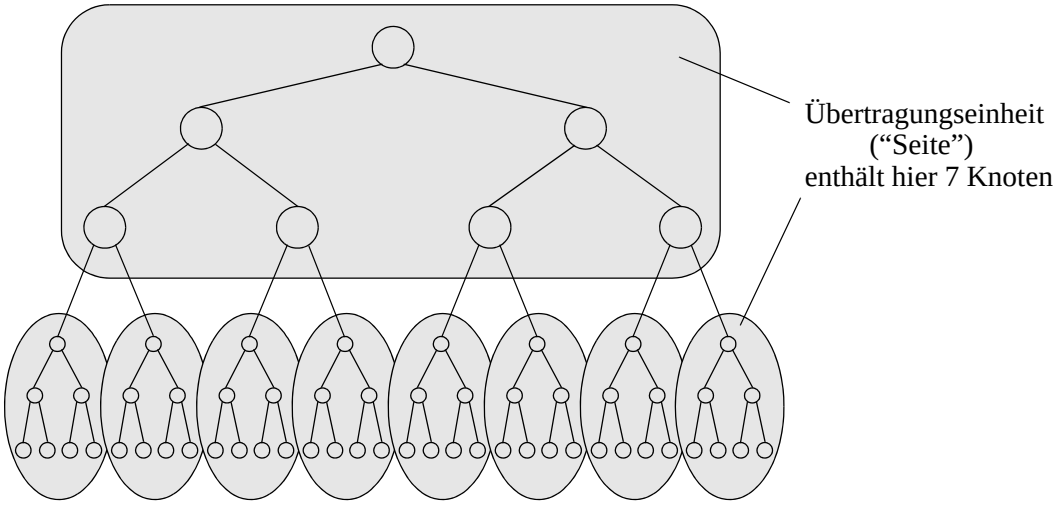
Beispiel:

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: Zusammenfassen mehrerer binärer Knoten zu einer Seite



Beispiel für einen B-Baum:

99 Knoten in einer Seite ➠ 100-fache Verzweigung:

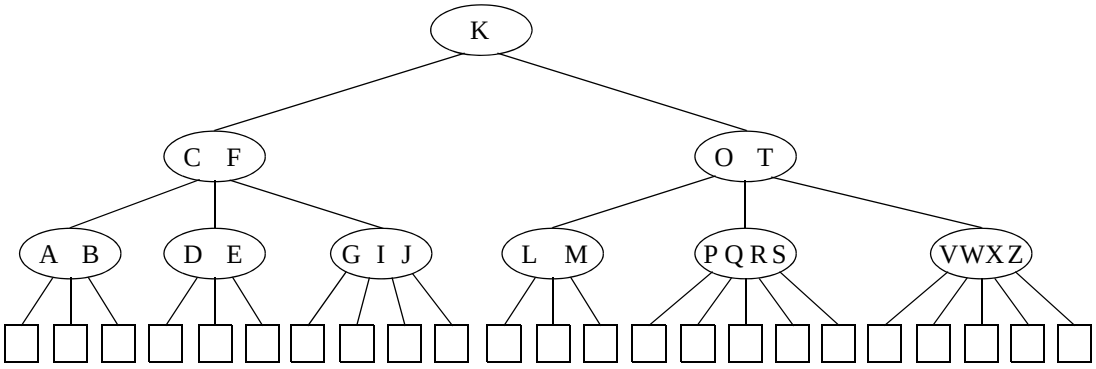
➠ $\log_{100}(10^6) = 3$ Plattenspeicherzugriffe.

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

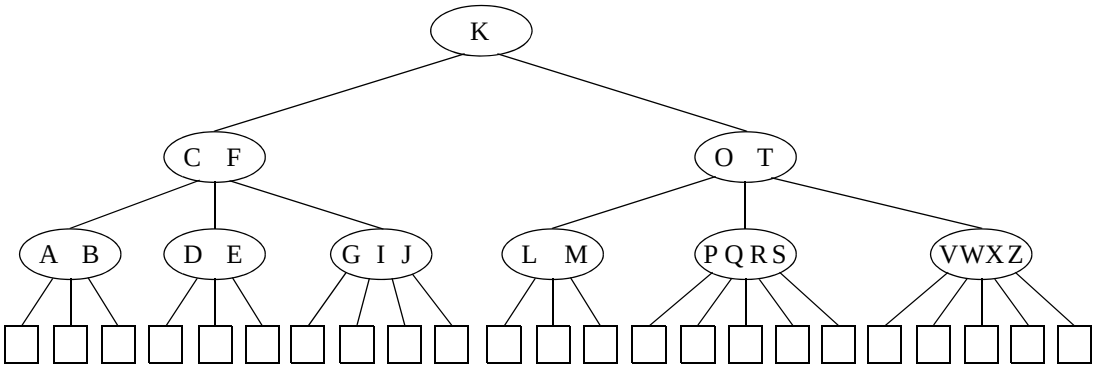
Definition: B-Baum der Ordnung *m* (Bayer und McCreight (1972))

(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



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

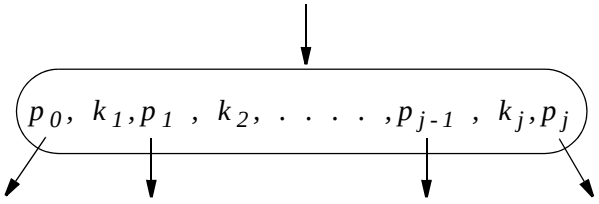
$$k_{h+1} = n + 1 \geq 2(m+1)^{h-1}, \text{ d.h. } h \leq 1 + \log_{m+1}\left(\frac{n+1}{2}\right).$$

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

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

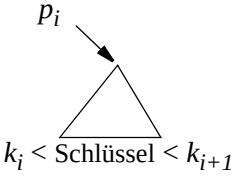
Beobachtung: Jeder Knoten (außer der Wurzel) ist mindestens mit der Hälfte der möglichen Schlüssel gefüllt. Die Speicherplatzausnutzung beträgt also mindestens 50%!

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.

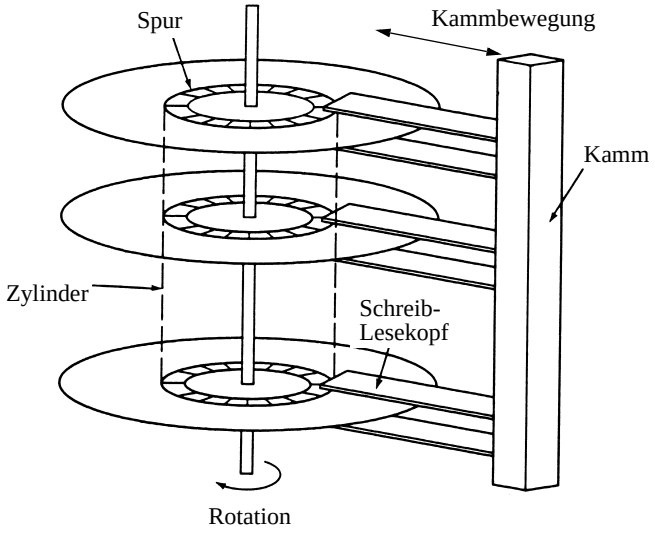
```
class Entry { public int key;
              public Page son;
              ... // Konstruktor..
              }

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:

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

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 (eines HDD) ein:

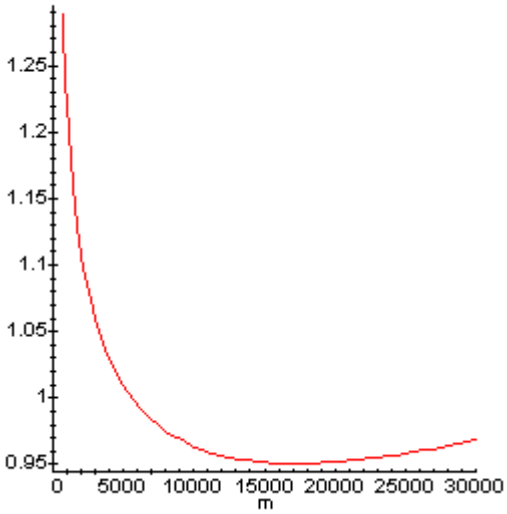
$a = 0{,}012s,\ a + d \approx a = 0{,}012s = 12\ ms$

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

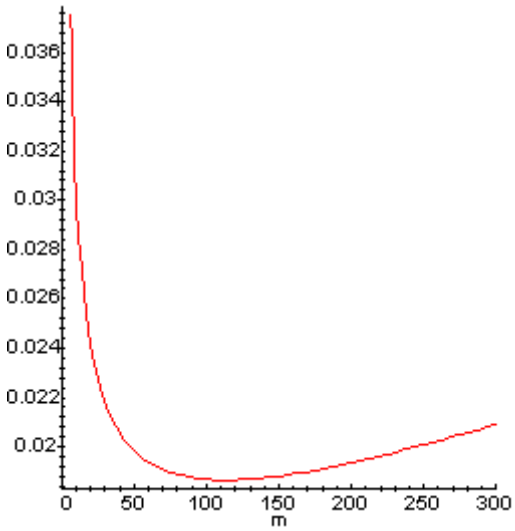
Damit ist die folgende Funktion $\overline{MS(m)}$ zu minimieren:

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

$\overline{MS(m)}$ 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 216 KByte. Für die exemplarischen Werte der Solid-State-Disk ergibt sich eine „optimale" Seitengröße von 1,5 KB (für m = 125).

Einfügen in B-Bäumen:

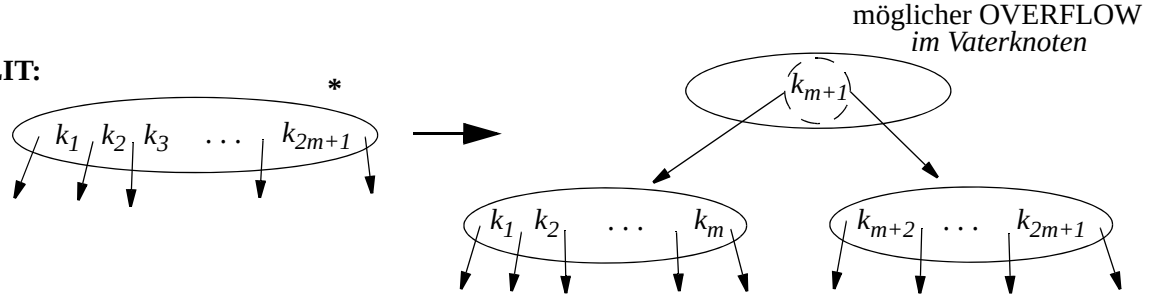
Zunächst wird der Knoten *K* gesucht, in den der neue Schlüssel einzufügen ist. Dieser ist stets ein Blatt. Der Schlüssel wird in *K* an der entsprechenden Stelle eingefügt.
Sei *s* die Anzahl der Schlüssel in *K* nach dem Einfügen:

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.

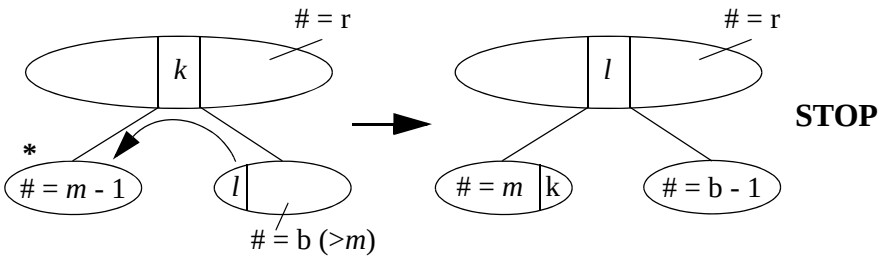
Sei s die Anzahl der Schlüssel in K nach dem Entfernen:

Fall 1: $s \geq m$: ➙ **STOP**

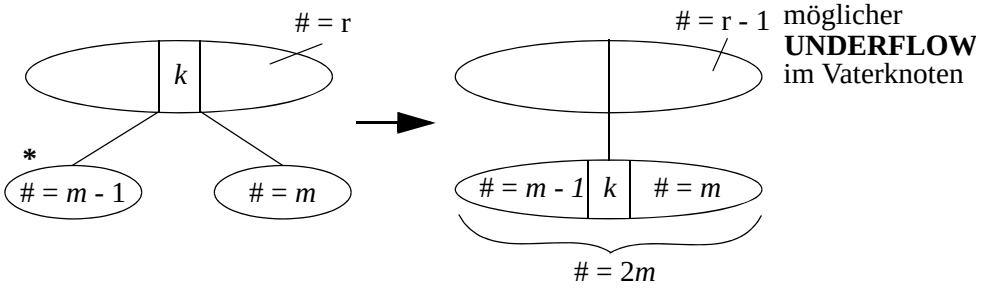
Fall 2: $s = m - 1$: ➙ **UNDERFLOW**

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.: $600 \leq m \leq 900$ (s.o.).

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

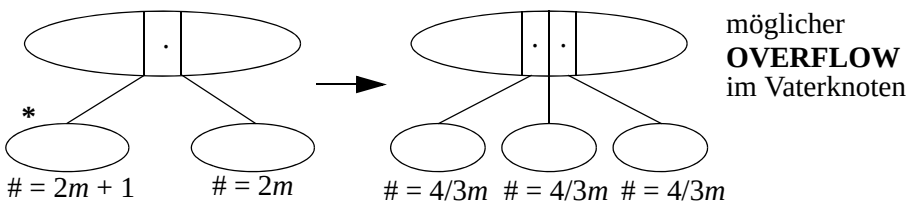
Denn:
- bei jeder Aufspaltung wird mindestens ein zusätzlicher Knoten geschaffen
- Aufspalten der Wurzel ➞ 2 zusätzliche Knoten
- wenn ein B-Baum mehr als einen Knoten hat, dann ist die Wurzel mindestens einmal aufgespalten worden
- dem ersten Knoten des B-Baums geht keine Aufspaltung voraus.

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.

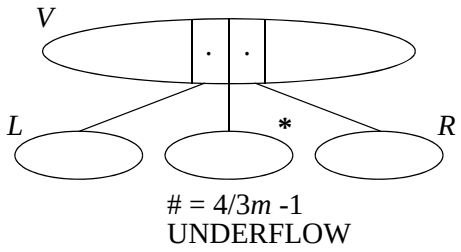
(1) Jeder Knoten außer der Wurzel enthält mindestens $\frac{4}{3}m$, höchstens $2m$ Schlüssel.
(2) Die Wurzel enthält mindestens einen, höchstens $\frac{8}{3}m$ Schlüssel.
(3) Ein Knoten mit k Schlüsseln hat genau k+1 Söhne.
(4) Alle Blätter befinden sich auf demselben Level.

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:

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

Entfernen in B*-Bäumen

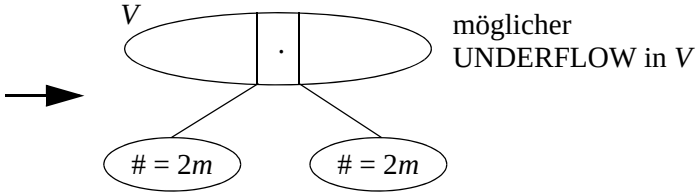


Fall 1: R hat mehr als $\frac{4}{3}m$ Schlüssel ➞ Ausgleichen von * und R

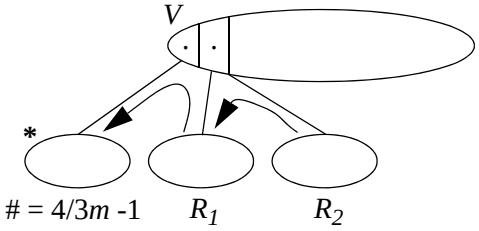
Fall 2: R hat $\frac{4}{3}m$ Schlüssel:

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

2.2 L hat $\frac{4}{3}m$ Schlüssel:



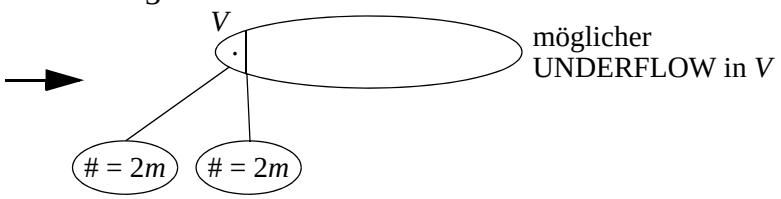
Fall 3: * hat nur einen direkten Bruder, aber weitere indirekte Brüder:



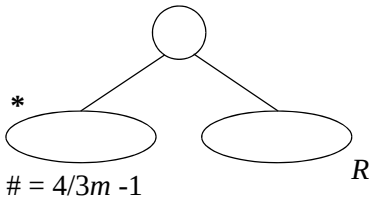
3.1 Falls R_1 mehr als $\frac{4}{3}m$ Schlüssel besitzt ➞ Ausgleichen von * und R_1

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.

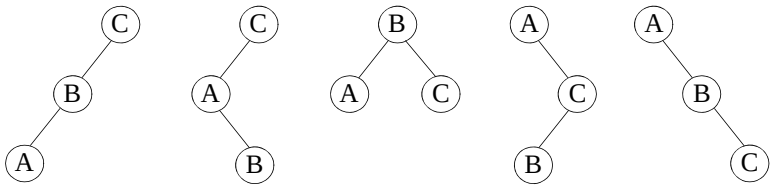
Problem: Wie finden wir unter diesen Voraussetzungen einen "optimalen" Suchbaum aus den $\binom{2n}{n} / (n+1) \approx 4^n / (\sqrt{\pi} \cdot n^{3/2})$ verschiedenen Suchbäumen für n Schlüssel?

Beispiel:

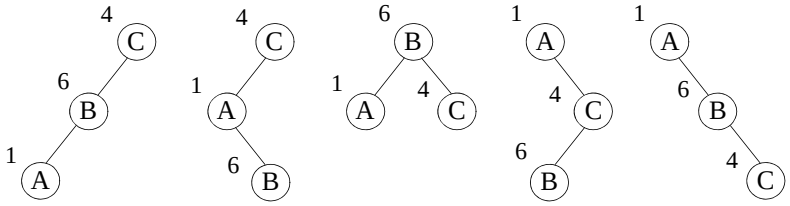
Sei $n = 3$;

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 = $(\alpha + \beta)$ · (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 $< k_{i+1}$)

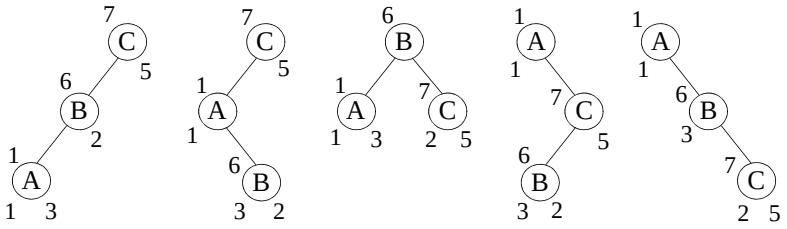
(o.E. $k_0 = -\infty$, $k_{n+1} = \infty$)

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 $q_0 = 1, q_1 = 3, q_2 = 2, q_3 = 5$.

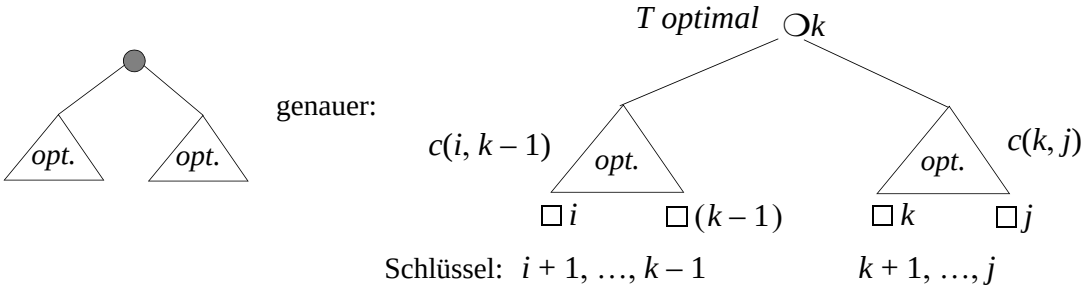


Optimalitätskriterium: Alle **Teilbäume** eines optimalen Suchbaumes sind **optimal**.

→ **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:

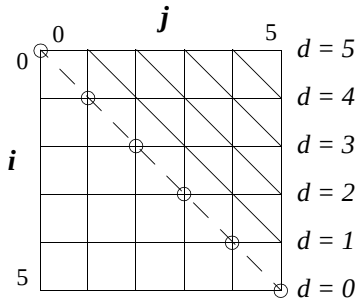
$c(i, i) = 0$ ➞ der Baum besteht aus $\square i$ (*leerer Baum*).

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

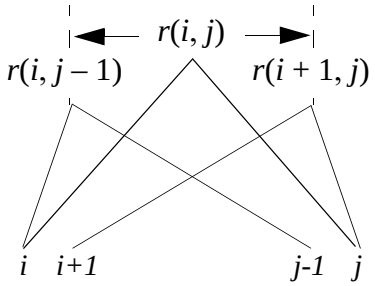
$$\left(\sum_{d=1}^{n} d^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}\right)$$

➞ **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.

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



➙ 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 $O(n^2)$.

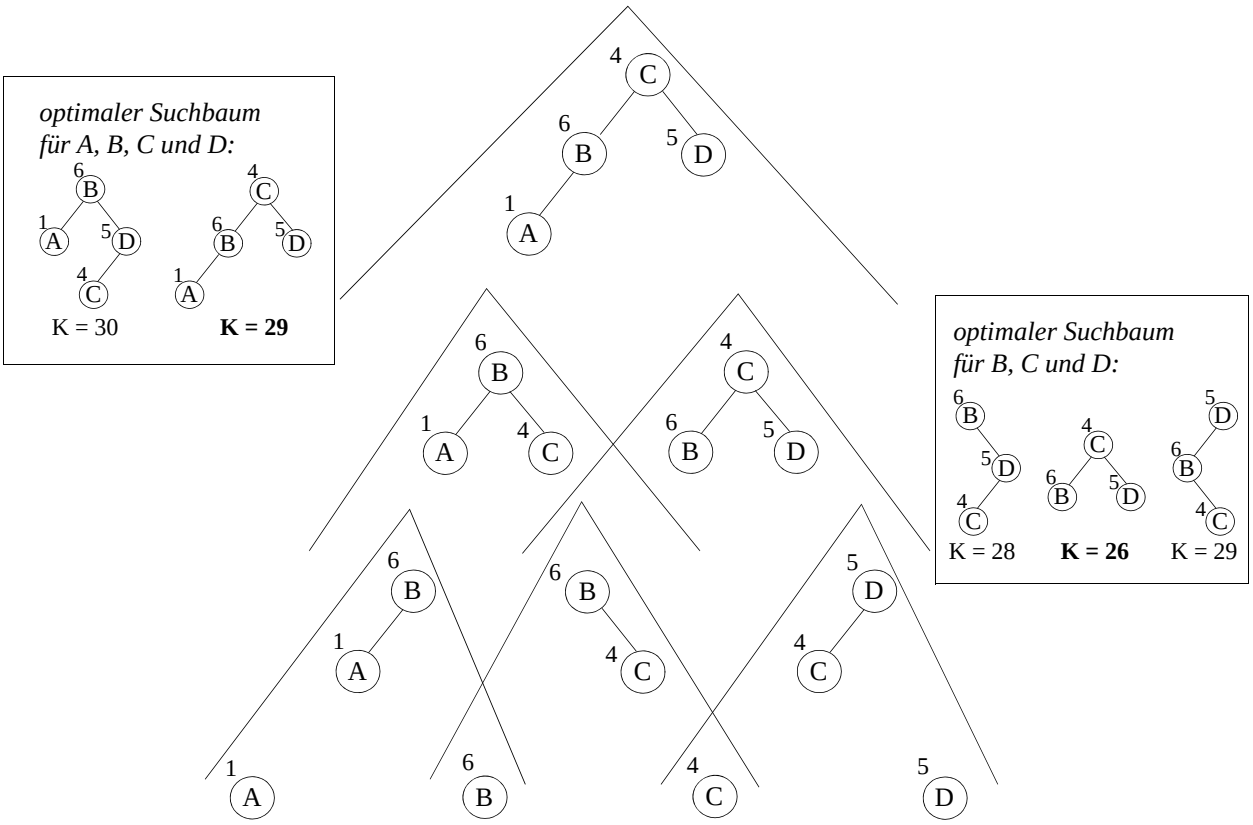
Abschließendes Beispiel:

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)