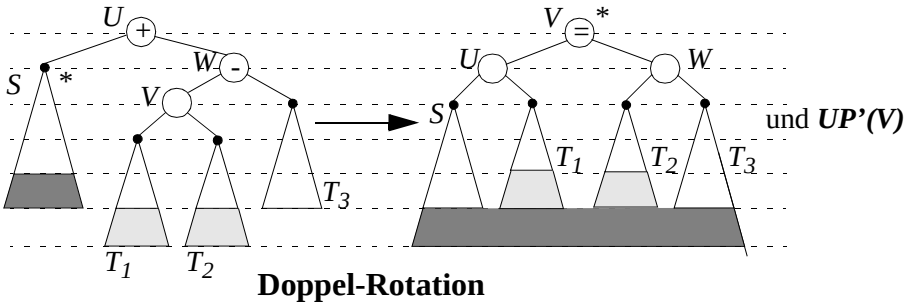
Rotation

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



Rotation

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.**

8.4 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:

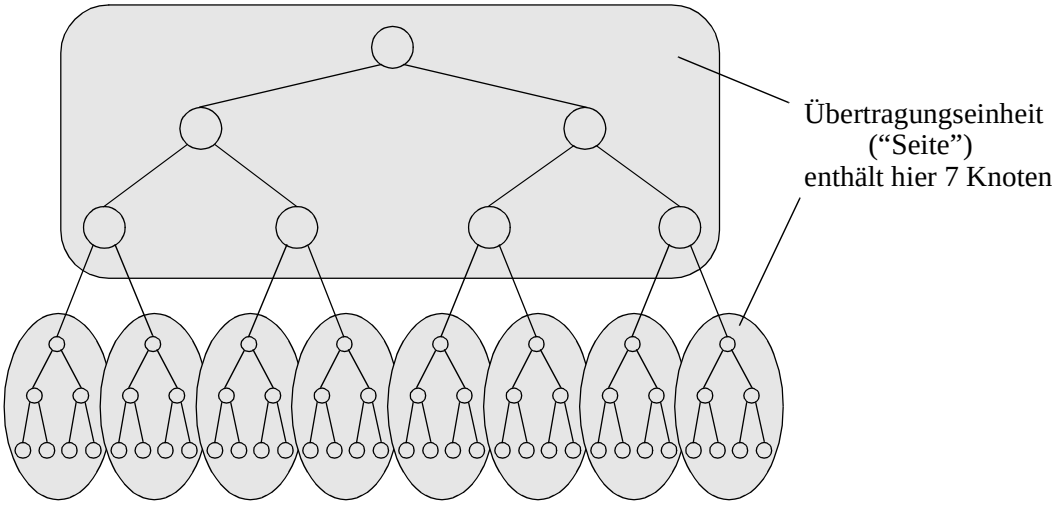
left oder right-Zeiger folgen $\triangleq$ 1 Plattenspeicherzugriff

Beispiel:

Sei die Anzahl der Datensätze: $n = 10^6$

→ $\log_2(10^6) = \log_2(10^3)^2 = 2 \cdot \log_2(10^3) \approx 20$ Plattenspeicherzugriffe

Idee: Zusammenfasssen mehrerer binärer Knoten zu einer Seite



Beispiel für einen B-Baum:

99 Knoten in einer Seite → 100-fache Verzweigung:

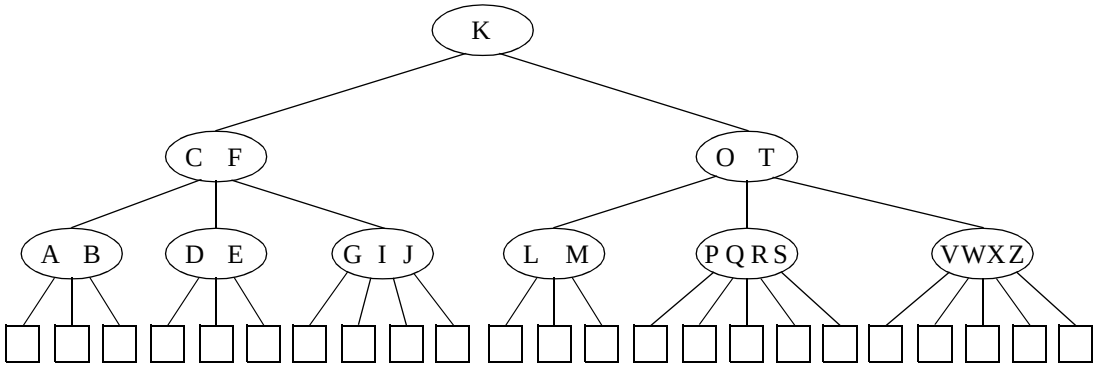
→ $\log_{100}(10^6) = 3$ Plattenspeicherzugriffe.

(reduziert auf 2, falls die Wurzelseite immer im Hauptspeicher liegt)

Definition: B-Baum der Ordnung *m* (Bayer und McCreight (1972))

(1) Jeder Knoten enthält höchstens 2*m* Schlüssel.
(2) Jeder Knoten außer der Wurzel enthält mindestens *m* Schlüssel.
(3) Die Wurzel enthält mindestens einen Schlüssel.
(4) Ein Knoten mit *k* Schlüsseln hat genau *k*+1 Söhne.
(5) Alle Blätter befinden sich auf demselben Level.

Beispiel: für einen B-Baum der Ordnung 2



Berechnung der maximalen Höhe h_{max} eines B-Baumes der Ordnung m mit n Schlüsseln:

Level 1 hat $k_1 = 1$ Knoten

Level 2 hat $k_2 \geq 2$ Knoten

Level 3 hat $k_3 \geq 2(m+1)$ Knoten

. . . .

Level h+1 hat $k_{h+1} \geq 2(m+1)^{h-1}$ (äußere, leere) Knoten

Ein B-Baum mit n Schlüsseln teilt den Wertebereich der Schlüssel in $n + 1$ Intervalle. Es gilt also

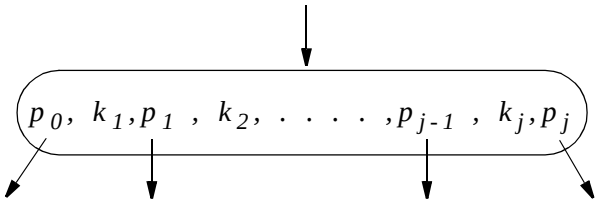
$$k_{h+1} = n + 1 \geq 2(m+1)^{h-1}, \text{ d.h. } h \leq 1 + \log_{m+1}\left(\frac{n+1}{2}\right).$$

Da die Höhe immer ganzzahlig ist, und da diese Rechnung minimalen Füllgrad annimt, folgt:

$$h \leq \left\lfloor \log_{m+1}\left(\frac{n+1}{2}\right) \right\rfloor + 1$$

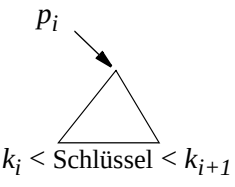
Beobachtung: Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt. Die Speicherplatzausnutzung beträgt also mindestens 50 %!

Allgemeine Knotenstruktur:



wobei: $k_1 < k_2 < \ldots < k_j$ und $m \leq j \leq 2m$;

p_i zeigt auf den Teilbaum mit Schlüsseln zwischen k_i und k_{i+1}.



Anmerkung: Da die Schlüssel in jedem Knoten eines B-Baumes aufsteigend sortiert sind, kann ein Knoten nach Übertragung in den Hauptspeicher binär durchsucht werden.

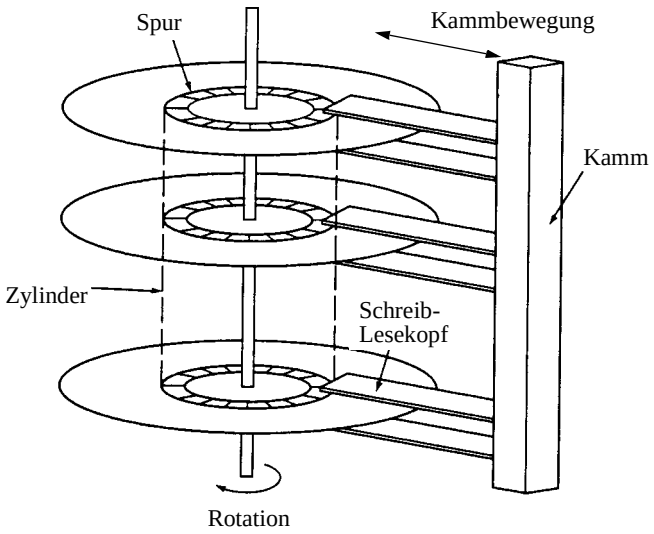
```
class Entry { public int key;
              public Page son;
              ... // Konstruktor..
              }

class Page { public int numberOfEntries;
             public Page firstSon;
             public Entry [] entries;
                    // Im Konstruktor intialisieren mit new Entry [2*m+1];
             ...
             }

class Btree { protected Page root;
              protected int m;
              ...
              }
```

Welche Ordnung *m* von B-Bäumen ist auf realen Rechnern und Plattenspeichern günstig?

Hierzu betrachten wir zunächst den physischen Aufbau eines Magnetplattenspeichers. Dieser besteht aus einer Reihe übereinanderliegender rotierender Magnetplatten, die in Zylinder, Spuren und Sektoren unterteilt sind. Der Zugriff erfolgt über einen Kamm mit Schreib-/Leseköpfen, der quer zur Rotation bewegt wird.



Der Seitenzugriff erfolgt nun in mehreren Phasen. Etwas vereinfacht läßt sich die Zugriffszeit in die Zeiten für die folgenden Phasen zerlegen:

- *Positionierung des Schreib-/Lesekopfes: Positionierungszeit (PZ)*
  Zeit für die Kammbewegung 6 ms
- *Warten auf den Sektor / die Seite: Latenzzeit (LZ)*
  Im Mittel halbe Rotationszeit der Platte 4 ms
- *Übertragung der Seite: Übertragungszeit (ÜZ)*
  Zeit für das Schreiben / Lesen $1 \cdot 10^{-4}$ ms / Byte

→ **Zugriffszeit für eine Seite**: PZ + LZ + ÜZ · (Seitengröße).

Sei die Größe eines Schlüssels durch α Bytes und die eines Zeigers durch β Bytes gegeben.

→ **Seitengröße** $\approx 2m(\alpha + \beta)$

→ **Zugriffszeit** pro Seite = PZ + LZ + ÜZ · $\approx 2m(\alpha + \beta) = a + b \cdot m$

mit: a = PZ + LZ und $b = 2(\alpha+\beta) \cdot$ ÜZ.

Andererseits ergibt sich für die **interne Verarbeitungszeit** pro Seite bei binärer Suche:

$c \cdot \log_2(m) + d$ für Konstanten c und d.

Die **gesamte Verarbeitungszeit pro Seite** ist damit: $a + b \cdot m + c \cdot \log_2(m) + d$.

Die maximale Anzahl von Seiten auf einem Suchpfad eines B-Baumes mit n Schlüsseln ist:

$$h_{max} = \left\lfloor \log_{m+1}\left(\frac{n+1}{2}\right) \right\rfloor + 1 = f \cdot \frac{\log_2\left(\frac{n+1}{2}\right)}{\log_2(m)} \quad \text{für eine Konstante } f.$$

Die **maximale Suchzeit** MS ist damit gegeben durch die Funktion:

$$MS(m) = g \cdot \left(\frac{a+d}{\log_2(m)} + \frac{b \cdot m}{\log_2(m)} + c\right) \quad \text{mit } g = f \cdot \log_2\left(\frac{n+1}{2}\right).$$

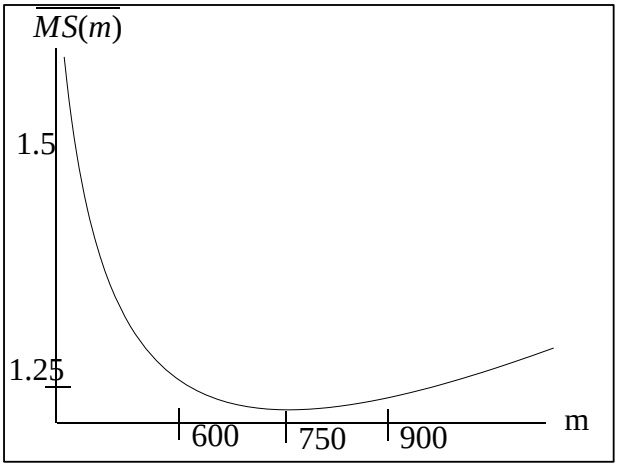
Für die obigen Konstanten setzen wir beispielhaft die folgenden Werte ein:

$a = 0{,}01s,\ a + d \approx a = 0{,}01s$. $= 10\ ms$

$\alpha = 8, \beta = 4 \rightarrow b = 24 \cdot 1 \cdot 10^{-4} ms = 24 \cdot 10^{-4} ms$.

Damit ist die folgende Funktion $\overline{MS(m)}$ zu minimieren:

$$\overline{MS(m)} = \left(\frac{10}{\log_2(m)} + \frac{0{,}0024 \cdot m}{\log_2(m)}\right) ms \quad \rightarrow \textbf{MIN}.$$



Die maximale Suchzeit ist somit nahezu minimal für $600 \le m \le 900$. Dies entpricht in obigem Beispiel einer “optimalen” Seitengröße von 12 KByte.

Einfügen in B-Bäumen:

Zunächst wird der Knoten *K* gesucht, in den der neue Schlüssel einzufügen ist. Dieser ist stets ein Blatt. Der Schlüssel wird in *K* an der entsprechenden Stelle eingefügt.

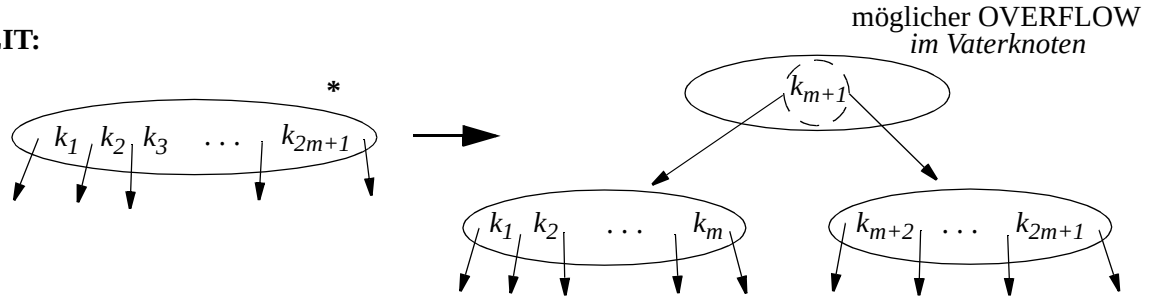
Sei *s* die Anzahl der Schlüsel in *K* nach dem Einfügen:

Fall 1: $s \le 2m$: **→STOP**

Fall 2: $s = 2m + 1$: **→OVERFLOW**

Ein **OVERFLOW** wird behandelt durch Aufspalten (**SPLIT**) des Knotens.
Dies kann einen OVERFLOW im Vaterknoten zur Folge haben. Auf diese Weise kann sich ein OVERFLOW bis zur Wurzel des Baumes fortsetzen.
Falls die Wurzel gesplittet wird wächst die Höhe des Baumes um 1.

SPLIT:



Entfernen aus B-Bäumen:

Der zu entfernende Schlüssel *k* wird im Baum gesucht und aus dem gefundenen Knoten *K* gelöscht. Falls *K* kein Blatt ist, wird der entfernte Schlüssel durch den Schlüssel *p* ersetzt, der der kleinste Schlüssel im Baum ist, der größer als *k* ist. Sei *P* der Knoten, in dem *p* liegt. Dann ist *P* ein Blatt, aus dem *p* nun entfernt wird. Auf diese Weise wird der Fall "Löschen in einem inneren Knoten" auf den Fall "Löschen in einem Blatt" zurückgeführt.

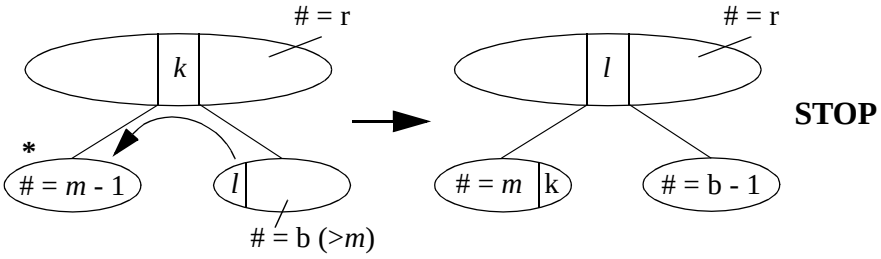
Sei *s* die Anzahl der Schlüssel in *K* nach dem Entfernen:

Fall 1: $s \geq m$: →**STOP**

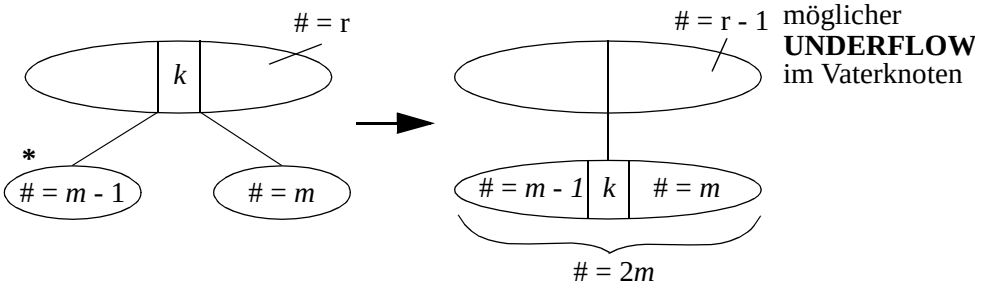
Fall 2: $s = m - 1$: →**UNDERFLOW**

UNDERFLOW-Behandlung:

Fall 1: Der Bruder hat $\geq m+1$ Schlüssel: → **Ausgleichen** mit dem Bruder



Fall 2: Der Bruder hat *m* Schlüssel: → **Verschmelzen** mit dem Bruder unter Hinzunahme des trennenden Vaterschlüssels
→möglicher **UNDERFLOW** im Vaterknoten.



So kann sich auch die UNDERFLOW-Behandlung bis zur Wurzel des Baumes fortsetzen.
Wird aus der Wurzel des Baumes der letzte Schlüssel entfernt, so wird diese gelöscht; die Höhe des Baumes verringert sich damit um 1.

Korollar: Die B-Bäume bilden eine Klasse balancierter Suchbäume.

Aufwandsabschätzung für die durchschn. Anzahl von Aufspaltungen (Splits) pro Einfügung:

Bezeichne *s* die durchschnittliche Anzahl der Knoten-Splits pro Einfügung:

*Modell***:**

Ausgehend von einem leeren Baum wird ein B-Baum der Ordnung *m* für die Schlüssel $k_1, k_2, \ldots, k_n$ durch *n* aufeinanderfolgende Einfügungen konstruiert.

Sei *t* die Gesamtzahl der Aufspaltungen bei der Konstruktion dieses B-Baumes

$\rightarrow s = t/n$.

Sei *p* die Anzahl der Knoten (Seiten) des B-Baumes mit $p \geq 3$

$\rightarrow t < p - 1$ (genauer: $t \leq p - 2$)

Für die Anzahl *n* der Schlüssel in einem B-Baum der Ordnung *m* mit *p* Knoten gilt:

$n \geq 1 + (p - 1) \cdot m$.

$\rightarrow p - 1 \leq \frac{n-1}{m} \rightarrow s = \frac{t}{n} < \frac{p-1}{n} \leq \frac{1}{n} \cdot \frac{n-1}{m} < \frac{1}{m}$ wobei im allg.: $600 \leq m \leq 900$ (s.o.).

Es gilt: $t \leq p - 2 \rightarrow t < p - 1$ für $p \geq 3$

Denn:

- bei jeder Aufspaltung wird mindestens ein zusätzlicher Knoten geschaffen
- Aufspalten der Wurzel → 2 zusätzliche Knoten
- wenn ein B-Baum mehr als einen Knoten hat, dann ist die Wurzel mindestens einmal aufgespalten worden
- dem ersten Knoten des B-Baums geht keine Aufspaltung voraus.