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)).

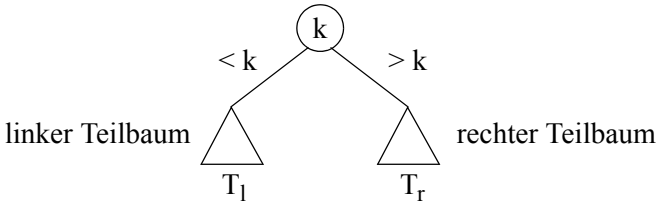
2.1 Binäre Suchbäume

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:

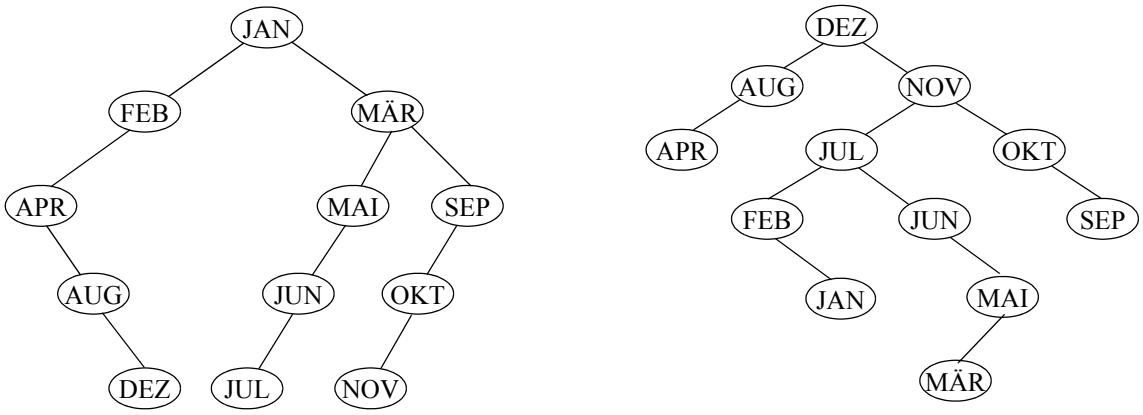


Anmerkungen:

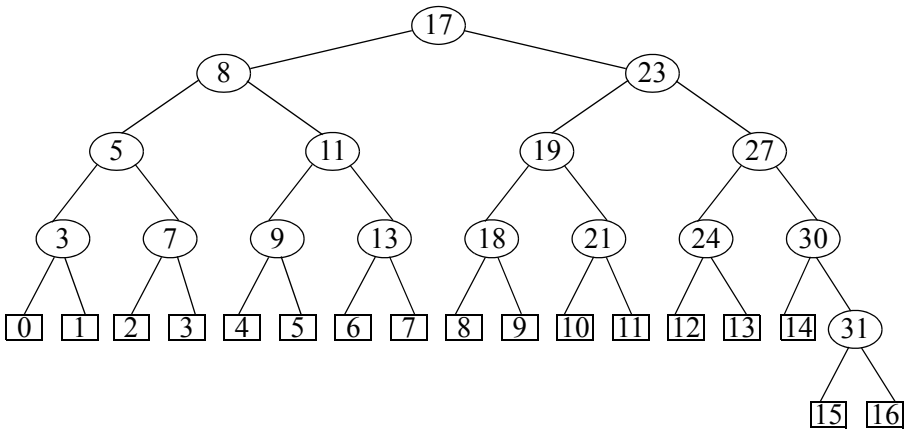
- Zur Organisation einer gegebenen Menge von n Schlüsseln gibt es eine große Anzahl unterschiedlicher binärer Suchbäume.
- Der *inorder*-Durchlauf eines binären Suchbaumes generiert die (aufsteigend) sortierte Folge der gespeicherten Schlüssel.

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.

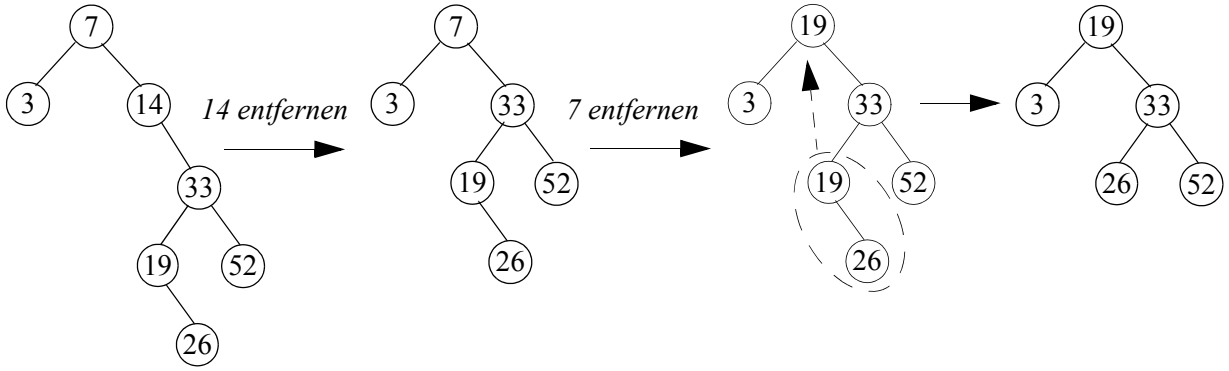
Die folgende Methode *find (x)* sucht in einem binären Suchbaum den Schlüssel *x* und liefert diesen zurück, falls er gefunden wurde. Andernfalls wird eine Ausnahmebehandlung durchgeführt.

```
// 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:

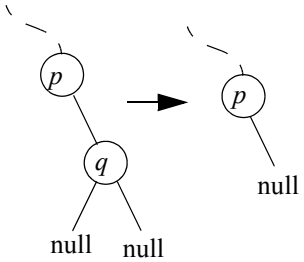


Allgemein treten zwei grundsätzlich verschiedene Situationen auf:

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

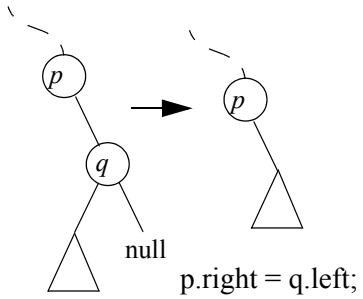
Fall 2: der Knoten *q* mit dem zu entfernenden Schlüssel besitzt zwei Söhne (innerer Knoten)

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



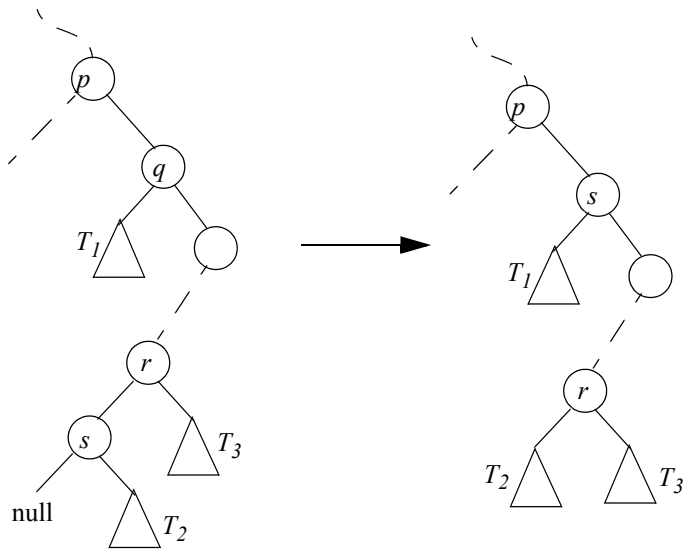
p.right = **null**;

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:

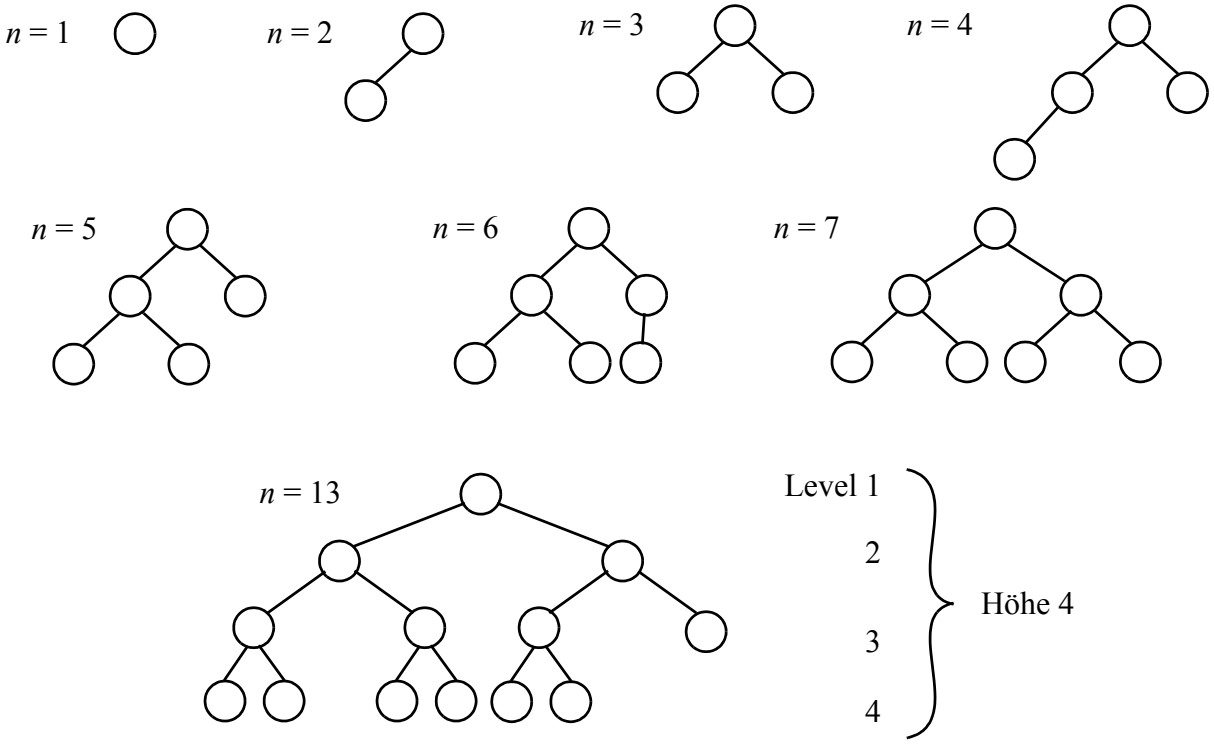
➠ im **worst-case** ist der Aufwand aller drei Operationen $O(n)$.

2.1.2 Vollständig ausgeglichene binäre Suchbäume

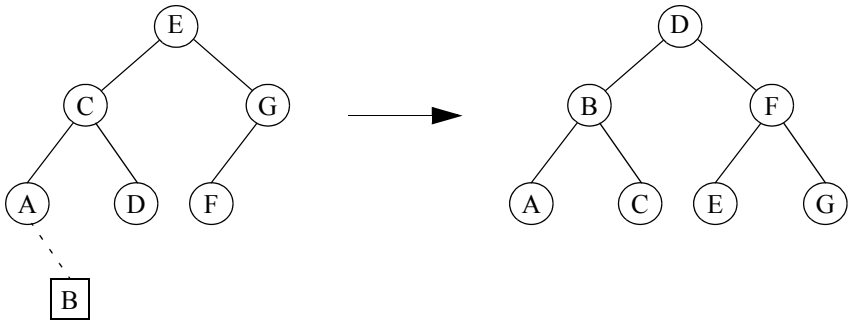
Die minimale Höhe $\lceil \log_2(n+1) \rceil$ unter allen binären Suchbäumen besitzen die **vollständig ausgeglichenen binären Suchbäume**, d.h. binäre Suchbäume, bei denen alle Levels bis auf das unterste vollständig besetzt sind.

➠ **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:** $O(n)$**.**

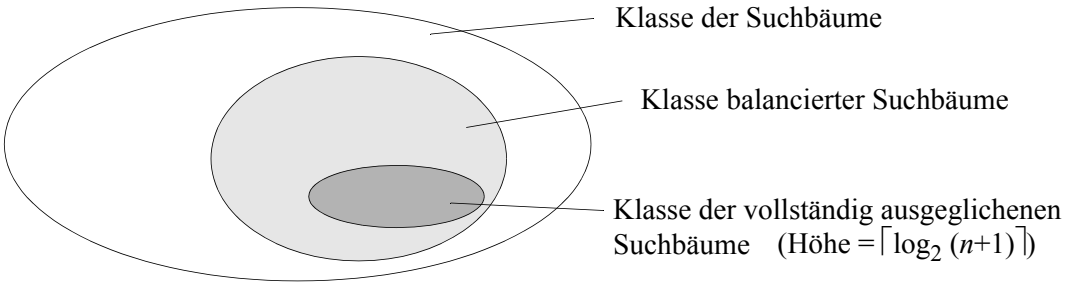
➠ Auswahl einer Kompromisslösung mit den Eigenschaften:

- 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:

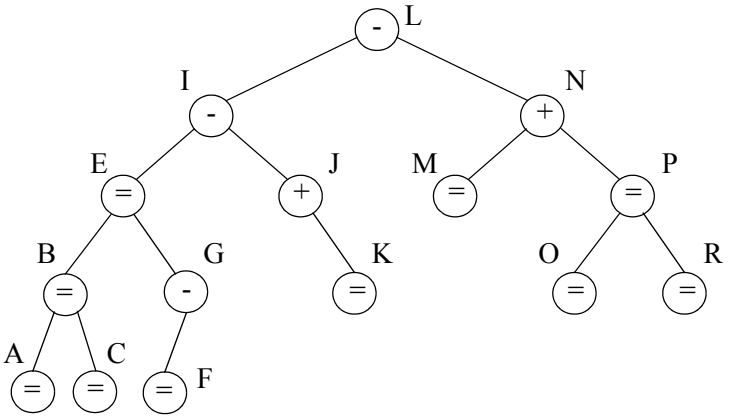
Ein binärer Suchbaum heißt **AVL-Baum**, falls für die beiden Teilbäume T_r und T_l der Wurzel gilt:

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

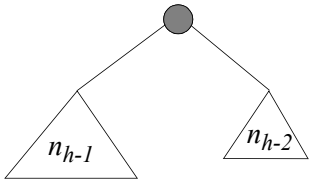


Behauptung: Die minimale Höhe $h_{min}(n)$ eines AVL-Baumes mit n Schlüsseln ist $\lceil \log_2(n+1) \rceil$. Dies folgt aus der Tatsache, dass ein AVL-Baum minimaler Höhe einem vollständig ausgeglichenen binären Suchbaum entspricht.

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

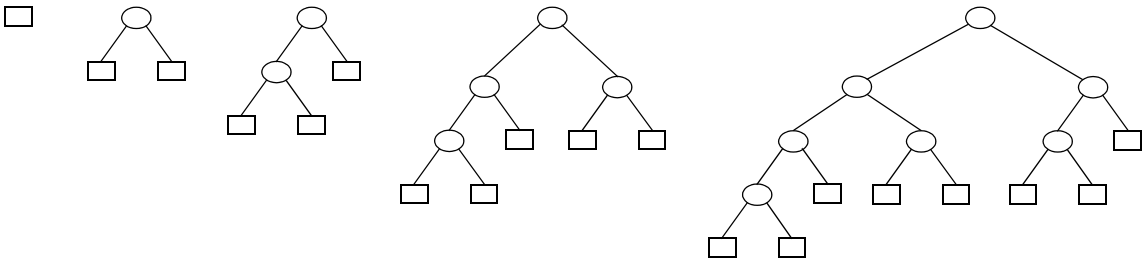
Beweis: Die maximale Höhe wird realisiert von sogenannten *minimalen AVL-Bäumen*. Dies sind AVL-Bäume, die für eine gegebene Höhe h die minimale Anzahl von Schlüsseln abspeichern.

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:
$$\begin{aligned} n_{h+1} &= n_h + n_{h-1} + 1 \\ &= 1 + fib(h+2) - 1 + fib(h+1) - 1 \\ &= fib(h+3) - 1 \end{aligned}$$

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.

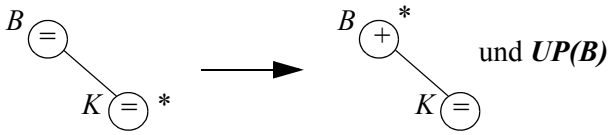
Also gilt die Behauptung: $h_{max}(n) = O(\log n)$.

Einfügen von Schlüsseln: *Insert(k)*

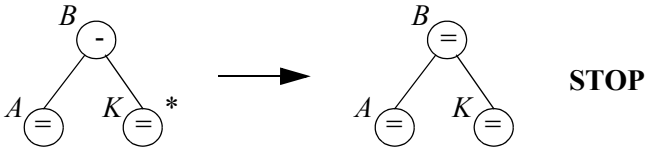
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

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

Fall 1: *B* ist ein Blatt

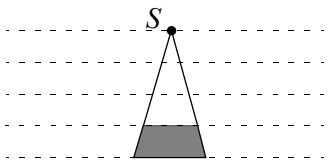


Fall 2: *B* hat einen linken Sohn



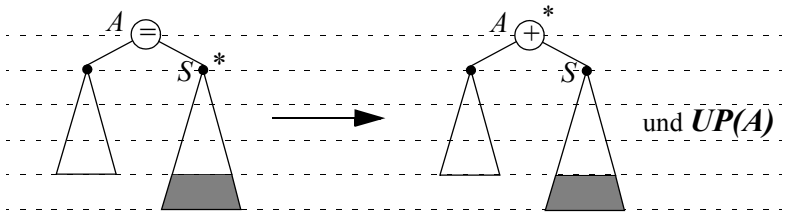
Die **Methode *UP(S)*** wird aufgerufen für einen Knoten *S*, dessen Teilbaum in seiner Höhe um 1 gewachsen ist. *S* ist die Wurzel eines korrekten AVL-Baumes.

➠ **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



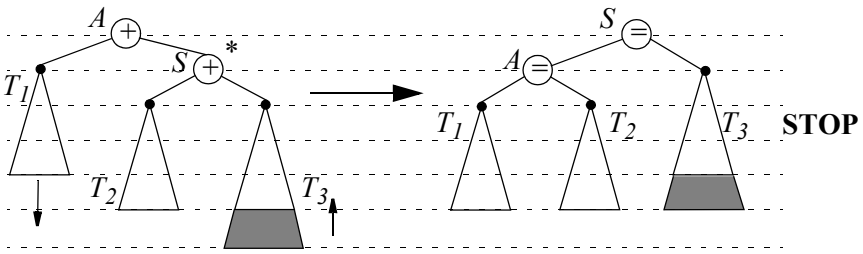
1.2 der Vater von *S* ist die Wurzel: ➙ dieselbe Transformation und **STOP**.

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.

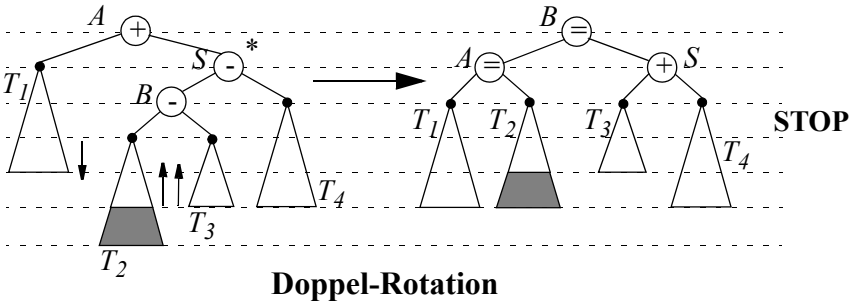
3.1 der Vater von *S* hat BF '+' und *S* hat BF '+':



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)*

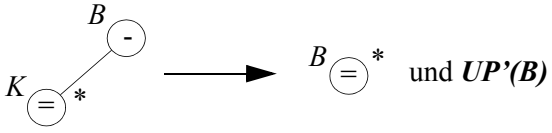
(In der folgenden Beschreibung sind symmetrische Fälle nicht dargestellt.)

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

Fall 1: *K* hat höchstens einen Sohn

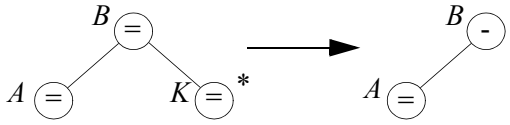
1.1 *K* ist ein Blatt

1.1.1 *K* hat keinen Bruder



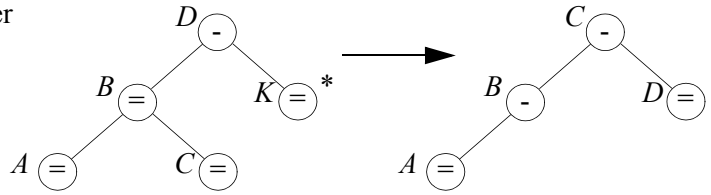
und ***UP'(B)***

1.1.2 *K* hat einen Bruder



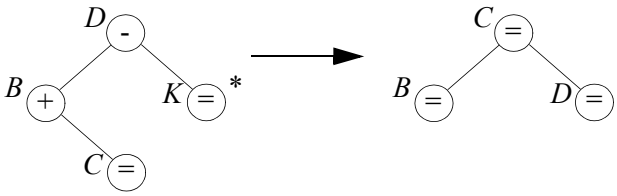
STOP

oder



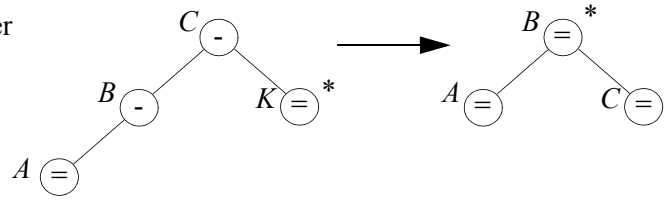
STOP

oder



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

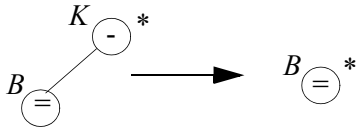
oder



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

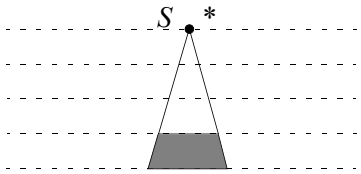
Fall 2: *K* ist ein innerer Knoten (hat zwei Söhne)

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*.

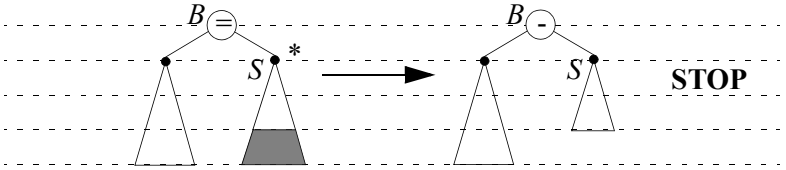
➠ damit haben wir Fall 2 auf Fall 1 zurückgeführt.

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.

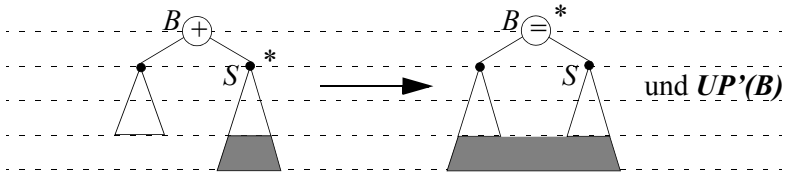
➙ **mögliche Strukturverletzung** durch einen zu niedrigen Teilbaum!



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



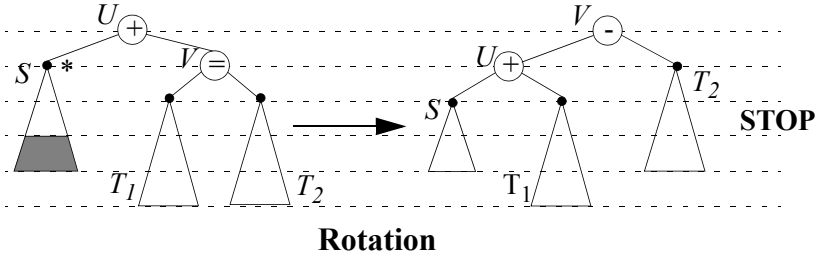
Fall 2: der Vater von *S* hat BF '+' oder '-' und *S* ist die Wurzel des höheren Teilbaums.



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

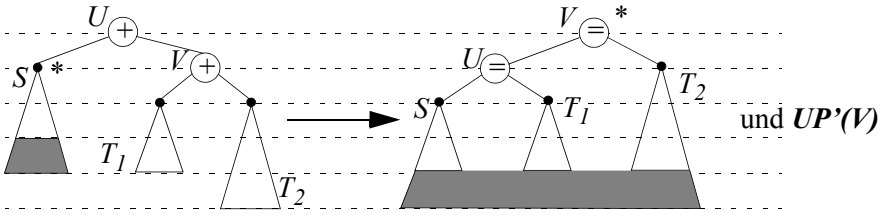
3.1 der Vater von *S* hat BF '+'

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



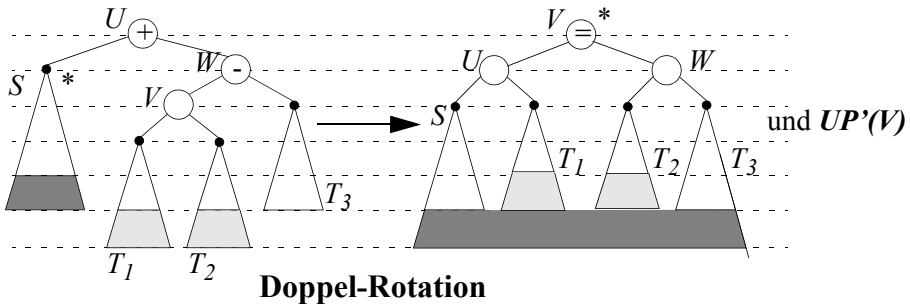
Rotation

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.**

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:

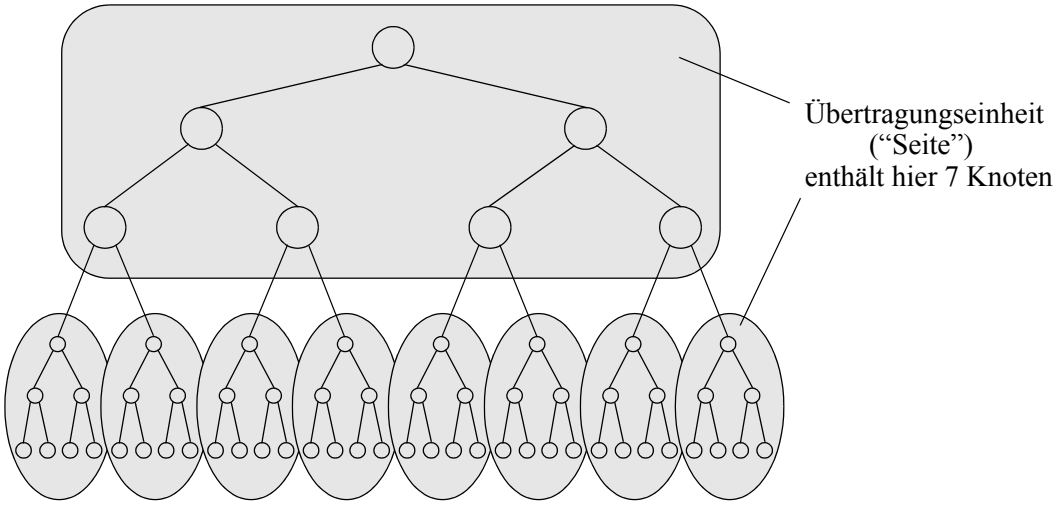
left oder right-Zeiger folgen $\triangleq$ 1 Plattenspeicherzugriff

Beispiel:

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

$$\Rightarrow \log_2(10^6) = \log_2(10^3)^2 = 2 \cdot \log_2(10^3) \approx 20 \text{ Plattenspeicherzugriffe}$$

Idee: Zusammenfassen mehrerer binärer Knoten zu einer Seite



Beispiel für einen B-Baum:

99 Knoten in einer Seite ➠ 100-fache Verzweigung:

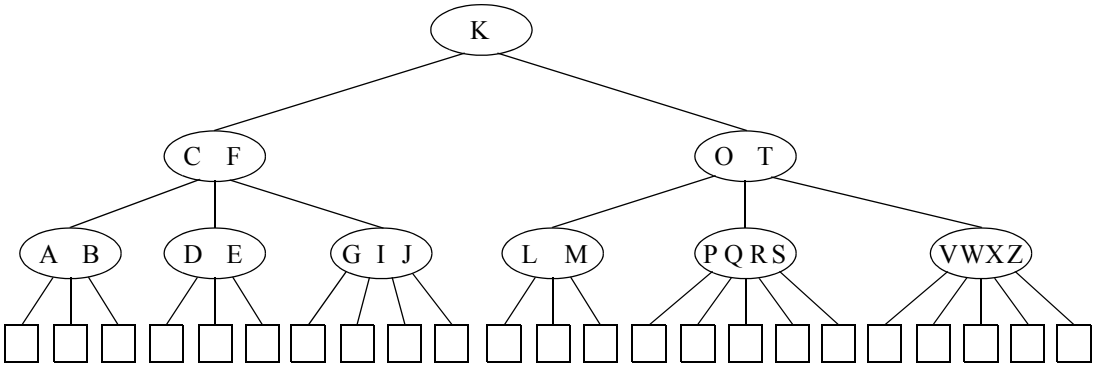
$$\Rightarrow \log_{100}(10^6) = 3 \text{ Plattenspeicherzugriffe.}$$

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

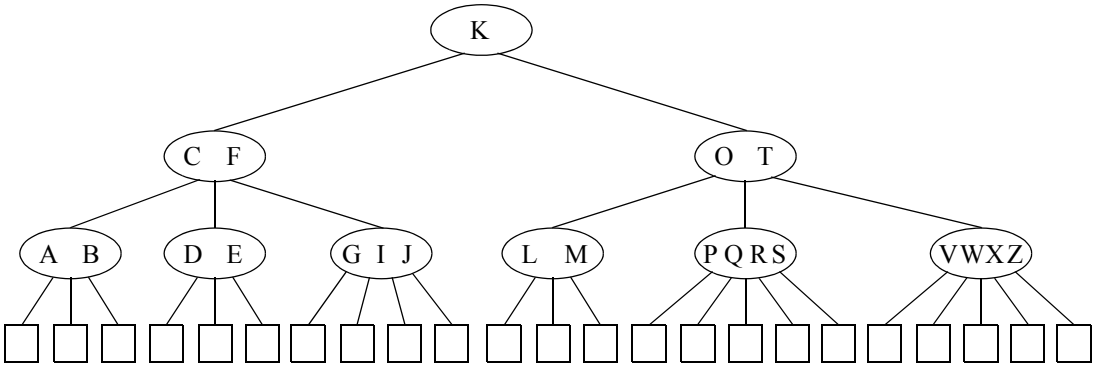
Definition: B-Baum der Ordnung *m* (Bayer und McCreight (1972))

(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



Beispiel: derselbe B-Baum nochmals zum Üben



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

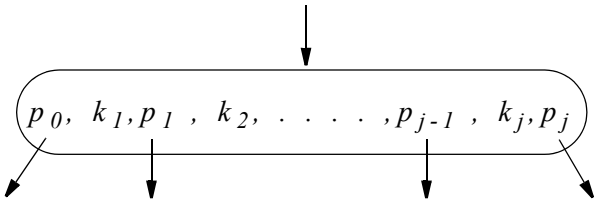
$$k_{h+1} = n + 1 \geq 2(m+1)^{h-1}, \text{ d.h. } h \leq 1 + \log_{m+1}\left(\frac{n+1}{2}\right).$$

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

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

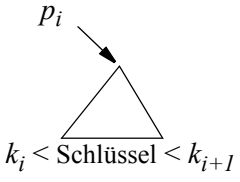
Beobachtung: Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt. Die Speicherplatzausnutzung beträgt also mindestens 50%!

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.

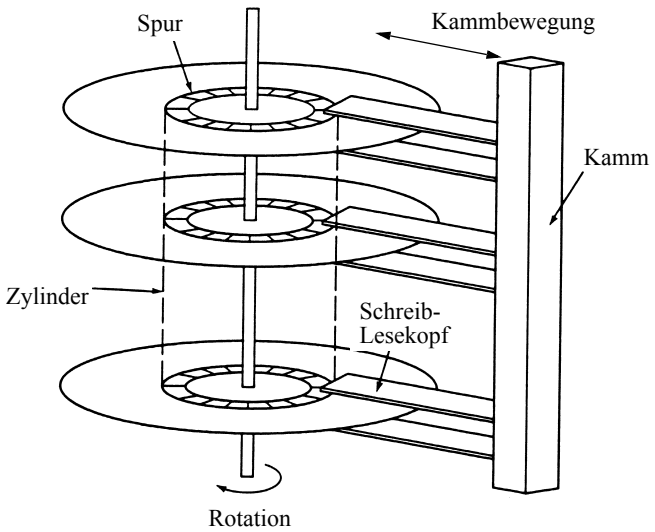
```
class Entry { public int key;
              public Page son;
              ... // Konstruktor..
              }

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:

Phase	**Hard Disk Drive**		**Solid State Disk**	
	Ursache	**Zeit**	**Ursache**	**Zeit**
Positionierungszeit (PZ)	Kammbewegung	8 ms	Addressberechung	0.1 ms
Latenzzeit (LZ)	Warten auf Sektor	4 ms	---	---
Übertragungszeit (ÜZ)	Übertragung der Daten	$3.3 \cdot 10^{-6}$ ms / Byte	Übertragung der Daten	$1{,}0 \cdot 10^{-5}$ 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 · $2m(\alpha + \beta) = a + b \cdot m$

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:

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

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

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

Für die obigen Konstanten setzen wir beispielhaft die folgenden Werte (eines HDD) ein:

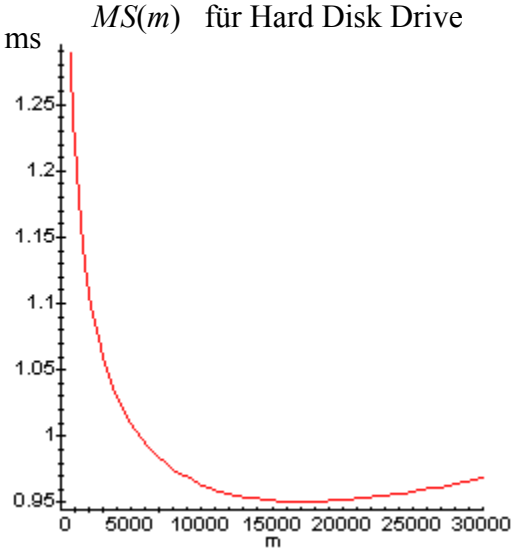
$a = 0{,}012s,\ a + d \approx a = 0{,}012s = 12\ ms$

$\alpha = 8, \beta = 4 \rightarrow b = 24 \cdot 3{,}3 \cdot 10^{-6} ms = 79{,}2 \cdot 10^{-6} ms$.

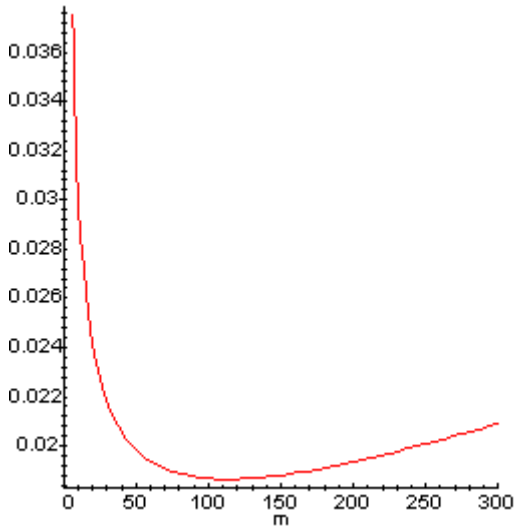
Damit ist die folgende Funktion $\overline{MS(m)}$ zu minimieren:

$$\overline{MS(m)} = \left(\frac{12}{\log_2(m)} + \frac{0{,}0000792 \cdot m}{\log_2(m)}\right) ms \quad \rightarrow \textbf{MIN}.$$

$\overline{MS(m)}$ für Hard Disk Drive



$\overline{MS(m)}$ für Solid State Disk



Die maximale Suchzeit ist somit nahezu minimal für $16000 \le m \le 19000$. Dies entspricht in obigem Beispiel einer "optimalen" Seitengröße von 432 KByte. Für die exemplarischen Werte der Solid-State-Disk ergibt sich eine „optimale" Seitengröße von 3,0 KB (für m = 125).

Einfügen in B-Bäumen:

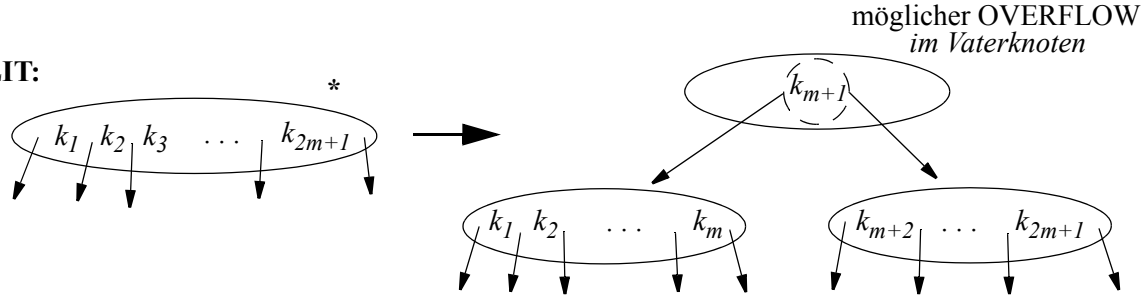
Zunächst wird der Knoten K gesucht, in den der neue Schlüssel einzufügen ist. Dieser ist stets ein Blatt. Der Schlüssel wird in K an der entsprechenden Stelle eingefügt.
Sei s die Anzahl der Schlüssel in K nach dem Einfügen:

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.

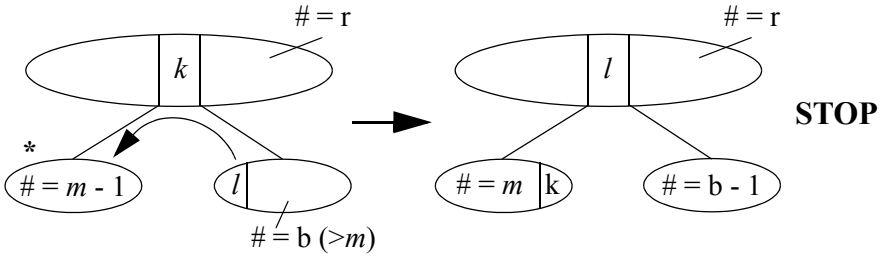
Sei s die Anzahl der Schlüssel in K nach dem Entfernen:

Fall 1: $s \geq m$: ➙ **STOP**

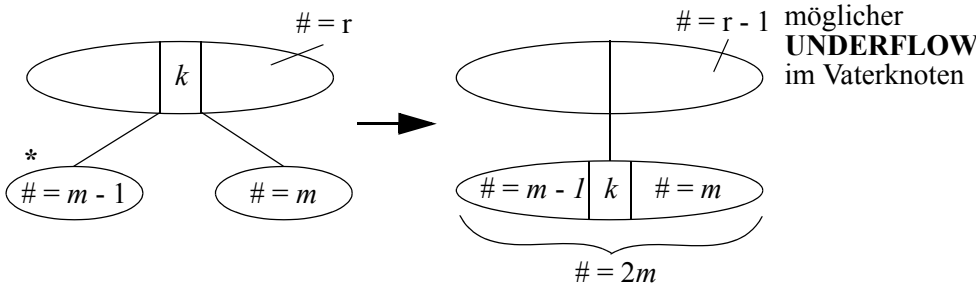
Fall 2: $s = m - 1$: ➙ **UNDERFLOW**

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
➞ $s = t/n$.

Sei *p* die Anzahl der Knoten (Seiten) des B-Baumes mit $p \geq 3$
➞ $t < p - 1$ (genauer: $t \leq p - 2$)

Für die Anzahl *n* der Schlüssel in einem B-Baum der Ordnung *m* mit *p* Knoten gilt:
$n \geq 1 + (p-1) \cdot m$.

➞ $p-1 \leq \frac{n-1}{m}$ ➞ $s = \frac{t}{n} < \frac{p-1}{n} \leq \frac{1}{n} \cdot \frac{n-1}{m} < \frac{1}{m}$ wobei im allg.: $16000 \leq m \leq 19000$ (s.o.).

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

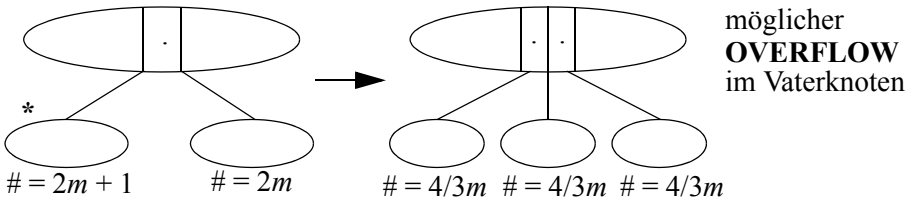
Denn:
- bei jeder Aufspaltung wird mindestens ein zusätzlicher Knoten geschaffen
- Aufspalten der Wurzel ➞ 2 zusätzliche Knoten
- wenn ein B-Baum mehr als einen Knoten hat, dann ist die Wurzel mindestens einmal aufgespalten worden
- dem ersten Knoten des B-Baums geht keine Aufspaltung voraus.

B*-Bäume

B*-Bäume verhalten sich im wesentlichen wie B-Bäume, jedoch wird bei **OVERFLOW** eines Knotens dieser nicht gleich aufgespalten, sondern wie beim UNDERFLOW der Bruder betrachtet:

Fall 1: Bruder hat $b \leq 2m-1$ Schlüssel ➞ **Ausgleichen** mit dem Bruder

Fall 2: Bruder hat $b = 2m$ Schlüssel ➞ **Verteilen** auf drei Knoten:



Definition: B*-Baum der Ordnung *m*

m sei ein Vielfaches von 3.

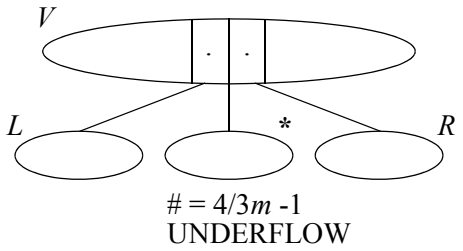
(1) Jeder Knoten außer der Wurzel enthält mindestens $\frac{4}{3}m$, höchstens 2*m* Schlüssel.
(2) Die Wurzel enthält mindestens einen, höchstens $\frac{8}{3}m$ Schlüssel.
(3) Ein Knoten mit *k* Schlüsseln hat genau *k*+1 Söhne.
(4) Alle Blätter befinden sich auf demselben Level.

B*-Bäume besitzen eine Speicherplatzausnutzung von mindestens 66%!

Nach einer zum B-Baum äquivalenten Berechnung ergibt sich für die maximale Höhe h_{max} eines B*-Baumes der Ordnung *m* mit *n* Schlüsseln:

$$h_{max} = \left\lfloor \log_{4/3m+1}(\frac{n+1}{2}) \right\rfloor + 1 .$$

Entfernen in B*-Bäumen

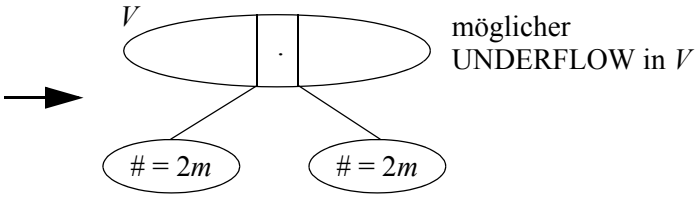


Fall 1: *R* hat mehr als $\frac{4}{3}m$ Schlüssel → Ausgleichen von * und *R*

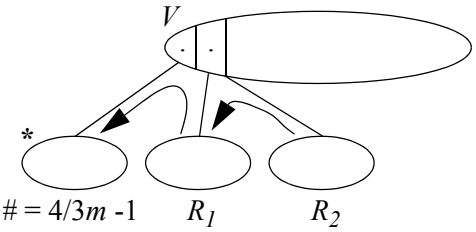
Fall 2: *R* hat $\frac{4}{3}m$ Schlüssel:

2.1 *L* hat mehr als $\frac{4}{3}m$ Schlüssel → Ausgleichen von * und *L*

2.2 *L* hat $\frac{4}{3}m$ Schlüssel:



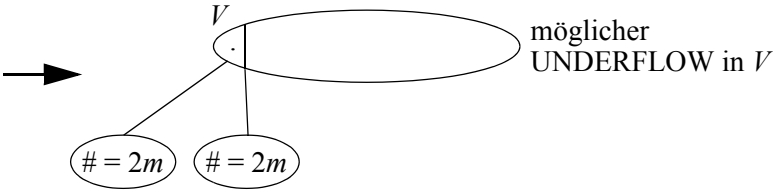
Fall 3: * hat nur einen direkten Bruder, aber weitere indirekte Brüder:



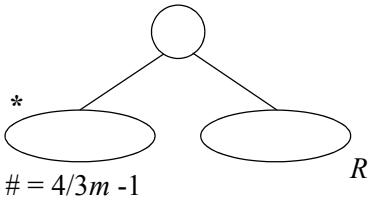
3.1 Falls R_1 mehr als $\frac{4}{3}m$ Schlüssel besitzt → Ausgleichen von * und R_1

3.2 Falls R_1 genau $\frac{4}{3}m$ Schlüssel besitzt und R_2 mehr als $\frac{4}{3}m$ Schlüssel besitzt
→ zuerst Ausgleichen von R_1 und R_2, dann von * und R_1

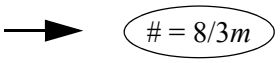
3.3 Falls R_1 und R_2 jeweils genau $\frac{4}{3}m$ Schlüssel besitzen



Fall 4: * ist Sohn einer binären Wurzel:



4.1 R hat $\frac{4}{3}m$ Schlüssel:



neue Wurzel : **STOP**

4.2 R hat mehr als $\frac{4}{3}m$ Schlüssel → Ausgleichen von * mit R

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

Bemerkung: In fast allen gängigen Datenbanksystemen sind B-Bäume, B*-Bäume oder deren Varianten zur effizienten Ausführung von Suchoperationen implementiert.

2.3 Optimale binäre Suchbäume

Bisher: Häufigkeiten, mit denen einzelne Schlüssel gesucht werden, bleiben unberücksichtigt. → **Annahme der Gleichverteilung** der Suchhäufigkeiten

Jetzt: Wir **berücksichtigen Häufigkeiten**, mit denen einzelne Schlüssel gesucht werden
→ Je häufiger ein Schlüssel gesucht wird, desto höher im Baum soll er plaziert werden.

Anwendungsbeispiel:
→ "Tabelle" der Schlüsselworte bei Compilern (*statisch!*).

Seien die Zugriffshäufigkeiten zu den n einzufügenden Schlüsseln $k_1, k_2, \ldots, k_n$ mit $p_1, p_2, \ldots, p_n$ ($\sum_{i=1}^{n} p_i = 1$ *nicht unbedingt erforderlich*) zum Zeitpunkt der Einfügung bekannt.

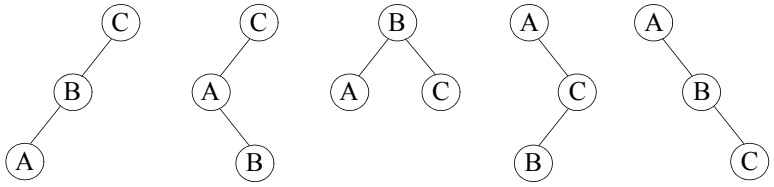
Problem: Wie finden wir unter diesen Voraussetzungen einen "optimalen" Suchbaum aus den $\binom{2n}{n}/(n+1) \approx 4^n/(\sqrt{\pi} \cdot n^{3/2})$ verschiedenen Suchbäumen für n Schlüssel?

Beispiel:

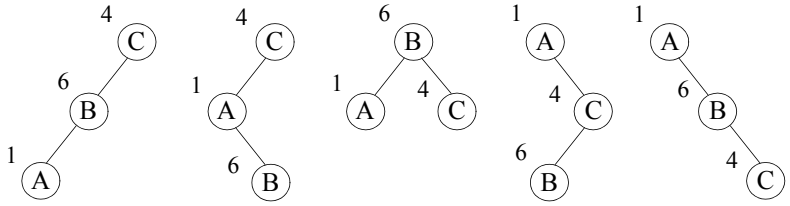
Sei $n = 3$;

Gegeben seien die Schlüssel A, B und C mit den Zugriffshäufigkeiten p_A, p_B und p_C.

→ 5 mögliche Suchbäume; welcher optimal ist, hängt vom Gewicht (den "Kosten") des Baumes und damit von den Gewichten der einzelnen Schlüssel ab.



Konkretes Beispiel: $p_A = 1$, $p_B = 6$ und $p_C = 4$.



Kosten → Min. 19 24 **16** 27 25

Kosten = α·(erwartete Anzahl von Knotenzugriffen) + β·(erwartete Anzahl von Schlüsselvergleichen)

wobei: α : Kosten für einen Knotenzugriff

β : Kosten für einen Schlüsselvergleich

Für einen binären Baum gilt: Kosten = (α + β) · (erwartete Anzahl von Schlüsselvergleichen)

Für eine allgemeine Lösung werden wir zusätzlich erfolglose Suchoperationen, d.h. die Suche von Schlüsseln, die zwischen den vorhandenen Schlüsseln liegen, berücksichtigen.

Damit ergibt sich die folgende allgemeine **Problemstellung**:

Gegeben: n Schlüssel: $k_1 < k_2 < \ldots < k_n$.

$2n+1$ Gewichte (Häufigkeiten):

$p_1, p_2, \ldots, p_n$ und $q_0, q_1, q_2, \ldots, q_n$ mit $w = \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i$, wobei:

p_i/w = Wahrscheinlichkeit (k_i = Suchargument)

q_i/w = Wahrscheinlichkeit ($k_i <$ Suchargument $< k_{i+1}$)

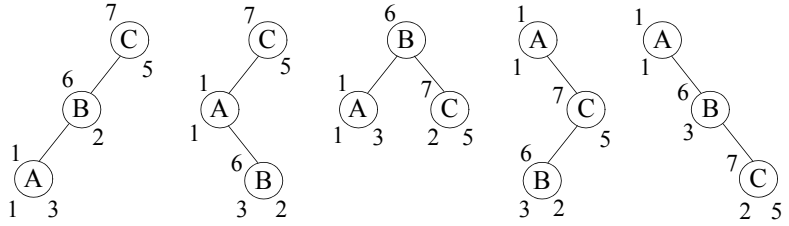
(o.E. $k_0 = -\infty$, $k_{n+1} = \infty$)

Zu konstruieren ist ein binärer Suchbaum derart, dass die zu erwartende Anzahl der bei einer Suchoperation auszuführenden Schlüsselvergleiche, ausgedrückt durch das folgende **Kostenmaß *K***, minimiert wird:

$$K = \sum_{i=1}^{n} p_i \cdot lev(\bigcirc i) + \sum_{j=0}^{n} q_j \cdot (lev(\square j) - 1) \quad \rightarrow \textbf{MIN}.$$

mit: $\bigcirc j$ bezeichne den Baumknoten mit Schlüssel k_j und
$\square k$ bezeichne den äußeren Baumknoten "zwischen" $\bigcirc k$ und $\bigcirc(k+1)$.

Beispiel: Seien im obigen Beispiel: $p_A = 1, p_B = 6, p_C = 7$ und $q_0 = 1, q_1 = 3, q_2 = 2, q_3 = 5$.

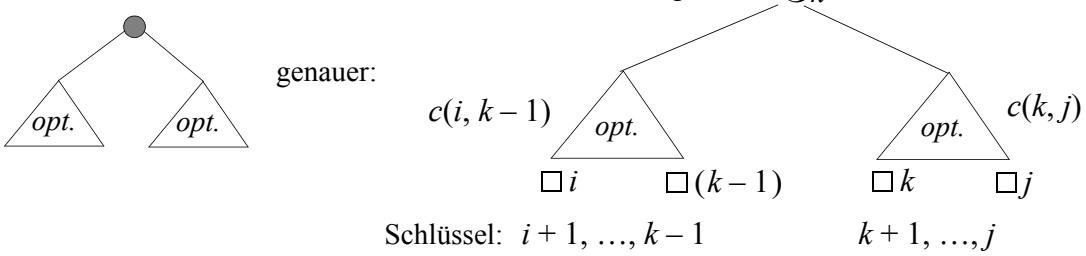


Optimalitätskriterium: Alle **Teilbäume** eines optimalen Suchbaumes sind **optimal**.

→ **Idee**: ausgehend von einzelnen Knoten als minimalen Teilbäumen werden systematisch "immer größere" optimale Teilbäume konstruiert.

"Bottom-up" - Methode:

Ein neuer optimaler Teilbaum ergibt sich aus einer geeigneten Wurzel und zwei optimalen Teilbäumen (*bereits berechnet*) dieser Wurzel.



Seien $c(i,j)$, $0 \le i \le j \le n$, die Kosten eines optimalen Teilbaumes mit den Gewichten $p_{i+1}, \ldots, p_j$; $q_i, \ldots, q_j$; sei weiterhin: $w(i,j) = \sum_{k=i+1}^{j} p_k + \sum_{l=i}^{j} q_l$.

Dann können die Werte $c(i,j)$ nach folgendem Rekursionsschema berechnet werden:

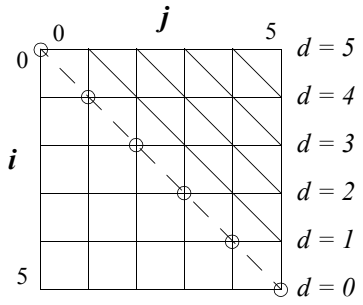
$c(i, i) = 0$ ➞ der Baum besteht aus $\square i$ (*leerer Baum*).

$$c(i,j) = w(i,j) + \min_{i < k \le j} (c(i, k-1) + c(k,j)) \qquad \text{für } i < j .$$

Denn: das minimale Gewicht des Teilbaums mit Wurzel $\bigcirc k$ und den Grenzen i bzw. j ist gegeben durch: $w(i,j) + c(i, k-1) + c(k,j)$.

Mit Hilfe dieses Rekursionsschemas wird $c(0, n)$ und der zugehörige optimale binäre Suchbaum bestimmt.

Es gibt hierbei $\approx \frac{n^2}{2}$ Werte $c(i,j)$ ➞ **Platzbedarf** $O(n^2)$.



Für $d = j - i$, $1 \le d \le n$, wird $c(i,j)$ ermittelt.

Laufzeitbedarf für diese Art der Konstruktion eines optimalen binären Suchbaumes:

Für $d = j - i$, $1 \le d \le n$, berechnen wir $c(i,j)$.

Wieviele Werte für $\bigcirc k$ müssen bei der Minimum-Bestimmung betrachtet werden?

Für jedes d, $1 \le d \le n$, gibt es $n - d + 1$ verschiedene $c(i,j)$'s.

Für jedes dieser $c(i,j)$ mit $j - i = d$ gibt es d mögliche Werte für $\bigcirc k$.

Damit gibt es für jedes d, $1 \le d \le n$, $d \cdot (n - d + 1)$ mögliche Werte für $\bigcirc k$.

Insgesamt $\sum_{d=1}^{n} d \cdot (n - d + 1) = \frac{n^3}{6} + O(n^2)$ mögliche Werte für $\bigcirc k$.

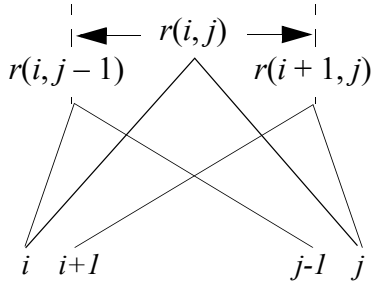
$$\left(\sum_{d=1}^{n} d^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6} \right)$$

➞ **Laufzeitbedarf**: $O(n^3)$; dies ist für einen praktischen Einsatz indiskutabel!

Es gilt jedoch folgende **Monotonie-Eigenschaft**:

Sei $r(i,j)$ ein Wert für $\bigcirc k$, für den $c(i,j)$ minimal wird.

$$\rightarrow r(i, j-1) \le r(i,j) \le r(i+1, j)$$



➙ Es genügt: $c(i,j) = w(i,j) + \min_{r(i,j-1) \le k \le r(i+1,j)} (c(i,k-1) + c(k,j))$ zu berechnen.

Für jedes d, $1 \le d \le n$, müssen damit also weniger Werte für Θk untersucht werden, nämlich nur noch:

$$\sum_{\substack{d \le j \le n \\ i = j-d}} (r(i+1,j) - r(i,j-1) + 1) = r(n-d+1,n) - r(0,d-1) + (n-d+1) < 2n$$

➙ **Laufzeitbedarf**: reduziert auf $O(n^2)$.

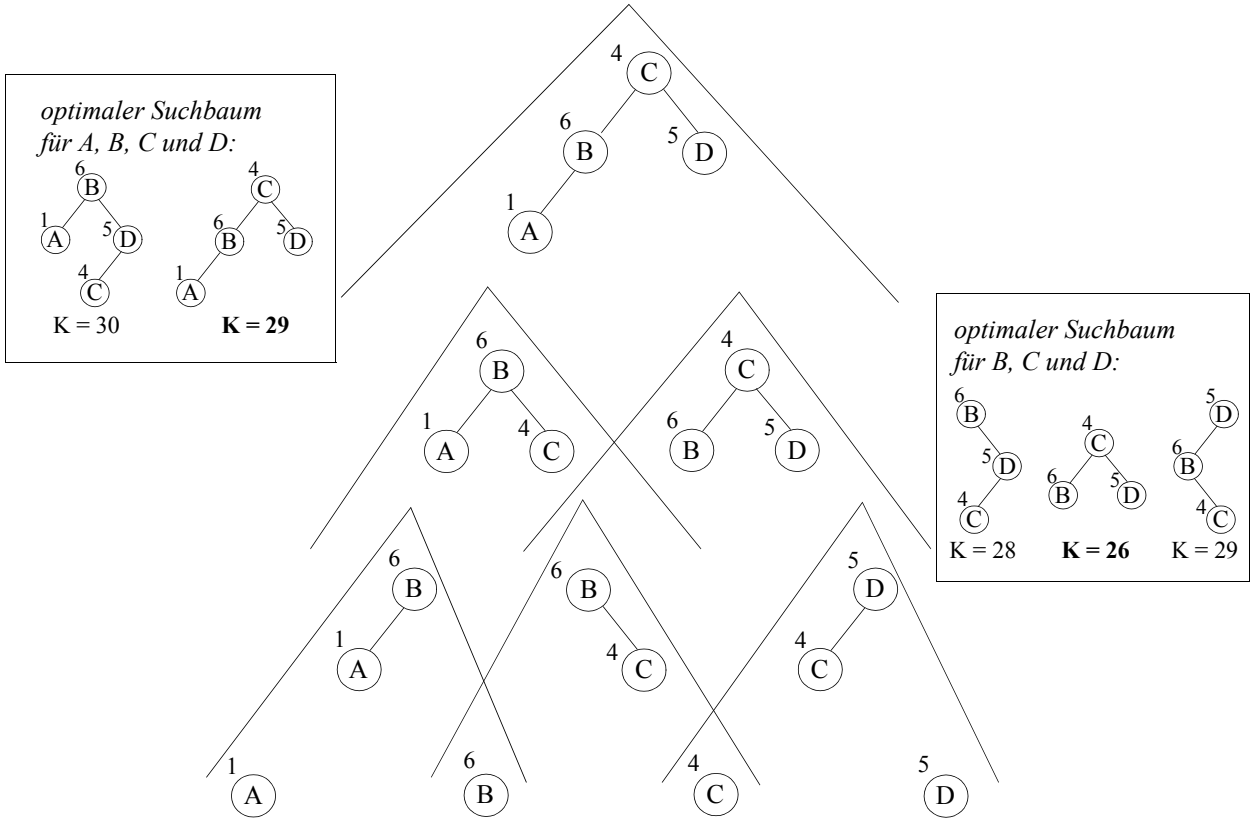
Abschließendes Beispiel:

Sei $n = 4$;

Gegeben seien die Schlüssel A, B, C und D mit den Zugriffshäufigkeiten:

$p_A = 1$, $p_B = 6$, $p_C = 4$ und $p_D = 5$.

Sei weiterhin: $q_i = 0$, $0 \le i \le 4$.



Wie bereits angedeutet, sind optimale binäre Suchbäume nur für statische Anwendungen geeignet (kein Einfügen oder Entfernen). Nahezu-optimale Suchbäume zur Handhabung von Zugriffshäufigkeiten und gleichzeitig Einfügen bzw. Entfernen werden später behandelt.

2.4 Hashverfahren

Bisher: Suchen mit Hilfe von **Schlüsselvergleichen**.

Jetzt: stattdessen **Adressberechnung**

→ Auswertung einer Funktion, der sogenannten **Hash-** oder **Adressfunktion**.

Vorteil: Die Suche erfolgt weitgehend unabhängig von den anderen Schlüsselwerten

→ sie ist in der Regel schneller.

(im Durchschnitt konstanter statt logarithmischer Aufwand)

2.4.1 Allgemeine Einführung

Prinzipielle Idee von Hashverfahren:

Der Speicher sei aufgeteilt in eine Folge von m Speicherzellen (*buckets*) $B_0, B_1, \ldots, B_{m-1}$, die jeweils einen oder mehrere (b = *'bucketsize'*) Datensätze (Schlüssel) aufnehmen können. Jeder Schlüssel wird durch die Hashfunktion eindeutig einem bucket B_i zugeordnet.

Sei K die Menge der Schlüssel, so stellt eine *Hashfunktion* h eine Abbildung der Schlüssel in die Menge der Bucketadressen dar:

$$h : K \to [0, 1, \ldots, m-1] .$$

Eine Suchoperation teilt sich dann auf in:

- Auswerten der Hashfunktion um die Adresse der Speicherzelle zu finden
  ➞ *konstanter* Zeitaufwand
- Zugriff auf die entsprechende Speicherzelle
  ➞ im Idealfall: *konstanter* Zeitaufwand

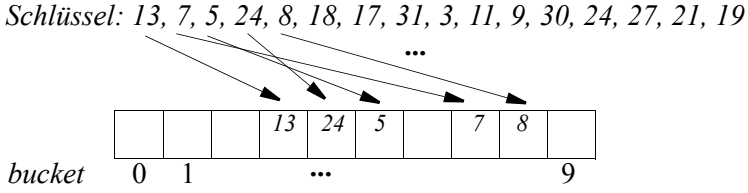
➞ im Idealfall: Gesamtzeitaufwand für eine Suchoperation: *konstant*.

Bemerkung: Ein Hashverfahren, bei dem die obige Suchstrategie in jedem Fall zum Erfolg führt, d.h. gewährleistet, dass jeder Adresse höchstens b Schlüssel zugeordnet werden, nennt man **perfektes Hashing** ('*perfect hashing*').
Wir werden jedoch sehen, dass diese Eigenschaft in der Regel nicht erfüllt ist. Sie kann im Gegenteil nur unter ganz bestimmten Voraussetzungen, z.B. bei statischen oder sehr kleinen Schlüsselmengen, erfüllt werden.
Damit ist auch unsere Zeitabschätzung für den Idealfall 'in Gefahr'.

Beispiel:

Für *ganzzahlige Schlüssel*: $h : K \to [0, 1, \ldots, m-1]$ mit $h(k) = k \text{ MOD } m$.

sei: $m = 10$



Für *Zeichenketten*: Benutze die 'ORD-Funktion' (ORD ('a') = 1, ORD ('b') = 2, . . .) zur Abbildung auf ganzzahlige Werte, z.B.:

$$h : STRING \rightarrow \left(\sum_{i=0}^{\text{s.length - 1}} (ORD(s[i]) - ORD(\text{'a'}) + 1) \right) \text{MOD } m$$

sei: $m = 15$

JAN	➞ 25 MOD 15 = 10	JUL	➞ 43 MOD 15 = 13
FEB	➞ 13 MOD 15 = 13	AUG	➞ 29 MOD 15 = 14
MAR	➞ 32 MOD 15 = 2	SEP	➞ 40 MOD 15 = 10
APR	➞ 35 MOD 15 = 5	OKT	➞ 46 MOD 15 = 1
MAI	➞ 23 MOD 15 = 8	NOV	➞ 51 MOD 15 = 6
JUN	➞ 45 MOD 15 = 0	DEZ	➞ 35 MOD 15 = 5

Erwünschte Eigenschaften für solche Hashfunktionen:

- *Surjektivität*, d.h. alle Speicherzellen werden erfasst (keine leeren Speicherbereiche)
- die Schlüssel sollten *gleichmäßig* über alle Speicherzellen verteilt werden. (*ideales Hashverfahren*)
- der Wert der Hashfunktion sollte *effizient* zu berechnen sein.

Im allgemeinen, wie auch in obigem Beispiel gilt: $card(K) \gg m$.

Weiteres Beispiel: *Personaldatei*

mögliche Schlüssel sind alle Nachnamen bis zur Länge 10 ➞ $card(K) = 26^{10}$

aber: die Menge der zu erfassenden Personen nur z.B. 10 000 ➞ $m = 10000$

Beobachtung: die Menge K aller möglichen Schlüssel ist häufig um mehrere Größenordnungen umfangreicher als die Menge der verfügbaren Adressen.

➞ keine *injektive* Abbildung möglich!

Prinzipielles Problem des Ansatzes von **Hashverfahren**:

Kollisionen, d.h. die Hashfunktion ordnet einem Bucket mehr Schlüssel zu, als dieses fasst. Im allgemeinen ist für Hashverfahren im Hauptspeicher die Bucketkapazität $b = 1$, bei Verfahren für den Sekundärspeicher ist $b > 1$.

➞ negative Auswirkungen auf die Performanz, da Kollisionen die Suchzeit verlängern

➞ die Hashfunktion sollte nicht einfach zufällig ausgewählt werden, sondern auf die Minimierung solcher Kollisionen ausgerichtet sein!

Dass man zur Minimierung von Kollisionen selbst bei kleiner Schlüsselmenge K und großem Adressraum (großem m) nicht einfach irgendeine zufällige Funktion $h : K \to \{0, 1, \ldots, m-1\}$ wählen darf, zeigt das folgende

Geburtstagsparadoxon:

Wir wählen 23 Personen zufällig aus ($card(K) = 23$). Dann sind auch deren Geburtstage zufällig, d.h. die 'Hashfunktion' h, die jeder Person ihren Geburtstag zuordnet, ist eine zufällige Adressfunktion mit $m = 365$.

Nach wahrscheinlichkeitstheoretischen Überlegungen gilt:

$$W(\text{die Geburtstage der 23 Personen sind verschieden}) \approx 0{,}493 \quad .$$

➞ bereits bei einer derart geringen Anzahl von Schlüsseln ist die Wahrscheinlichkeit einer Kollision bei zufällig gewählter Adressfunktion > 50%.

Bemerkung: Auch bei nach obigen Kriterien 'guten' Hashfunktionen können Kollisionen nicht grundsätzlich vermieden werden, sofern die tatsächlich auftretenden Schlüssel nicht von vorneherein (statisch) bekannt sind.
➞ geeignete Strategien zur Behandlung von Kollisionen sind nötig.

Folgerung:

Bei der Definition eines Hashverfahrens sind zwei unabhängige Teilprobleme zu lösen:
- Bestimmung einer 'guten', kollisionsminimierenden Hashfunktion
- Angabe einer geeigneten Kollisionsstrategie

In den folgenden Untersuchungen gehen wir zunächst auf Hashverfahren **im Hauptspeicher** ein. Hierbei geht man von einer Kapazität der Speicherzellen von einem Eintrag ($b = 1$) aus.

2.4.2 Hashfunktionen

Problem: Gesucht ist eine Hashfunktion $h : K \to [0, 1, \ldots, m-1]$, die nichtzufälligen Schlüsseln möglichst zufällige Adressen zuordnet. (*Randomisierung*)

Im folgenden sei die Schlüsselmenge K stets als **int** vorgegeben.

Divisionsmethode:

Allgemeiner Ansatz: $h(k) = k \text{ MOD } m$.

(*Modifikation*: $h(k) = (\text{ORD}^*(k)) \text{ MOD } m$, falls k ein nichtnumerischer Schlüssel und $\text{ORD}^*(k)$ eine Funktion ist, die k auf einen ganzzahligen Wert abbildet)

Welche Werte sind für m geeignet?

$m = 2^d$: $h(k) \equiv$ *letzte d Bits der Binärzahl k* → *keine zufällige Zuordnung!*

geradzahliges m: $h(k)$ gerade $\Leftrightarrow$ k gerade → *ebenfalls ungünstig*!

m Primzahl > 2: → *gut geeignet*

aber: aufeinanderfolgende Schlüssel $k, k+1, \ldots$ werden mit hoher Wahrscheinlichkeit auf aufeinanderfolgende Adressen ($h(k+1) = h(k) + 1$) abgebildet.

Middle-Square-Methode:

Allgemeiner Ansatz: Der Schlüssel k wird quadriert, und man nimmt als Hashadresse $h(k)$ die "mittleren" Ziffern der Repräsentation von k^2.

Sei z.B. $m = 10^3$ und seien die Schlüssel 10-stellige Zahlen.

Für einen Schlüssel $k = a_1 a_2 \ldots a_{10}$ berechne $k^2 = b_1 b_2 \ldots \boldsymbol{b_9 b_{10} b_{11}} \ldots b_{20}$.

Als Wert der Hashfunktion verwenden wir: $h(k) = b_9 b_{10} b_{11}$.

Vorteil: $b_9 b_{10} b_{11}$ hängt von allen Ziffern $a_1 a_2 \ldots a_{10}$ von k ab
→ bessere Streuung aufeinanderfolgender Schlüssel.

aber: dieses Verfahren versagt, falls viele 'Nullen' am Anfang oder am Ende von k stehen.

Beispiel:

Sei $m = 100$

k	$h_{\text{Div.}}(k)$	k^2	$h_{\text{M-Sq.}}(k)$
550	50	30**25**00	25
551	51	30**36**01	36
552	52	30**47**04	47
553	53	30**58**09	58

2.4.3 Kollisionsstrategien

Beispiel:

$m = 11$ und $h(k) = k$ MOD 11

k	7	16	21	30	57	62	78	80
$h(k)$	**7**	5	10	8	2	**7**	1	3

Kollision

Bei der Behandlung von Kollisionen gibt es zwei prinzipielle Ansätze: offene Hashverfahren und geschlossene Hashverfahren, die im folgenden beide behandelt werden.

Offene Hashverfahren:

Hierbei geht man davon aus, dass jeder Hashadresse beliebig viele Schlüssel zugeordnet werden können. Diese werden in geeigneter Form (mittels einer *overflow area*) organisiert. Kollisionen werden somit durch ein 'erlaubtes Überlaufen' einer Speicherzelle behandelt.

Vorteile: die Gesamtanzahl der Schlüssel ist nicht beschränkt.

Probleme: zusätzlicher Speicherbedarf; hoher Suchaufwand im schlechtesten Fall.

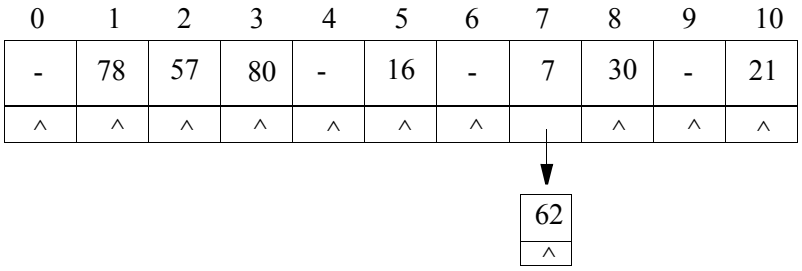
Direkte Verkettung (*'separate chaining'*):

Jedem Eintrag der Hashtabelle wird eine separate Liste von Elementen zugeordnet.

Sei:
```
class Entry {        public int key;
                     public Entry next;}

class OpenHashtable{
    protected Entry[] hTable;
    protected int m;
    public OpenHashtable (int size)
        { hTable = new Entry[size];
          m = size;}
    // weitere Methoden
    }
```

OpenHashtable T = **new** OpenHashtable(11);



Methode für das Suchen eines Objektes mit Schlüssel *k*:

```
public boolean search (int k)
{     Entry p;
      p = hTable [h(k)]; // h(k) sei die Hashfunktion, z.B.: h(k) = k % m;
      while (p != null)
      {     if (p.key == k) return true;
            else p = p.next;
      }
      return false;
}
```

Bei n Schlüsseln und m Adressen enthält jede der Listen im Mittel n/m Elemente
➞ das Hashing verbessert somit den Aufwand der sequentiellen Suche um den Faktor m.

Einschränkung für die Cardinalität m der Menge der Hashadressen:
m zu klein : sehr lange Listen von Objekten ➞ schlechte Zugriffszeit
m zu groß : viele leere Listen und damit unnötig belegter Speicherplatz.
➞ gute Wahl von m: $m \approx n$.

Gute Hashfunktion ➞ keine der Listen wird übermäßig lang (*Ziel*: alle Listen gleich lang)
➞ guter durchschnittlicher Zeitaufwand: Suche, Einfügen und Entfernen in $O(1)$ Zeit (für $m = n$).

aber: Degenerierung im worst-case möglich, auch bei gut gewählter Hashfunktion.
➞ Zeitaufwand: $O(n)$ im worst-case und Platzbedarf: $O(n+m)$.

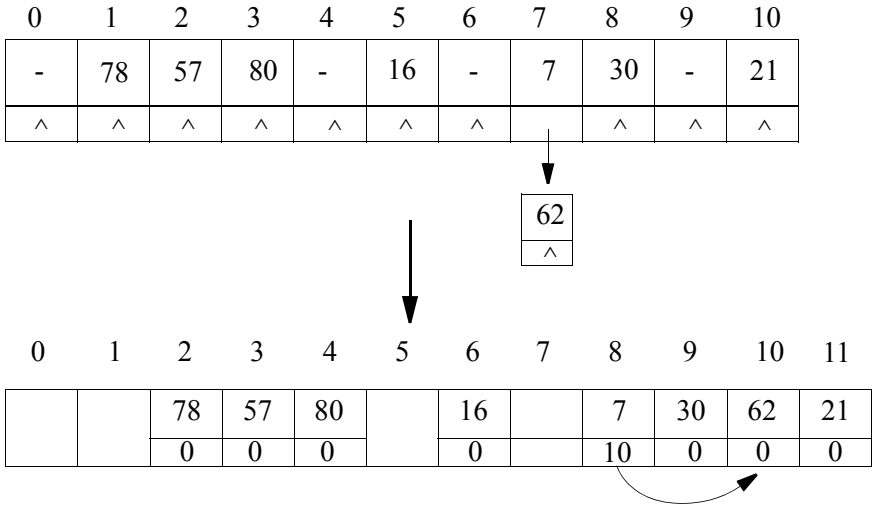
Geschlossene Hashverfahren:

Jede der Speicherzellen darf grundsätzlich nur mit der durch die Bucketkapazität b festgelegten maximalen Anzahl von Einträgen belegt werden.
Ist eine Speicherzelle bereits maximal belegt, so ist für weitere Schlüssel mit derselben Hashadresse eine spezielle **Kollisionsstrategie** anzuwenden.

Probleme: • die Gesamtanzahl der Schlüssel ist auf $m \cdot b$ beschränkt;
• in der Regel große Effizienzprobleme, falls die Hashtabelle 'fast voll' ist.

Direkte Verkettung mit Verschmelzen:

Wie bei der direkten Verkettung, nur dass die Listen innerhalb der Hashtabelle selbst gespeichert werden.
Hierdurch wird ein Teil des benötigten Speicherplatzes eingespart (nämlich der für das Array von Zeigern), falls $m \approx n$. Die Speicherverwaltung wird jedoch komplexer.



Sei: **class** Entry { **public boolean** frei;
public int key;
public int next; // *0 <= next <= m*
}

```
class ClosedHashtable{
    protected Entry hTable[];
    protected int m;
    protected int r;   // Anzahl besetzter Adressen
    void ClosedHashtable (int size)
        { hTable = new Entry[size+1];
         m = size;
         r = 0;
        hTable[0].frei = true;}
    // weitere Methoden}
```

Vorgehensweise beim Einfügen eines Schlüssels *k*:

- berechne die Hashadresse $h(k) + 1$
- wenn diese Adresse noch nicht belegt ist, so besetze sie mit *k*,
  im anderen Fall suche eine andere freie Adresse und speichere *k* dort.

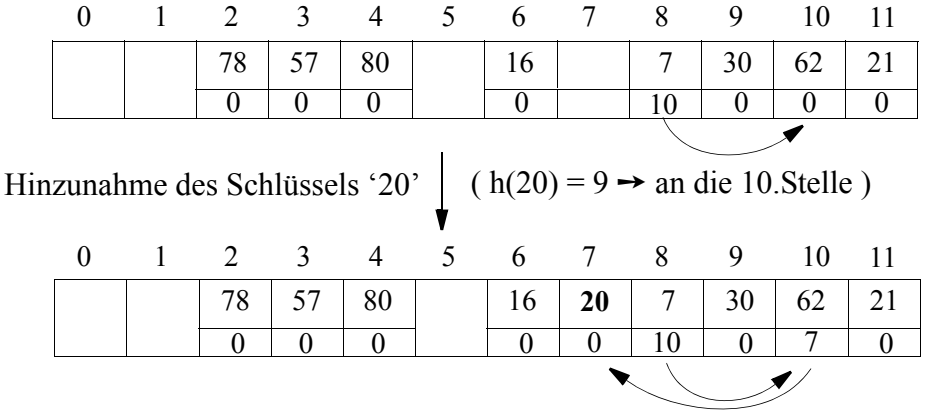
Methode für das Einfügen eines Schlüssels:

- Hierbei: Verwendung einer Hilfsvariablen *j* mit Initialisierung $j = m + 1$.
  → sucht die höchste freie Komponente in der Hashtabelle beim Einfügen.
- Erweiterung der Hashtabelle um eine Komponente; dabei gilt stets: T[0] ist frei.
  → notwendig um OVERFLOW der Hashtabelle zu erkennen.

```
public boolean insert (int k)
{  int i, j;
   boolean b;
   i = h(k) + 1;
   if (!hTable[i].frei)
      {b = false;
      j = m + 1;
      while (!b)
         {if (hTable[i].key == k) return false; // Schlüssel vorhanden
         else if (hTable[i].next != 0) i = hTable[i].next;
               else b = true;
         } // i mit hTable[i].next == 0 und hTable[i].key != k -> k einfügen
      do j = j - 1; while (!hTable[j].frei); // hTable[0] ist stets frei
      if (j == 0) return false; // overflow
      else hTable[i].next = j; i = j;
      } // end if
   hTable[i].frei = false;
   hTable[i].next = 0;
   hTable[i].key = k;
   return true;}
```

```
ClosedHashtable T = new ClosedHashtable(11);
```

Beispiel



Verschiedene Listen werden miteinander verschmolzen.

➞ die Suche ist damit häufig nicht nur auf Elemente einer Hashadresse beschränkt

➞ ungünstig für die Suchzeit

Offene Adressierung (*rehashing*):

Eine explizite Verzeigerung von Elementen wird vermieden.
Stattdessen: die *Adressberechnung* wird für überlaufende Elemente *iteriert*.

Lineares Sondieren ('linear probing'):

Betrachte ausgehend von $h(k)$ jeweils das Folgeelement:

$h(k)$, $h(k) + 1$, $h(k) + 2$, ..., m - 1, 0, 1, ... $h(k) - 1$ (*zyklische Sondierungsfolge*)

Für jeden Schlüssel k werden die Elemente der Hashtabelle in dieser zyklischen Sondierungsfolge besucht, bis der Schlüssel k oder eine leere Position gefunden wird.
Einfügen eines neuen Schlüssels: erfolgt nur falls r < m - 1

➞ r bezeichnet die Anzahl besetzter Adressen. ('voll': $r == m - 1$)

```
public boolean insert (int k)
{  int i;
   boolean b;

   if ( r == m - 1) return false; // overflow
   else
      {r = r + 1;
      i = h(k);
      if (!hTable[i].frei)
         {b = false;
         while (!b)
            {if (hTable[i].key == k) return false; // Schlüssel vorhanden
            else
               { i = i + 1;
               if ( i > m -1) i = i - m;
               if (hTable[i].frei) b = true;
               }
            }
         }      // nach der Schleife gilt hTable[i].frei == true
      hTable[i].frei = false;
      hTable[i].key = k;
      return true;
      }
}
```

Lineares Sondieren ist gut, falls die Hashtabelle nicht zu voll ist.

Ist die Hashtabelle dagegen annähernd voll

➞ lange Ketten besetzter Positionen wachsen mit höherer Wahrscheinlichkeit als kurze.

➞ ausgeprägte Tendenz zur Kettenbildung ➞ ungünstiges Suchverhalten.

Beispiel:

k ➞ Position 15 $\Leftrightarrow 11 \le h(k) \le 15$
k ➞ Position 8 $\Leftrightarrow h(k) = 8$
$\Rightarrow$ Einfügen in Position 15 ist 5 mal so wahrscheinlich wie in Position 8

allgemein: Einfügen an eine Stelle mit l besetzten Vorgängern ist $(l+1)$ mal so wahrscheinlich wie das Einfügen an eine isolierte Position.

Allgemein gibt man bei der offenen Adressierung eine Folge von Hashfunktionen vor: $h_0(k), h_1(k), h_2(k), \ldots, h_{m-1}(k)$. Diese sollte so gewählt sein, dass sämtliche Adressen 0 ... m-1 auftreten.
Für einen einzufügenden Schlüssel k werden nacheinander die entsprechenden Zellen der Hashtabelle inspiziert (sondiert), bis eine freie Zelle gefunden ist.
Diesselbe Strategie muss bei Suchoperationen angewandt werden.

Im Beispiel des linearen Sondierens gilt: $h_i(k) = (h(k) + i) \text{ MOD } m$, $0 \le i \le m-1$.

Problem: Einträge dürfen nicht einfach gelöscht werden, da Kollisionspfade anderer Elemente unterbrochen werden können!
Stattdessen werden entfernte Elemente nur als “entfernt” markiert.
➞ ungünstig für stark dynamische Anwendungen.

2.4.4 Externe Hashverfahren

Um sehr große Datenmengen zu organisieren, reicht der Hauptspeicher nicht mehr aus
➞ Organisation der Datensätze auf dem Sekundärspeicher
➞ externe Verfahren notwendig!

Wesentliche Eigenschaft: Bucketgröße $b > 1$

Ziel: Anzahl der Sekundärspeicherzugriffe (Zugriffe auf Buckets) gering halten
➞ gute Hashfunktion sehr wichtig!

In Abhängigkeit von der Kollisionsbehandlung unterscheiden wir:
- Verfahren mit Directory (= geschlossene Hashverfahren) und
- Verfahren ohne Directory (= offene Hashverfahren)

Wir beschränken uns hier auf ein Beispielverfahren ohne Directory.
(vgl. auch die Vorlesung: '*Index- und Speicherungsstrukturen für Datenbanksysteme*')

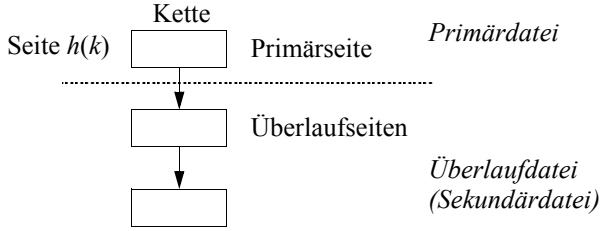
Das lineare Hashing (Litwin, 1980)

Zwei grundlegende Eigenschaften des linearen Hashing:
- die Größe der Datei (Hashtabelle) ist abhängig von der Anzahl der Schlüssel (*dynamisch*)
- Überlaufsätze einzelner Buckets ➞ **getrennte Verkettung** in einem *Überlaufbereich*

Die *Überlaufstrategie der getrennten Verkettung*:

➞ zwei verschiedene Seitentypen:
- *Primärseiten* ➞ *Primärdatei*
  ➞ Datenseiten (Buckets), deren Adressen von der Hashfunktion berechnet werden.
- *Überlaufseiten* ➞ *Überlaufdatei (Sekundärdatei)*
  ➞ speichern die *Überlaufsätze* der Primärdatei.

Es gilt:
(i) Überlaufseiten und Primärseiten sind verschieden.
(ii) jede Überlaufseite ist genau einer Hashadresse (einem Bucket) zugeordnet.
(iii) die Überlaufseiten derselben Primärseite werden verkettet.



➞ die Suche nach einem Überlaufsatz benötigt also mindestens 2 Seitenzugriffe (Primärseite + 1 Überlaufseite).
Entstehen lange Ketten von Überlaufseiten, so degenerieren die Suchzeiten.

Bis ca. 1978 ging man davon aus, dass
- die Größe der Primärdatei (Hashtabelle) *statisch* vorgegeben ist
- Überläufe nur durch Einführung von *Überlaufsätzen* gelöst werden können.

➞ schnelle Degenerierung der Suchzeiten, wenn die Primärseiten voll werden.

Jetzt: stattdessen eine *dynamische Erweiterung der Primärdatei*:
- wenn die Primärseiten zu voll werden ➞ Erweiterung der Datei
- neue Seiten werden über eine *neue Hashfunktion* adressiert

➞ man benötigt eine Folge von Hashfunktionen $h_0, h_1, h_2, \ldots$

Das Verfahren:

- zu Beginn ➞ kleine Primärdatei (z.B. $m = 5$); Hashfunktion: $h_0: K \to \{0, 1, 2, 3, 4\}$.
  - die ersten b Schlüssel ➞ sicher in Primärseiten.
  - zunehmende Schlüsselanzahl ➞ Wahrscheinlichkeit für Überlaufsätze steigt.

- *Idee*: Vergrößerung (**Expansion**) der Primärdatei um jeweils ein Bucket, falls eine vorgegebene Auslastung der Primärseiten erreicht ist. (**Kontrollfunktion**)

  Definition: **Belegungsfaktor**

$$bf = \frac{\text{Anzahl der Schlüssel}}{\text{Anzahl Schlüssel, die in Primärseiten passen}} = \frac{\text{Anzahl der Schlüssel}}{b \cdot \text{Anzahl der Primärseiten}}.$$

  Es wird vorgeschlagen: Expansion der Datei, falls $bf > 0{,}8$.

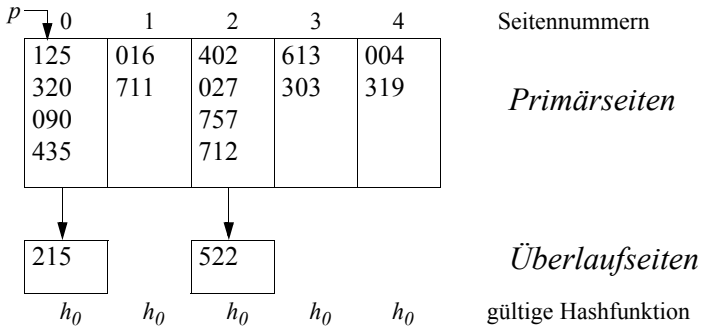
- die *Expansion der Primärdatei*
  - jeweils *eine* Primärseite wird *in zwei* Seiten aufgespalten.
  - der Split der Primärseiten erfolgt *zyklisch* in einer *fest vorgegebenen Reihenfolge*.
  - ein *Expansionszeiger p* zeigt auf die als nächstes zu splittende Seite.
  - für gesplittete Seiten gilt eine *neue Hashfunktion* ($h_i \to h_{i+1}$)
  - nach einem Seitensplit ➞ Setzen des Expansionszeigers auf die nächste Seite.
  - sind alle Seiten gesplittet ➞ die Datei wurde verdoppelt

    ➞ die neue Hashfunktion gilt für die gesamte Datei; das Verfahren beginnt von neuem.

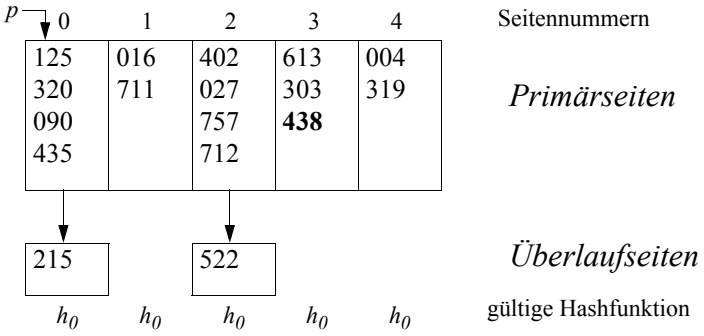
Beispiel:

Ausgangssituation:

- Datei mit 16 Datensätzen auf 5 Primärseiten (Buckets) der Seitengröße $b = 4$.
- Folge von Hashfunktionen: $h_0(k) = k \text{ MOD } 5$, $h_1(k) = k \text{ MOD } 10$, ...
- Aktuelle Hashfunktion: h_0.
- Belegungsfaktor $bf = \frac{16}{20} = 0{,}8$
- Schwellenwert für den Belegungsfaktor = 0,8
- Expansionszeiger p zeigt auf Seite 0.

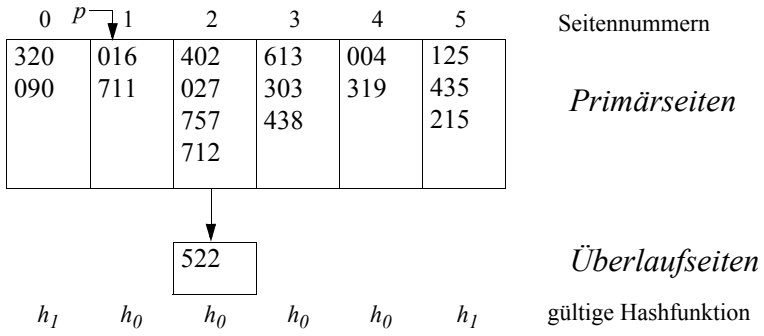


Einfügen eines neuen Datensatzes mit Schlüssel 438:



- Der Belegungsfaktor übersteigt den Schwellenwert: $bf = \frac{17}{20} = 0{,}85 > 0{,}8$

  ➞ Expansion durch Split der Seite 0 auf die Seiten 0 und 5.



Welche Datensätze von Seite 0 nach Seite 5 umgespeichert werden, bestimmt die Hashfunktion h_1:

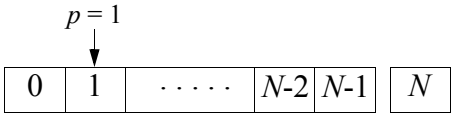
➞ Sätze mit $h_1(k) = 5$ werden umgespeichert, Sätze mit $h_1(k) = 0$ bleiben

Anschließend wird der Expansionszeiger p um 1 heraufgesetzt.

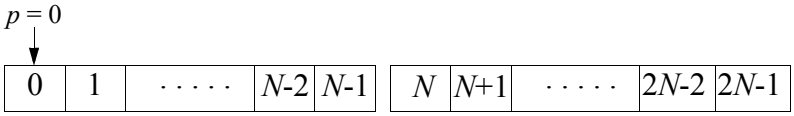
Das Prinzip der Expansion durch Splits:

Ausgangssituation:

nach dem ersten Split:



nach der Verdoppelung der Datei:



Anforderungen an die Hashfunktionen

Die Folge von Hashfunktionen $\{h_i\}$, $i \geq 0$, muss die folgenden Bedingungen erfüllen:

Bereichsbedingung:

$$h_L: K \rightarrow \{0, 1, \ldots, N \cdot 2^L - 1\}\ , L \geq 0.$$

Splitbedingung:

$$h_{L+1}(k) = h_L(k) \quad \text{oder} \quad h_{L+1}(k) = h_L(k) + N \cdot 2^L, \quad L \geq 0$$

Der *Level L* gibt dabei an, wie oft sich die Datei bereits vollständig verdoppelt hat.

Beispiel für eine mögliche Folge von Hashfunktionen:

$h_L(k) = k$ MOD $(N \cdot 2^L)$ erfüllt die Bereichs- und die Splitbedingung.

Zusätzlich erwünschte Eigenschaft:

Für $p = 0$ (nach einer Verdoppelung) besitzen alle Buckets die gleiche Wahrscheinlichkeit, einen neu eingefügten Datensatz aufzunehmen. (*Ideale Hashfunktion*)

Wichtige Eigenschaften des linearen Hashing

- die Splitreihenfolge der Seiten ist fest vorgegebenen.
  ➞ Seiten, die überlaufen werden erst mit einer gewissen Verzögerung behandelt.
- der Adressraum wächst **linear** an und ist gerade so groß wie nötig.
- unter der Voraussetzung einer guten Hashfunktion ist der Prozentsatz der Überlaufsätze gering.
- gute Suchzeiten für gleichverteilte Schlüssel.

Das lineare Hashing mit partiellen Erweiterungen (Larson, 1980)

Beobachtung: Hashverfahren sind am effizientesten, wenn die Schlüssel möglichst gleichmäßig auf die Seiten der Datei verteilt sind.

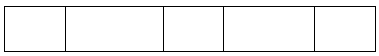
Problem: Während jeder Verdoppelung der Datei gilt:
Der durchschnittliche Belegungsfaktor bereits gesplitteter Seiten ist nur halb so hoch wie der durchschnittliche Belegungsfaktor noch nicht gesplitteter Seiten.
➞ keine gleichmäßige Verteilung der Datensätze auf die Buckets.

Verbesserung: Einführung ***partieller Expansionen:***
➞ das Verdoppeln der Datei erfolgt nicht in einem, sondern in mehreren Durchläufen
➞ Serie von $n_0 \geq 2$ partiellen Expansionen.

Vorgehensweise des **linearen hashing mit partiellen Erweiterungen*:*** (für $n_0 = 2$)

Ausgangssituation:

Datei mit *2N* Seiten



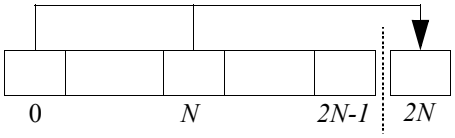
logisch unterteilt in *N* Paare (j, j+*N*) für j = 0, 1, ... ,*N*-1

1. partielle Expansion

- nach Einfügen von Schlüsseln wird aufgrund der Kontrollfunktion eine Expansion nötig.
  ➞ Expansion der Datei um die Seite *2N*:

Etwa $1/3$ der Sätze aus den Seiten 0 und N werden nach Seite $2N$ umgespeichert.

Datei mit $2N+1$ Seiten

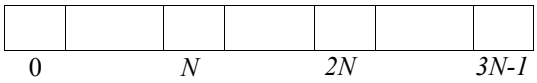


logisch unterteilt in die verbleibenden $N-1$ Paare $(j, j+N)$ für $j = 1, \ldots, N-1$
sowie das Tripel $(0, N, 2N)$

- bei weiteren Einfügungen wird, wenn es die Kontrollfunktion verlangt, für $j = 1, 2, \ldots, N-1$ jeweils das Paar $(j, j+N)$ um die Seite $j+2N$ expandiert.

  ➞ nach N Schritten: die Datei ist von $2N$ auf $3N$ Seiten (auf das 1,5-fache) angewachsen:

  Datei mit $3N$ Seiten

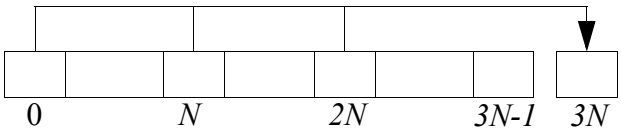
  logisch unterteilt in N Tripel $(j, j+N, j+2N)$ für $j = 0, 1, \ldots, N-1$

2. Partielle Expansion:

- Zunächst Expansion der Datei um die Seite $3N$:

  Etwa je $1/4$ der Sätze aus den Seiten 0, N und $2N$ werden umgespeichert auf die Seite $3N$.

  Datei mit $3N+1$ Seiten

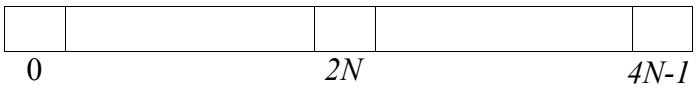
  logisch unterteilt in $N-1$ Tripel $(j, j+N, j+2N)$ für $j = 1, 2, \ldots, N-1$
  und ein Quadrupel $(0, N, 2N, 3N)$

- nachfolgend werden für $j = 1, 2, \ldots, N-1$ die Tripel $(j, j+N, j+2N)$ jeweils um die Seite $j+3N$ expandiert, wobei ca. $1/4$ der Sätze aus den Seiten $j, j+N, j+2N$ umgespeichert werden.

Resultat:

- die Datei hat sich auf $4N$ Seiten verdoppelt:

  Datei mit $4N$ Seiten

  ➞ dies ist die neue Ausgangssituation für die nächste Verdoppelung der Datei.

Analyse der partiellen Expansionen

Für das lineare Hashing gilt während der 1. bzw. 2. partiellen Expansion:

➞ Der durchschnittliche Belegungsfaktor einer bereits gesplitteten Seite beträgt $2/3$ bzw. $3/4$ des durchschnittlichen Belegungsfaktors einer noch nicht gesplitteten Seite.
(*verglichen mit dem Faktor* $1/2$ *ohne partielle Expansionen (* $n_0 = 1$*)*)

Kontrollfunktion

Larson schlägt vor, als Kontrollfunktion die *Speicherplatzausnutzung* zu wählen.

$$\text{Speicherplatzausnutzung} = \frac{\text{Anzahl der Schlüssel}}{\text{Gesamtkapazität von Primär- und Überlaufseiten}}$$

Expansionsregel

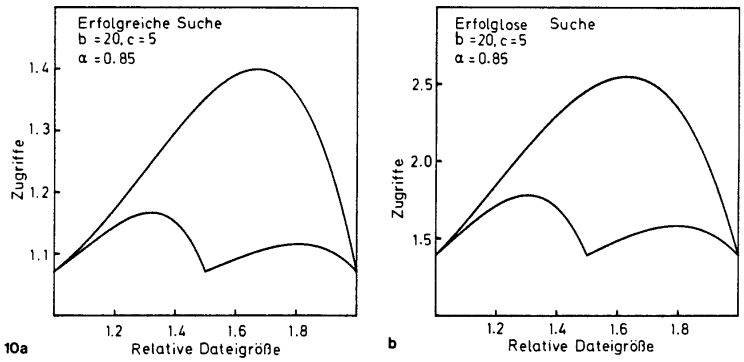
Bei jeder Einfügung wird die Speicherplatzausnutzung überprüft. Falls ihr Wert größer ist als ein vorgegebener Schwellenwert α, $0 < \alpha < 1$
➞ die Datei wird um eine weitere Seite expandiert.

➞ die Speicherplatzausnutzung liegt annähernd konstant bei α, falls nur Einfügungen auftreten.

Leistungsverhalten

Sei *b* die Kapazität einer Primärseite und *c* die Kapazität einer Überlaufseite.

Graphik ➞ Anzahl benötigter Seitenzugriffe bei erfolgreicher bzw. bei erfolgloser Suche für $n_0 = 1$ und $n_0 = 2$.



Beobachtungen:

- die Effizienz ist dann am größten, wenn die Belegung aller Ketten nahezu gleich ist.
- zyklisches Verhalten ➞ Länge eines Zyklus = 2 · Länge des vorangegangenen Zyklus.

Vergleich: durchschnittliche Anzahl benötigter Zugriffe für $n_0 = 1$, $n_0 = 2$ und $n_0 = 3$ mit den zugrundeliegenden Parametern $b = 20$, $c = 5$, $\alpha = 0{,}85$.

	$n_0 = 1$	$n_0 = 2$	$n_0 = 3$
Erfolgreiche Suche	1,27	1,12	1,09
Erfolglose Suche	2,12	1,58	1,48
Einfügen	3,57	3,21	3,31
Entfernen	4,04	3,53	3,56

Folgerung:

$n_0 = 2$ partielle Expansionen stellen den besten Kompromiss dar zwischen

- Effizienz des Suchens,
- Effizienz der Update-Operationen (Einfügen und Entfernen) und
- Komplexität des Programmcodes