Suchen nach einem Schlüssel (FIND): Wie beim Suchbaum.

Einfügen (INSERT):
1. Füge den Schlüssel x wie beim Suchbaum als neues Blatt in den AVL-Baum ein.
2. Gehe den Suchpfad zurück und ändere die Balancefaktoren. Entsteht hierbei -1 oder +1, so fahre beim Vaterknoten fort; entsteht 0, so brich ab; entsteht -2 oder +2, so führe genau eine (R-, LR-, L- oder RL-) Rotation aus und brich ab (R=rechts, L=links).

Löschen (DELETE):
1. Finde den Schlüssel x im AVL-Baum und entferne seinen Knoten wie bei einem Suchbaum (inorder-Vorgänger bzw. -Nachfolger).
2. Gehe den Weg zur Wurzel zurück ab der untersten Stelle, an der eine Veränderung stattfand, und korrigiere entlang des Pfads die Balancefaktoren mittels Rotationen (evtl. muss man bis zur Wurzel alle Balancefaktoren ändern).

3.2.4.6 Ergebnis: Alle drei Operationen erfordern auch im schlechtetsten Fall höchstens O(log(n)) Schritte. [log ist der Logarithmus zur Basis 2.]
Der Grund hierfür: Ein AVL-Baum mit n Knoten kann höchstens die Tiefe $1{,}4404 \cdot \log(n)$ besitzen. Dies lässt sich folgendermaßen beweisen:

Man stelle fest, wieviel Knoten in einem AVL-Baum der Tiefe t maximal und minimal liegen können.
Maximal können es 2^t-1 Knoten sein (klar).
Sei m_t die Anzahl der Knoten, die sich mindestens in einem AVL-Baum der Tiefe t befinden müssen, dann gilt: $m_0=0$, $m_1=1$ und für alle $t \geq 2$: $m_t = 1 + m_{t-1} + m_{t-2}$. (Wurzel plus linker Unterbaum und rechter Unterbaum, die Folge der Zahlen ist 0, 1, 2, 4, 7, 12, 20, ...).
Die Lösung der Gleichung lautet: $m_t = F_{t+2} - 1$, wobei F_t die t-te Fibonacci-zahl ist. Dies ist durch Induktion leicht zu beweisen.

Wiederholung aus der Mathematik: Die *Fibonaccizahlen* sind definiert durch: $F_0=0$, $F_1=1$, $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$ (Folge: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...).
Man nehme an, F_k erfüllt eine Gleichung der Form $F_k = a \cdot x^k$, dann muss $x^k = x^{k-1} + x^{k-2}$ gelten, d.h., man muss die Gleichung $x^2 = x + 1$ lösen. Deren Lösungen sind c_1 und c_2 (siehe nächste Folie).
Da auch deren Linearkombinationen Lösungen darstellen, erhält man für die Fibonaccizahlen die Formel $F_k = a_1 \cdot c_1^k + a_2 \cdot c_2^k$. Mit den Anfangsbedingungen $F_0=0$ und $F_1=1$ ergibt sich $a_1 = -a_2 = f$, woraus man $F_k = f \cdot (c_1^k - c_2^k)$ erhält.
Wegen $|c_2| < 1$ ist F_k stets die nächste natürliche Zahl zu $f \cdot c_1^k$.

Einschub: Fibonaccizahlen

$F_0 = 0,\ F_1 = 1$ und $F_k = F_{k-1} + F_{k-2}$ für alle $k \geq 2$.

$$F_k = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^k - \left(\frac{1-\sqrt{5}}{2} \right)^k \right) =: f \cdot (c_1^k - c_2^k)$$

mit $c_1 = \left(\frac{1+\sqrt{5}}{2} \right) \approx 1.618035,$

$c_2 = \left(\frac{1-\sqrt{5}}{2} \right) \approx -0.618035, \quad f = \frac{1}{\sqrt{5}} \approx 0.447213$

und F_k = nächste natürliche Zahl zur Zahl $f \cdot c_1^k$.

Sei also ein AVL-Baum mit n Knoten der Tiefe t gegeben.

$n \geq m_t = F_{t+2} - 1 = f \cdot c_1^{t+2} - 1$

$\approx 0.447213 \cdot 1.618035 \cdot 1.618035 \cdot 1.618035^t - 1$

$\approx 1.17082 \cdot 1.618035^t - 1$

Es folgt mit $1/\log(1.618035) \approx 1.4404$:

$t \leq \log((n+1)/1.17082) / \log(1.618035)$

$\approx 1.4404 \cdot \log(n+1) - \log(1.17082) / \log(1.618035)$

$\approx 1.4404 \cdot \log(n)$ [Diese Abschätzung ist recht genau.]

Satz 3.2.4.7: Ein AVL-Baum mit n Knoten besitzt höchstens die Tiefe $1.4404 \cdot \log(n)$.

Hinweis: Messungen ergaben, dass diese maximale Tiefe in der Praxis fast nie auftritt und sich die Tiefe der AVL-Bäume meist nur wenig von log(n) unterscheidet.

Da wir häufiger die Bäume, die sehr gleichverzweigt sind und nur Blätter auf dem Level log(n) oder eines weniger besitzen, verwenden, so wollen wir ihnen einen Namen geben, nämlich "ausgeglichene" Bäume:

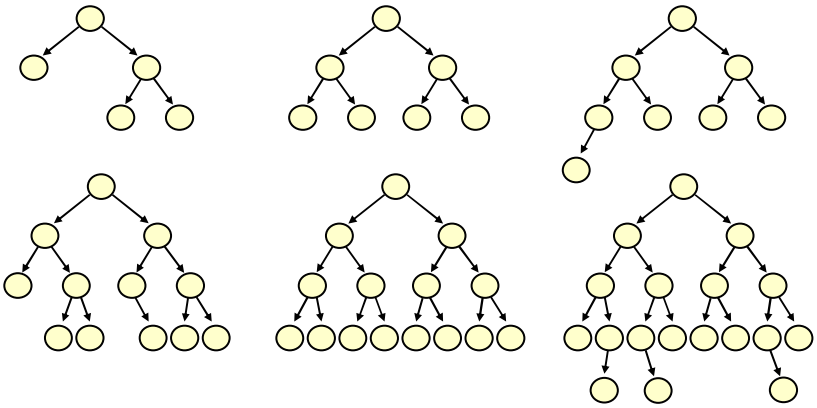
Definition 3.2.4.8:

Ein nicht-leerer, k-närer Baum (vgl. Definition 3.2.2.1, 10) heißt ausgeglichener Baum (der Tiefe r), wenn es eine natürliche Zahl r gibt, so dass jeder Knoten, der mindestens einen null-Zeiger enthält, das Level r-1 oder r besitzt und es mindestens ein Blatt der Tiefe r gibt.

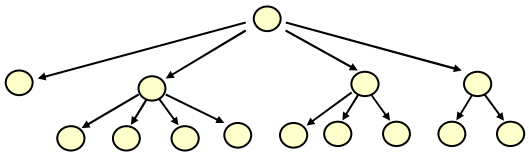
Man kann diese Definition leicht von k-nären auf beliebige geordnete Bäume übertragen. Die B-Bäume des nächsten Abschnitts bilden solch ein Beispiel.

Sofern es Knoten mit null-Zeigern in verschiedenen Leveln gibt, ist in einem ausgeglichenen Bäumen das Level r-1 komplett mit Knoten besetzt. Überschüssige Knoten können sich dann nur noch auf dem nächsten Level r befinden.

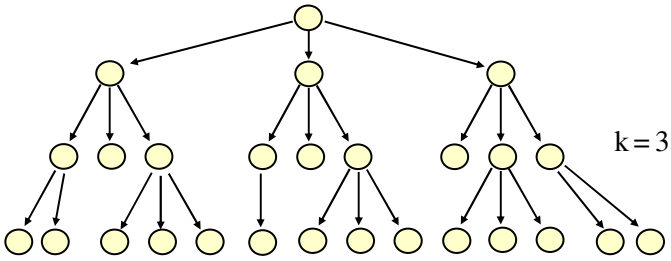
Beispiele für k = 2 (also binäre Bäume) und für r = 3, 4 und 5:



Beispiele für k > 2 (null-Zeiger wurden nicht eingetragen, wodurch zu jeder Skizze mehrere k-näre Bäume gehören können):



k = 4, r = 3



k = 3, r = 4

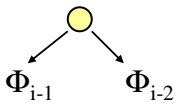
3.2.4.9 Fibonacci-Bäume

Φ_0 ist der leere Baum, Φ_1 ist der einknotige Baum.

Definition:

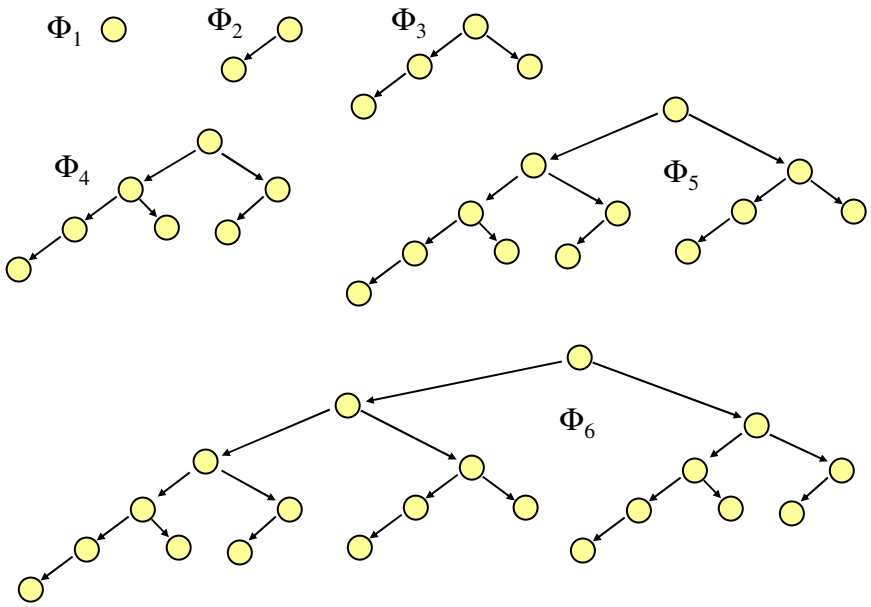
(1) Φ_0 und Φ_1 sind die Fibonaccibäume der Ordnung 0 und 1.

(2) Wenn Φ_{i-1} der Fibonaccibaum der Ordnung i-1 und Φ_{i-2} der Fibonaccibaum der Ordnung i-2 sind ($i \geq 2$), dann ist

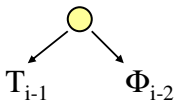


der Fibonaccibaum Φ_i der Ordnung i.

Der Fibonaccibaum der Ordnung i ist der "dünnste" AVL-Baum der Tiefe i, also der AVL-Baum mit minimal vielen Knoten zu gegebener Tiefe i, vgl. Nachweis von 3.4.2.6.
Der Baum Φ_i besitzt genau $m_i = F_{i+2} - 1$ Knoten.



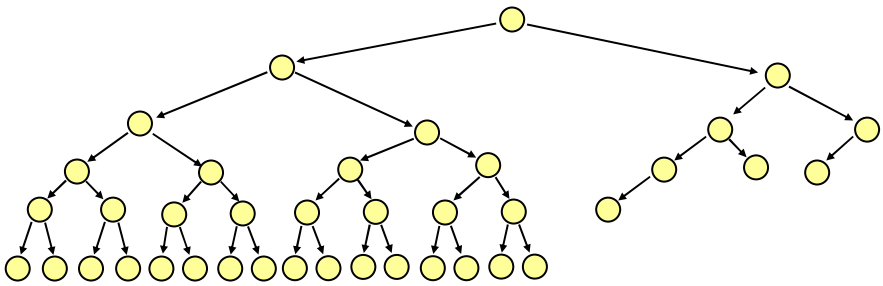
Nun kann man folgenden AVL-Baum der Tiefe i konstruieren:



wobei T_{i-1} der AVL-Baum der Tiefe i-1 mit maximal vielen Knoten ist. T_{i-1} besitzt $2^{i-1}-1$ und Φ_{i-2} besitzt $m_{i-2} = F_i - 1$ Knoten.

Der Quotient $2^{i-1}/F_i$ wächst exponentiell in i, woraus folgt, dass es für jede vorgegebene reelle Zahl $\alpha \in (0, ½]$ AVL-Bäume gibt, die nicht α-gewichtsbalanciert (siehe Definition 3.2.4.1) sind.

Der folgende Baum ist ein AVL-Baum. Man erkennt, dass seine Tiefe durch log(n)+2 beschränkt ist; die Knoten sind jedoch sehr unterschiedlich auf die beiden Unterbäume der Wurzel verteilt.



!! Keine weiteren Folien zu AVL-Bäumen !!

Die Details und die Programme für Einfügen und Löschen sind in der Literatur bestens beschrieben. In der Vorlesung verwenden wir die Unterlagen der Informatik-II-Vorlesung des SS 01 von Prof. Plödereder, S. 118-143 (in der Fachschaft erhältlich).

Datentypen für AVL-Bäume, vgl. Skript Plödereder, S. 132:

```
type Knotentyp;
type Baumtyp is access Knotentyp;
type Balancetyp is (-1, 0, 1);
type Knotentyp is record
    Schluessel: integer;
    Balance: Balancetyp;
    UBLinks, UBRechts: Baumtyp;
    end record;
```

Hinweis: Herr Plödereder verwendet statt (-1,0,1):
type Balancetyp is (LinksL, Neutral, RechtsL);

Gliederung des Kapitels

3.2 Suchverfahren

~~3.2.1 Suchen in sequentiellen Strukturen~~

~~3.2.2 Bäume und (binäre) Suchbäume~~

~~3.2.3 Optimale Suchbäume~~

~~3.2.4 Balancierte Bäume, AVL-Bäume~~

3.2.5 B-Bäume

3.2.6 Digitale Suchbäume (Tries)

3.2.5 B-Bäume (für externe Speicherung)

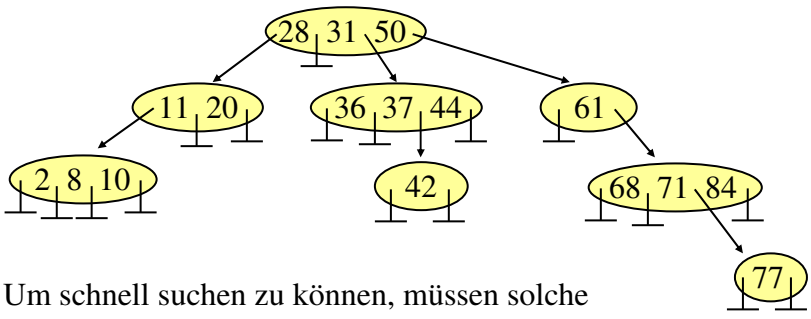
AVL-Bäume eignen sich gut als Darstellung von Mengen, wenn sich die gesamte Menge im Hauptspeicher befindet. Liegen die Elemente dagegen auf einem Hintergrundspeicher, so muss man bei jedem Übergang zu einem Kindknoten auf diesen externen Speicher zugreifen. Heute ist der Hintergrundspeicher meist eine Festplatte mit einer Zugriffszeit von im Mittel 5 Millisekunden. Hat man einen AVL-Baum der Tiefe 20 (dies entspricht einer Menge mit 1 Million Elementen), so werden alleine 100 Millisekunden für den Zugriff benötigt, während die durchgeführten 20 Vergleiche weniger als eine Millisekunde erfordern. Der Rechner verbringt seine Zeit daher zu über 99% mit Warten und ist nach rund 101 Millisekunden mit der Suche fertig.

Ziel ist daher eine Datenstruktur, die mit möglichst wenigen externen Zugriffen die gesuchten Daten findet bzw. einfügt bzw. löscht.

Nahe liegend ist eine gute Mischung aus baumartiger und sequenzieller Suche: Man holt bei jedem Zugriff - sagen wir - 500 Werte in den Hauptspeicher und grenzt den Bereich, wo der gesuchte Schlüssel zu finden ist, nicht wie beim AVL-Baum um den Faktor 2, sondern um den Faktor 500 ein.

Bei einem Datenbestand von 1 Million Werten sind dann nur noch drei Zugriffe erforderlich. Zusammen mit dem (internen) Durchlaufen der 500 Werte braucht man jetzt nur noch eine Gesamtlaufzeit von unter 20 Millisekunden, wobei 15 Millisekunden aus Warten bestehen.

Wir betrachten nun also Bäume, in deren Knoten sich nicht nur ein Schlüssel, sondern ein (sortiertes) k-Tupel von Schlüsseln befindet. Um effizient suchen zu können, durchläuft man dieses k-Tupel und folgt je nachdem, zwischen welchen Elementen der gesuchte Schlüssel liegt, einem Zeiger in den richtigen Unterbaum. Beispiel:



Um schnell suchen zu können, müssen solche Bäume folgende Suchbaumeigenschaft erfüllen.

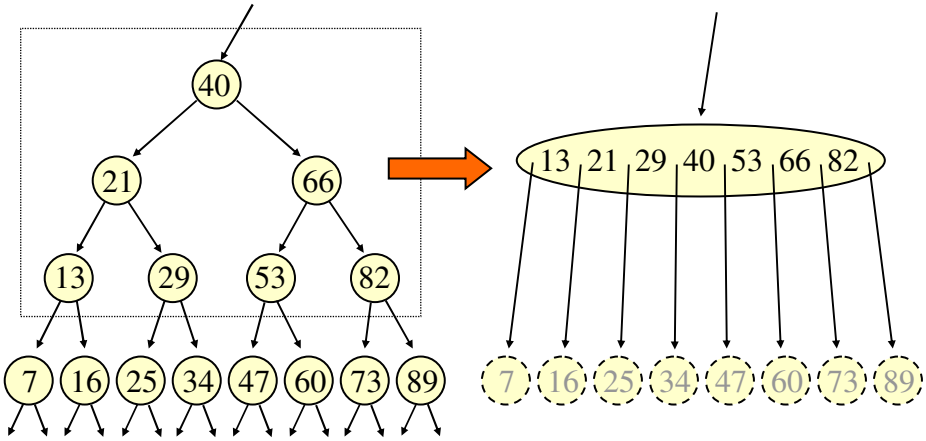
Definition 3.2.5.1: Verallgemeinerte Suchbaumeigenschaft

Gegeben sei ein geordneter Baum. In jedem Knoten steht ein nicht-leeres sortiertes Tupel von Schlüsseln einer geordneten Menge. Für alle Knoten u muss gelten:
Sind $s_1, s_2, ..., s_k$ die sortierten Schlüssel von u ($s_1 < s_2 < ... < s_k$), dann sind alle Schlüssel im Unterbaum des ersten Sohns kleiner als s_1, alle Schlüssel im Unterbaum des zweiten Sohns sind größer oder gleich s_1 und kleiner als s_2, alle Schlüssel im Unterbaum des i-ten Sohns sind größer oder gleich s_{i-1} und kleiner als s_i (für $i = 2, 3, ..., k$), und alle Schlüssel im Unterbaum des (k+1)-ten Sohns sind größer oder gleich s_k.
Ein geordneter Baum mit dieser Eigenschaft heißt (allgemeiner) Suchbaum.

Definition 3.2.5.2:

Es sei $m \geq 2$ eine natürliche Zahl. Ein geordneter nicht-leerer Baum, in dem jeder Knoten eine sortierte Folge von Schlüsseln aus einer geordneten Menge enthält, heißt **B-Baum** der Ordnung m, wenn für alle Knoten gilt:

1. Jeder Knoten enthält höchstens 2m Schlüssel.
2. Die Wurzel enthält mindestens einen Schlüssel.
3. Jeder andere Knoten enthält mindestens m Schlüssel.
4. Ein Knoten mit k Schlüsseln besitzt entweder genau k+1 Söhne oder keinen Sohn.
5. Alle Blätter (=Knoten ohne Söhne) besitzen das gleiche Level.
6. B erfüllt die verallgemeinerte Suchbaumeigenschaft.

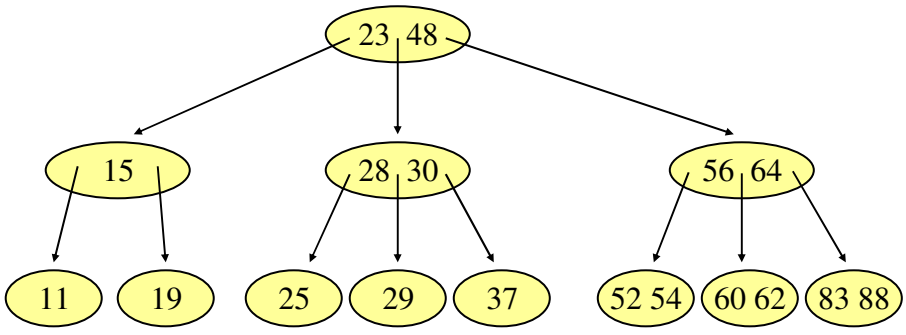


Ausschnitt aus einem binären Suchbaum

Zusammenfassung zu einem B-Baum-Knoten, dessen Inhalt linear durchlaufen wird

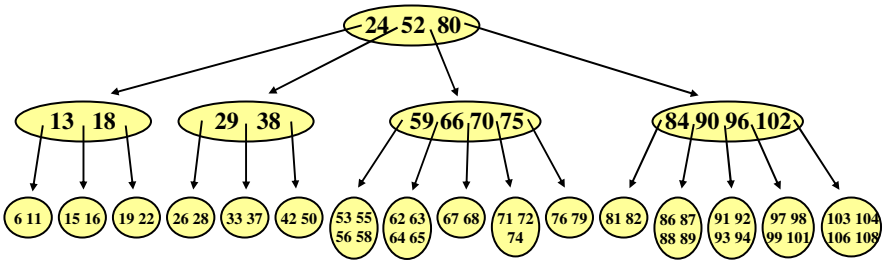
Beispiel für einen B-Baum der Ordnung 1

Man beachte, dass alle Blätter das gleiche Level besitzen.



Hinweis: B-Bäume der Ordnung 1 heißen auch 2-3-Bäume.

Beispiel für einen B-Baum der Ordnung 2:



Übliche Ordnungen liegen in der Praxis zwischen 128 und 4096.

Tiefe eines B-Baums:
Da jeder Knoten (außer der Wurzel) mindestens m+1 Söhne besitzt, muss die Tiefe bei n Schlüsseln kleiner als $\log_{m+1}(n)+1$ sein. Andererseits kann sie nicht weniger als $\log_{2m+1}(n)$ betragen.

Hinweis: Die Terminologie ist in der Literatur unterschiedlich. Oft verwendet man auch "2m" oder "2m+1" als die Ordnung des B-Baums. Vergewissern Sie sich daher bei B-Bäumen stets über die zugrunde liegenden Definitionen.

Die Operationen auf B-Bäumen verändern die Struktur durch zwei Maßnahmen: Aufspalten (Splitten) eines Knotens in zwei Knoten und Verschmelzen zweier Knoten zu einem Knoten.

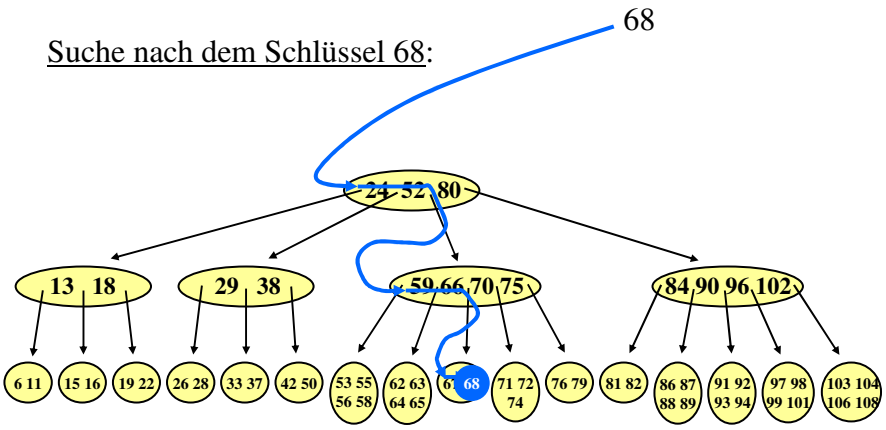
In vielen Anwendungen speichert man die Daten zwar im B-Baum, möchte die eigentlichen Inhalte beim Löschen und bei manchen Implementierungen auch beim Einfügen aber nicht im Baum verschieben. Dann legt man alle Daten in die Blätter des Baums und errichtet hierüber einen Baum aus Zeigern. Da ein B-Baum nur an der Wurzel wächst und schrumpft (siehe später), wird eine solche Struktur bei B-Bäumen automatisch aufrecht erhalten.
Ein B-Baum, bei dem alle Daten nur in den Blättern liegen, nennt man B*-Baum.

Wir betrachten nun die drei Operationen Suchen, Einfügen und Löschen.

3.2.5.3 Suchen in einem B-Baum

Das Suchen erfolgt wie oben angegeben: Man durchlaufe das sortierte Tupel des Knotens und folge dem "richtigen" Zeiger entsprechend der Suchbaumeigenschaft. Der Zeitaufwand ist proportional zur Tiefe des Baums. Man muss jedoch noch die sortierten Listen der Schlüssel in jedem betrachteten Knoten durchlaufen. Dies kostet mindestens m und höchstens 2m Vergleiche (durch Intervallschachtelung kommt man auch mit log(2m) Vergleichen aus, wenn die Schlüssel in einem array abgelegt sind). Insgesamt muss man daher mit bis zu $2m \cdot \log_{m+1}(n)$ Vergleichen rechnen, je nach Implementierung weniger, vgl. nächste Folie.

Suche nach dem Schlüssel 68:

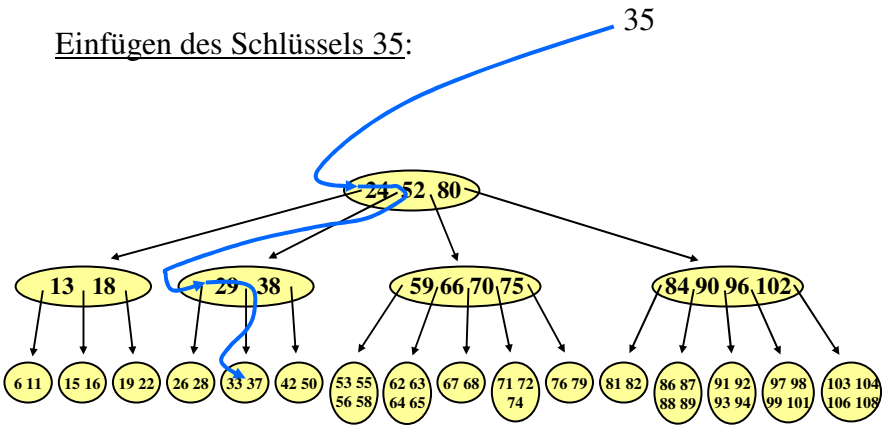


Für n Elemente im Baum gilt offenbar:
Zahl der Vergleiche ≤ 2m · Tiefe des Baumes ≤ 2m · $\log_{m+1}(n)$.
Für m = 512 ist dies beispielsweise für eine Wertemenge bis 134 Millionen Elemente durch 6m = 3072 Vergleiche und 3 Zugriffe auf den externen Speicher beschränkt.

3.2.5.4 Einfügen in einen B-Baum

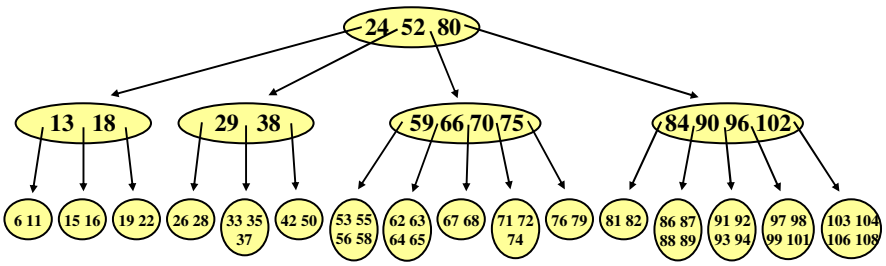
Beim Einfügen ermittelt man zunächst das Blatt, in das der neue Schlüssel s einzutragen ist. Befinden sich weniger als 2m Schlüssel in diesem Blatt, so füge man den neuen Schlüssel s sortiert ein. Anderenfalls liegt ein Überlauf im Knoten vor und das Blatt muss in zwei Blätter mit je m Schlüsseln aufgespalten werden, wobei der mittelste der 2m+1 Schlüssel an den Vaterknoten weiterge-reicht und dort eingetragen wird. Eventuell muss nun auch der Vaterknoten aufgespalten werden usw. Hierbei kann von unten nach oben ein Aufspalten bis zur Wurzel hin erfolgen. Wird die Wurzel aufgespalten, so wird eine neue Wurzel mit genau einem Schlüssel erzeugt.
Das Vorgehen wird in den folgenden Beispielen illustriert. Der Aufwand ist wie bei der Suche, jedoch muss ggf. der Pfad bis zur Wurzel zurückverfolgt werden. Hinzu kommt das Einsortieren der Schlüssel in die Knoten.

Einfügen des Schlüssels 35:

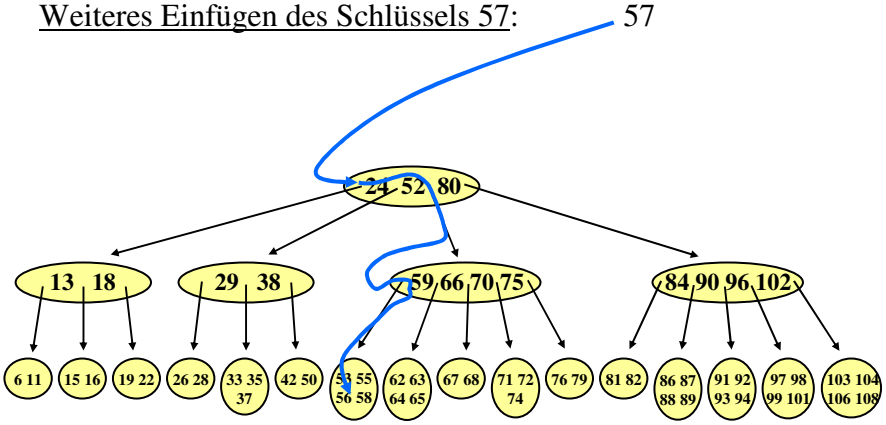


Es ist noch Platz, also wird der Schlüssel 35 sortiert hier eingetragen.

Einfügen des Schlüssels 35:

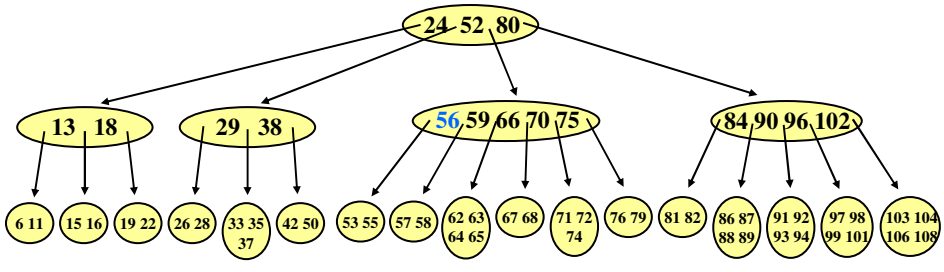


Weiteres Einfügen des Schlüssels 57: 57



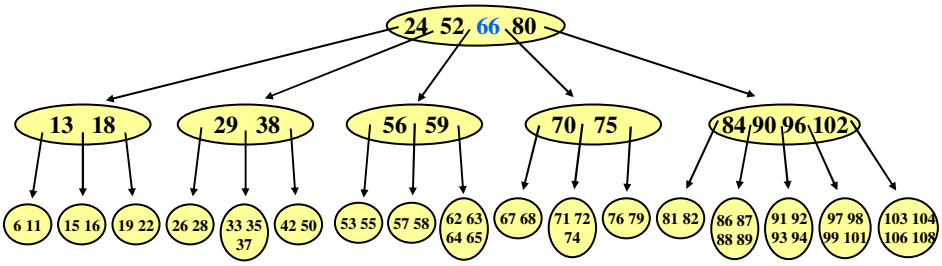
Beim Einfügen der 57 läuft der Knoten über, da er jetzt 5 Schlüssel enthält. Also wird er in zwei Knoten aufgespalten mit den Inhalten "53 55" bzw. "57 58". Der mittlere Wert "56" wird zum Vaterknoten hinauf gereicht.

Weiteres Einfügen des Schlüssels 57:



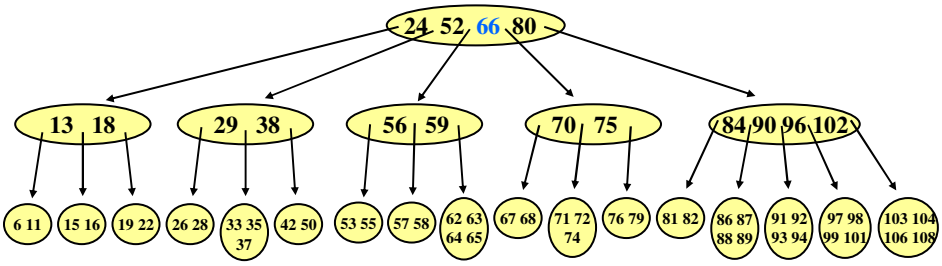
Beim Einfügen der 56 läuft aber nun der Vaterknoten über. Also wird er in zwei Knoten aufgespalten mit den Inhalten "56 59" bzw. "70 75". Der mittlere Wert "66" wird zu seinem Vaterknoten (dies ist hier die Wurzel) hinauf gereicht.

Weiteres Einfügen des Schlüssels 57:



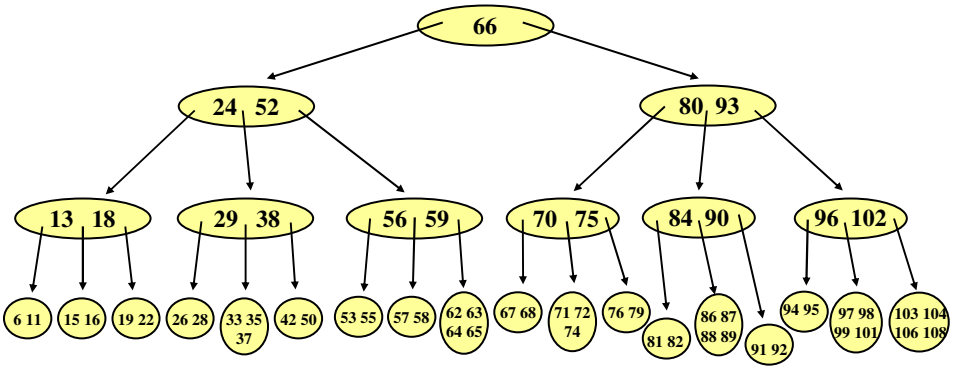
Die Wurzel darf (wie jeder Knoten in einem B-Baum der Ordnung 2) bis zu 4 Schlüssel besitzen, folglich ist dieser Baum das Ergebnis, wenn der Schlüssel 57 eingefügt wurde.

Weiteres Einfügen des Schlüssels 95:



Die 95 muss in den mit "91 92 93 94" beschrifteten Knoten eingetragen werden. Dieser läuft über und wird in zwei Knoten mit den Inhalten "91 92" und "94 95" aufgespalten; der Schlüssel "93" wird nach oben gereicht. Der Vaterknoten wird auch aufgepalten und reicht den Schlüssel "93" weiter nach oben. Auch die Wurzel läuft nun über, wird gespalten und reicht den Schlüssel 66 weiter. Dieser wird neue Wurzel des Baumes. Man hat also folgenden um ein Level größeren B-Baum der Ordnung 2 erhalten:

Ergebnis nach Einfügen der Schlüssel 35, 57 und 95:



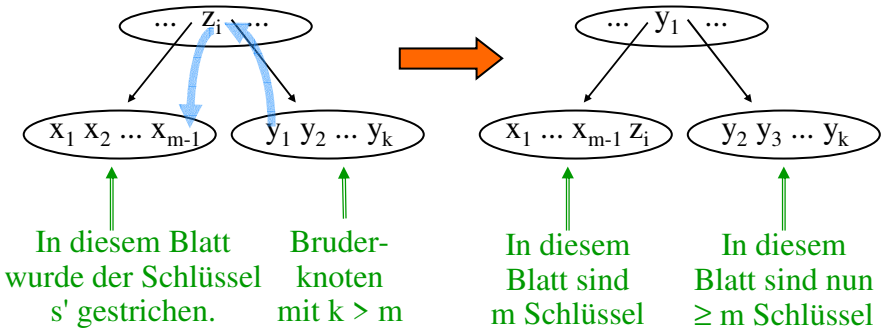
Man beachte, dass die Blätter hierbei stets auf dem gleichen Level liegen. Ein B-Baum kann nur an der Wurzel wachsen (und schrumpfen), während AVL-Bäume an den Blättern wachsen.

3.2.5.5 Löschen in einen B-Baum

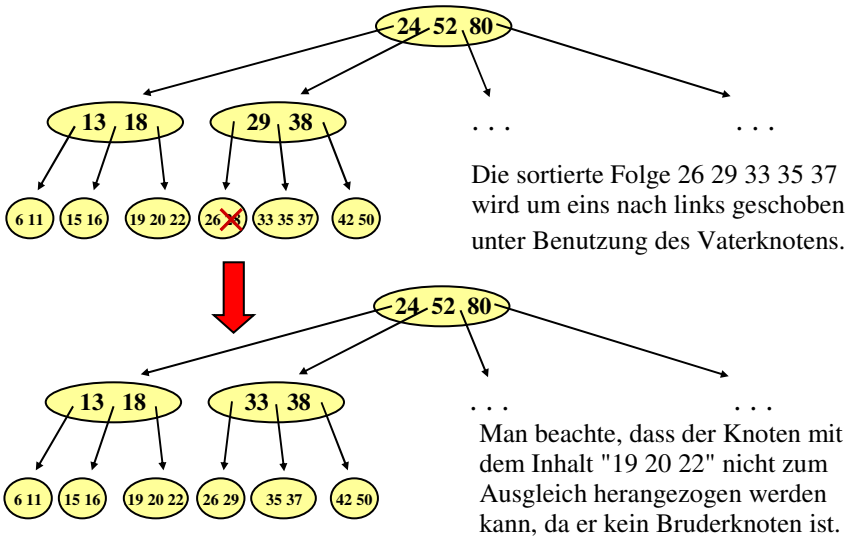
Beim Löschen wird zunächst der Knoten, in dem der gesuchte Schlüssel s steht, ermittelt. Ist dieser Knoten kein Blatt, so wird der Schlüssel s durch seinen Inorder-Nachfolger s' ersetzt (wie findet man diesen?). Da der Inorder-Nachfolger in einem Blatt steht, muss nun dieser Schlüssel s' im Blatt gelöscht werden.

Fall 1: Besitzt das Blatt mindestens m+1 Schlüssel, so wird der Schlüssel s' gelöscht und man ist fertig.
Fall 2: Besitzt das Blatt genau m Schlüssel "es liegt hier ein "Unterlauf" vor), so prüft man, ob ein Bruderknoten mit mindestens m+1 Schlüsseln exis-tiert. Ist dies der Fall, so führt man einen Ausgleich unter Verwendung des zugehö-rigen Schlüssels des Vaterknotens durch. Hierbei genügt eine Verschiebung von 2 Schlüsseln.

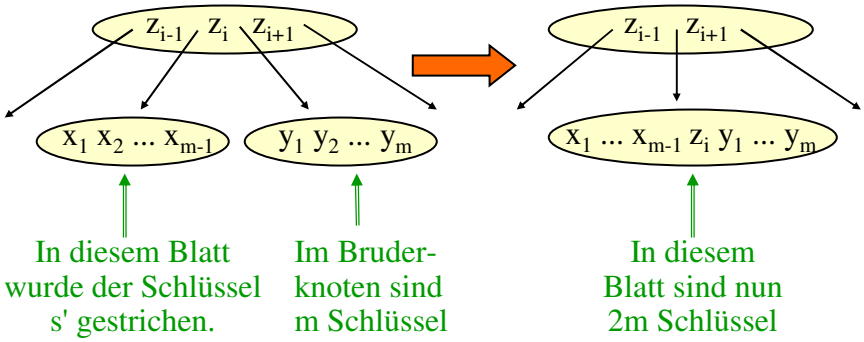
Genauer: Das Blatt besitzt m Schlüssel und mindestens ein Bruderknoten besitzt mindestens m+1 Schlüssel. Dann wird der zugehörige Schüssel des Vaterknotens in das Blatt geschoben und an seine Stelle wird der nächstgelegene Schlüssel aus dem Bruderknoten gesetzt. Skizze:



Beispiel zu Fall 2: Lösche den Schlüssel 28



Fall 3: Das Blatt besitzt m Schlüssel und alle Bruderknoten besitzen ebenfalls m Schlüssel. Dann wird das Blatt mit seinem rechten Bruderknoten verschmolzen unter Einbeziehung des zugehörigen Schlüssels im Vaterknoten (falls das Blatt der rechteste Sohn des Vaterknotens ist, dann nimm den links daneben liegenden Bruderknoten):



Fall 3 (Fortsetzung): In diesem Fall wird die Zahl der Schlüssel im Vaterknoten verringert. Wende daher rekursiv auf den Vaterknoten die Fälle 1 bis 3 an.
Hierbei kann es geschehen, dass jedes Mal Fall 3 auftritt und schließlich ein Schlüssel aus der Wurzel entfernt wird. Wenn die Wurzel hierbei noch mindestens einen Schlüssel behält, so ist man fertig. Falls die Wurzel aber nur einen Schlüssel besaß, so wird die Wurzel leer und der einzige, neu entstandene verschmolzene Knoten unter ihr wird zur neuen Wurzel des Baumes. Genau in diesem Fall wird die Tiefe des Baumes um 1 verringert.

Da der B-Baum höchstens die Tiefe $\log_{m+1}(n)+1$ besitzt, erfordert also auch das Löschen nur O(log(n)) Schritte.

Hinweise:

Bei der Implementierung kann man in jedem Knoten ein array [1..2*m] für die Schlüssel und ein array [0..2*m] für die Söhne mitführen. Zusätzlich ist eine natürliche Zahl "Anzahl" zu speichern, die die Zahl der Schlüssel angibt, sowie eine Boolesche Variable für die Eigenschaft "Blatt".
Ein B-Baum ist stets zu mindestesn 50% gefüllt. Wie viel Speicherplatz wird tasächlich ausgenutzt? Man wird 75% erwarten, jedoch zeigt eine theoretische Analyse, dass es im Mittel ca. 69% sind.
In der Praxis wurde durch Messungen bestätigt, dass B-Bäume meist nur zu zwei Drittel gefüllt sind. Man sollte daher das Einfügen von Schlüsseln so implementieren, dass das Aufspalten von Knoten möglichst lange vermieden wird (z.B., indem man die Bruderknoten einbezieht, wie es beim Löschen geschieht). Selbst nachdenken!

!! Keine weiteren Folien zu B-Bäumen !!

Weitere Hinweise zu B-Bäumen finden sich in den Unterlagen der Informatik-II-Vorlesung des SS 01 von Prof. Plödereder, S. 150-170 (in der Fachschaft erhältlich).

Datentypen für B-Bäume, vgl. auch Skript Plödereder, S. 160; ein Knoten ist hier eine "Seite":

```
type Seite;
type SeitePtr is access Seite;
type Item is record Schlüssel: SchlüsselTyp;
                    Inhalt: InhaltTyp;
                    Zeiger: SeitePtr;
                    end record;
type Seite is record Anzahl: Natural;
                    Blatt: Boolean;
                    linkeSeite: SeitePtr;
                    Items: array (1..2*m) of Item;
                    end record;
```