Mehrwegbäume

T. Härder, Universität Kaiserslautern

Binäre Suchbäume eignen sich vor allem zur Darstellung von Suchstrukturen oder Zugriffspfaden für Datenelemente in einstufigen Speichern (Hauptspeicher). Selbst bei einem Datenbestand von 10^6 Elementen bietet ein AVL-Baum für die direkte Suche mit etwa 20 Vergleichs- und Verzweigungsoperationen noch einen extrem schnellen Suchweg, da für solche interne Operationen nur Hauptspeicherzugriffe (kleiner 100 ns) erforderlich sind.

Bei der Organisation von Daten in mehrstufigen Speichern muß ein Baumknoten erst in den Hauptspeicher gebracht werden, bevor eine Vergleichs- und Verzweigungsoperation abgewickelt und damit die Adresse des Nachfolgerknotens im Suchbaum bestimmt werden kann (Bild 1). Da der Datentransport gewöhnlich in Einheiten fester Länge (Blöcke, Seiten) erfolgt, ist es günstig, die Suchbäume auf Externspeichern so auf die Seiten abzubilden, daß mit einem physischen Zugriff (Seitentransport) möglichst viel „Suchinformation“ zum Hauptspeicher übertragen wird.

Typische Seitenlängen liegen heute zwischen 2 K und 8 K Bytes. Für das wahlfreie Aufsuchen und den nachfolgenden Transport einer Seite zum Hauptspeicher werden bei Magnetplatten heute durchschnittlich etwa 10 ms benötigt. Wegen der Zugriffszeitcharakteristika der verschiedenartigen Speicher einer zweistufigen Speicherhierarchie tut sich eine „Zugriffslücke“ zwischen Hauptspeicher (elektronischer Speicher) und Externspeicher (magnetischer Speicher) auf, die momentan etwa einen Faktor von $>10^5$ ausmacht und wegen der Geschwindigkeitssteigerung der elektronischen Speicher tendenziell noch größer wird. Ein Zugriff auf eine Datenseite, die sich bereits im Hauptspeicher befindet, ist also um den Faktor $>10^5$ schneller als ein externer Zugriff auf die Magnetplatte, um die referenzierte Seite zu holen. Müßte im obigen Beispiel jeder Knoten im Suchpfad des AVL-Baumes getrennt in den Hauptspeicher übertragen werden, so kostet eine direkte Suche schon mehrere hundert Millisekunden – eine Größenordnung, die in zeitkritischen Aufgaben der Datei- oder Datenbankverwaltung nicht zu tolerieren ist.

Es werden deshalb für die Datenverwaltung auf mehrstufigen Speichern Baumstrukturen entwickelt, bei denen ein Knoten mehr als zwei Nachfolger hat. Ziel dabei ist es, möglichst viele Nachfolger pro Knoten (ein möglichst großes fan-out) zuzulassen. So entstehen „buschigere“ oder breitere Bäume mit einer geringeren Höhe. Außerdem wird, wie in Bild 1 skizziert, versucht, bei der Verarbeitung Lokalitätseigenschaften, die sich bei wiederholter Baumsuche ergeben, auszunutzen. So werden die Wurzelseite und die Seiten der höheren Baumebenen häufiger traversiert als Seiten auf tieferen Baumebenen. Durch spezielle Maßnahmen im E/A-Puffer (Seitenersetzungsalgorithmen) läßt es sich oft erreichen, daß beispielsweise die Wurzelseite und manchmal auch Seiten anderer Baumebenen schon im Hauptspeicher sind, wenn der entsprechende Baum traversiert werden soll. In praktischen Anwendungen werden also durch Suchbäume, die an die Speichercharakteristika angepaßt sind, Leistungsgewinne dadurch erzielt, daß die Ein-/Ausgabe beim Baumzugriff minimiert wird, und zwar durch große Zugriffsgranulate und damit „breite Bäume“ sowie durch Nutzung von Referenzlokalität.

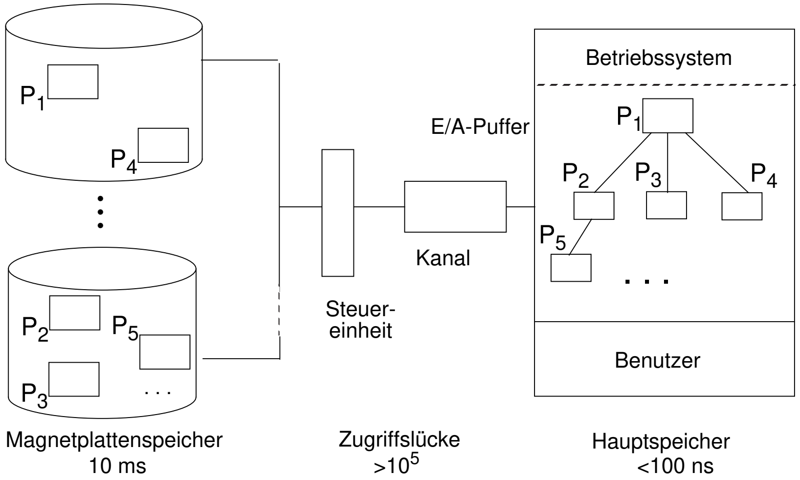


Bild 1: Zugriffswege bei einer zweistufigen Speicherhierarchie

Bei der Abbildung von Baumknoten (Einträge) liegt es nahe, einen großen binären Suchbaum in der in Bild 2 skizzierten Weise in Seiten zu unterteilen und jede Seite als Knoten aufzufassen, wobei bis zu m-1 ursprüngliche Knoten als Einträge im neuen Knoten (Seite) untergebracht werden. Eindeutiges Optimierungsziel ist die Minimierung der Höhe, da die Höhe die Anzahl der Externspeicherzugriffe im schlechtesten Fall angibt. Bei einem Kostenmaß für solche Bäume können die internen Operationen auf einem Knoten gegenüber den E/A-Operationen in erster Näherung vernachlässigt werden.

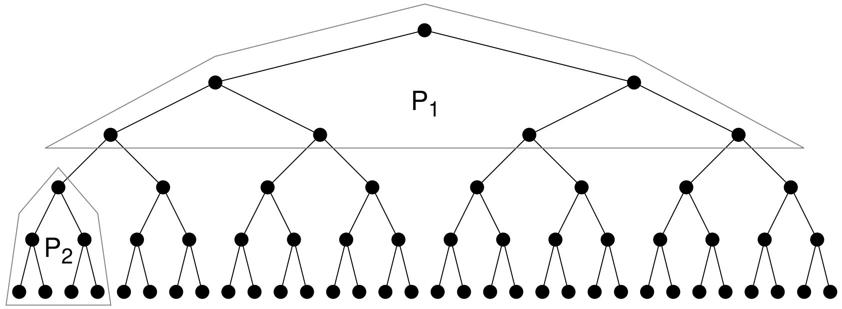


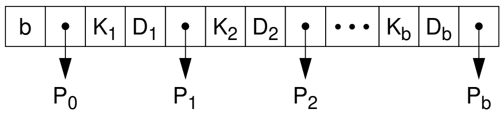
Bild 2: Unterteilung eines großen binären Suchbaumes in Seiten

1 m-Wege-Suchbäume

Die einfachste Abbildung, die sich nach dem in Bild 2 skizzierten Schema durch Zusammenfassung von bis zu m-1 Schlüsseln und m Zeigern ergibt, führt auf den (natürlichen) m-Wege-Suchbaum, der eine Analogie zum natürlichen binären Suchbaum darstellt. Ein m-Wege-Suchbaum verkörpert eine Sortierordnung auf der Menge der gespeicherten Schlüssel und erlaubt ähnliche Suchverfahren wie der binäre Suchbaum; er kann als seine Verallgemeinerung aufgefaßt werden. Von seiner Topologie her ist der m-Wege-Suchbaum jedoch ein allgemeiner Baum.

Definition: Ein <u>m-Wege-Suchbaum</u> oder ein <u>m-ärer Suchbaum</u> B ist ein Baum, in dem alle Knoten einen Grad ≤ m besitzen. Entweder ist B leer oder er hat folgende Eigenschaften:

i. Jeder Knoten des Baums hat folgende Struktur:



Die P_i, $0 \le i \le b$, sind Zeiger auf die Unterbäume des Knotens und die K_i und D_i, $1 \le i \le b$ sind Schlüsselwerte und Daten.

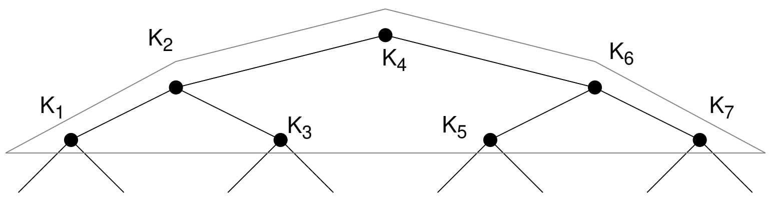
ii. Die Schlüsselwerte im Knoten sind aufsteigend geordnet: $K_i \le K_{i+1}$, $1 \le i < b$.

iii. Alle Schlüsselwerte im Unterbaum von P_i sind kleiner als der Schlüsselwert K_{i+1}, $0 \le i < b$.

iv. Alle Schlüsselwerte im Unterbaum von P_i sind größer als der Schlüsselwert K_i, $1 \le i \le b$.

v. Die Unterbäume von P_i, $0 \le i \le b$ sind auch m-Wege-Suchbäume.

Offensichtlich lassen sich durch das eingeführte Knotenformat die nachfolgend gezeigten drei Ebenen eines binären Suchbaums (mit b = 7) effizient als ein Knoten eines m-Wege-Suchbaums abbilden.



Die D_i können Daten oder Zeiger auf die Daten repräsentieren. Oft sind die Daten auf einem separaten Speicherplatz abgelegt. Dann stellt die Baumstruktur einen Index zu den Daten dar. Zur Vereinfachung werden wir die D_i weglassen.

In Bild 3 ist der Aufbau eines m-Wege-Suchbaumes für m=4 gezeigt. Seine Knoteninhalte lassen sich wie folgt interpretieren: $S(P_i)$ sei die Seite, auf die P_i zeigt, und $K(P_i)$ sei die Menge aller Schlüssel, die im Unterbaum mit Wurzel $S(P_i)$ gespeichert werden können.

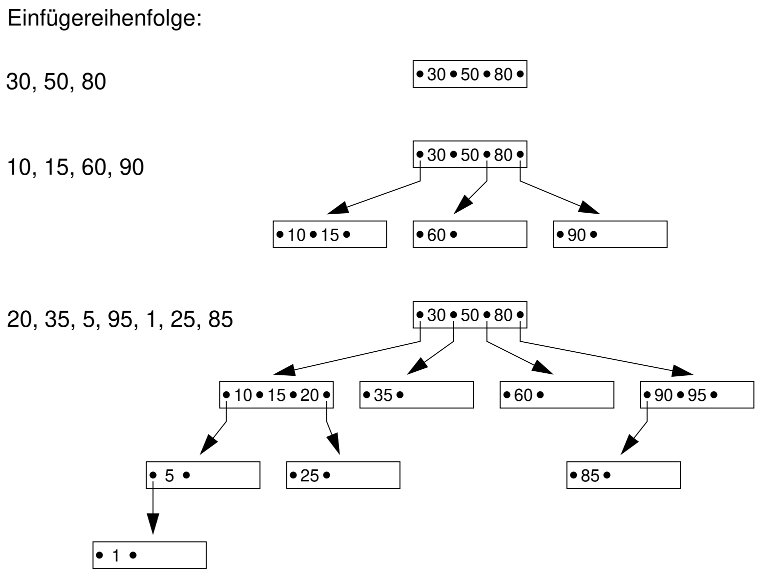


Bild 3: Aufbau eines m-Wege-Suchbaumes (m=4)

Dann gelten folgende Ungleichungen:

i. $x \in K(P_0):(x < K_1)$

ii. $x \in K(P_i):(K_i < x < K_{i+1})$ für $i = 1, 2, \ldots, b-1$

iii. $x \in K(P_b):(K_b < x)$

Aus der Betrachtung der Baumstruktur wird deutlich, daß die Schlüssel in den inneren Knoten zwei Funktionen haben. Sie identifizieren Daten(sätze), und sie dienen als Wegweiser in der Baumstruktur.

Für den m-Wege-Suchbaum wird nur die direkte und sequentielle Suche skizziert. Einfüge- und Löschoperationen erklären sich analog zum binären Suchbaum intuitiv. Da für den m-Wege-Suchbaum keine Balancierungsmechanismen definiert sind, kann oftmals „Wildwuchs" entstehen, der die praktische Anwendung solcher Bäume in Frage stellt.

Definition des Knotenformats:

```
CONST Emax = M - 1;              {maximale Anzahl von Einträgen/Knoten}
TYPE  Sptr = POINTER TO Seite;
      Index = [1..Emax];
      Eintrag =  RECORD
                     Key : Schluesseltyp;
                     Info : Infotyp;
                     Ptr : Sptr
                 END;
      Seite =    RECORD
                     B : Index;     {aktuelle Anzahl von Einträgen}
                     Po : Sptr;
                     Evektor : ARRAY Index OF Eintrag
                 END;
PROCEDURE Msuche (X : Schluesseltyp; P : Sptr; VAR Element : Eintrag);
VAR   I : Index;
BEGIN
   IF    P = NIL THEN
         WriteString('Schluessel X ist nicht im Baum')
   ELSIF X < P^.Evektor[1].Key THEN            {X < K1}
         Msuche(X, P^.Po, Element)
   ELSE
         I := 1;
         WHILE (I < P^.B) AND (X > P^.Evektor[I].Key) DO
               I := I + 1;
         END;
         IF   P^.Evektor[I].Key = X THEN      {Ki = X, 1 ≤ i ≤ b}
              Element := P^.Evektor[I]
         ELSE                                 {Ki < X < Ki+1, 1 ≤ i ≤ b oder X > Kb}
              Msuche(X, P^.Evektor[I].Ptr, Element)
         END;
   END;
END Msuche;
```

Programm 1: Rekursive Prozedur zum Aufsuchen eines Schlüssels in einem m-Wege-Suchbaum

Der Suchvorgang nach einem Schlüssel X in einem m-Wege-Suchbaum ist eine Erweiterung der direkten Suche in einem binären Suchbaum. Er läßt sich in analoger Weise als rekursive Prozedur Msuche formulieren. Dazu wird ein geeignetes Knotenformat durch die im Programm 1 gegebene MODULA-Definition eingeführt. Der Suchschlüssel ist in der Variablen X enthalten. Der Zeiger P zeigt anfänglich auf die Wurzel des Baumes. Falls die Suche erfolgreich ist, enthält Element den gesuchten Baumeintrag.

Im Suchalgorithmus wurde in jedem Knoten sequentiell gesucht. Wenn die Menge der Schlüssel pro Knoten sehr groß ist, könnte zur Beschleunigung eine binäre Suche eingesetzt werden.

Die sequentielle Suche kann durch Verallgemeinerung des Durchlaufs in Zwischenordnung bewerkstelligt werden. Der Durchlauf in symmetrischer Ordnung erzeugt die sortierte Folge aller Schlüssel (Programm 2).

```
PROCEDURE Symord(P : Sptr);
VAR   I : Index;
BEGIN
   IF  P <> NIL THEN
       Symord(P^.Po);
       FOR I := 1 to P^.B DO
           WriteString(P^.Evektor[I].Key);
           Symord(P^.Evektor[I].Ptr)
       END
   END;
END Symord;
```

Programm 2: Prozedur zum Durchlauf eines m-Wege-Suchbaums in symmetrischer Ordnung

Wie aus Bild 3 deutlich wird, ist der m-Wege-Suchbaum im allgemeinen nicht ausgeglichen. Es ist für Aktualisierungsoperationen kein Balancierungsmechanismus vorgesehen. Das führt dazu, daß Blätter auf verschiedenen Baumebenen auftreten können und daß der vorhandene Speicherplatz sehr schlecht ausgenutzt wird. Im Extremfall entartet der Baum zu einer geketteten Liste. Seine Höhe kann deshalb beträchtlich schwanken.

Die Anzahl der Knoten in einem vollständigen Baum der Höhe $h, h \geq 1$ ist

$$N = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}.$$

Da jeder Knoten bis zu m-1 Schlüssel besitzen kann, ergibt sich als maximale Anzahl von Schlüsseln

$$n_{max} = N \cdot (m-1) = m^h - 1 \; .$$

Im ungünstigsten Fall ist der Baum völlig entartet:

$$n = N = h$$

Daraus ergeben sich folgende Schranken für die Höhe eines m-Wege-Suchbaums:

$$\log_m(n+1) \le h \le n$$

Um das Entstehen ungünstiger Bäume zu verhindern, fand ebenso wie beim binären Suchbaum eine Weiterentwicklung statt, die auf die Einführung eines geeigneten Balancierungsverfahrens abzielte. Die Einhaltung des striktest möglichen Balancierungskriteriums – Ausgeglichenheit des Baumes bei minimal möglicher Höhe – hätte jedoch wiederum auf einen Wartungsaufwand von O(n) geführt und wäre damit viel zu teuer gewesen. Deshalb wurde ein Balancierungsmechanismus entwickelt, der mit Hilfe von lokalen Baumtransformationen den Mehrwegbaum fast ausgeglichen hält.

2 B-Bäume

B-Bäume sind fast ausgeglichene Mehrwegbäume. In Bezug auf ihre Knotenstruktur (Seiten) sind sie sogar vollständig ausgeglichen. Durch ihre lokalen Reorganisationsmaßnahmen garantieren sie jedoch nicht immer eine minimale Höhe, d. h., die Belegung der einzelnen Knoten kann in gewissen Grenzen schwanken.

Die B-Baumstruktur wurde 1970 von R. Bayer und E. McCreight entwickelt [BMc72]. Sie und einige ihrer Varianten werden bei einem breiten Spektrum von Anwendungen eingesetzt. Der Titel eines Aufsatzes „The Ubiquitous B-Tree“ von D. Comer charakterisiert diesen Sachverhalt sehr treffend [Co79]. In Datei- und Datenbanksystemen dienen sie zur Organisation von Zugriffspfadstrukturen. Ihr Zugriffsverhalten ist weitgehend unabhängig von der Menge der verwalteten Elemente, so daß sie als Indexstruktur für 10 als auch für 10^7 Elemente oder mehr herangezogen werden. Da sie sehr breite Bäume von geringer Höhe garantiert, bietet sie eine effiziente Durchführung ihrer Grundoperationen – direkte und sequentielle Suche, Einfügen und Löschen von Schlüsseln.

Definition: Seien k, h ganze Zahlen, $h \ge 0$, $k > 0$. Ein B-Baum B der Klasse $\tau(k, h)$ ist entweder ein leerer Baum oder ein geordneter Suchbaum mit folgenden Eigenschaften:

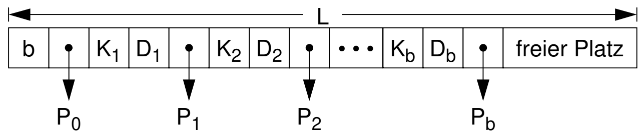
i. Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge.

ii. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.

iii. Jeder Knoten hat höchstens 2k+1 Söhne.

iv. Jedes Blatt mit der Ausnahme der Wurzel als Blatt hat mindestens k und höchstens 2k Einträge.

Für einen B-Baum ergibt sich folgendes Knotenformat:



Die Einträge für b, Schlüssel, Daten und Zeiger haben die festen Längen l_b, l_K, l_D und l_p. Als Knotengröße wird die Größe der Transporteinheit zwischen Haupt- und Externspeicher gewählt. Diese Knoten- oder Seitengröße sei L. Als maximale Anzahl von Einträgen pro Knoten erhalten wir dann

$$b_{max} = \left\lfloor \frac{L - l_b - l_p}{l_K + l_D + l_p} \right\rfloor = 2k .$$

Damit lassen sich die Bedingungen der Definition nun so formulieren:

iv. und iii. Eine Seite darf höchstens voll sein.

iv. und ii. Jede Seite (außer der Wurzel) muß mindestens halb voll sein. Die Wurzel enthält mindestens einen Schlüssel.

i. Der Baum ist, was die Knotenstruktur angeht, vollständig ausgeglichen.

Für den B-Baum gibt es eine geringfügig unterschiedliche Definition, bei der minimal $\lceil m/2 \rceil$ und maximal m Nachfolger pro Knoten zugelassen sind. Der einfacheren Schreibweise wegen bleiben wir bei unserer Definition, bei der 2k+1 dem m entspricht. In Bild 4 ist ein B-Baum der Klasse $\tau(2,3)$ veranschaulicht. In jedem Knoten stehen die Schlüssel in aufsteigender Ordnung mit $K_1 < K_2 < \ldots < K_b$. Bei b Schlüsseln hat er b+1 Söhne. Seine Knoteninhalte lassen sich in gleicher Weise wie beim m-Wege-Suchbaum interpretieren. Auch die Such- und Durchlauf-Algorithmen bleiben gleich. Jeder Schlüssel hat wieder die Doppelrolle als Identifikator eines Datensatzes und als Wegweiser in der Baumstruktur. Sie erschwert gewisse Optimierungen, auf die später eingegangen wird.

Bemerkung: Die Klassen $\tau(k,h)$ sind nicht alle disjunkt. Es ist leicht einzusehen, daß beispielsweise ein maximaler Baum aus $\tau(2,3)$ ebenso in $\tau(3,3)$ und $\tau(4,3)$ ist.

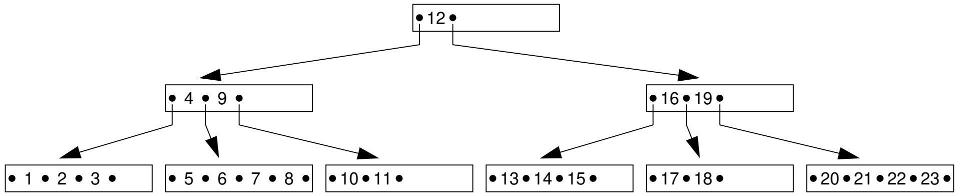


Bild 4: B-Baum der Klasse $\tau(2,3)$

2.1 Höhe des B-Baumes

Die Effizienz der direkten Suche wird wesentlich von der Höhe des B-Baums bestimmt.

Satz: Für die Höhe h eines Baums der Klasse $\tau(k,h)$ mit n Schlüsseln gilt:

$$\log_{2k+1}(n+1) \le h \le \log_{k+1}((n+1)/2) + 1 \qquad \text{für } n \ge 1$$

$$\text{und } h = 0 \qquad \text{für } n = 0$$

Für den Beweis der oberen und unteren Grenzen leiten wir die minimale und maximale Anzahl von Knoten in $\tau(k,h)$ ab. Einen minimalen Baum erhält man, wenn jeder Knoten die kleinstmögliche Anzahl von Söhnen hat. Das ergibt nach Definition

$$\begin{aligned} N_{min}(k,h) &= 1 + 2 + 2\cdot(k+1) + 2\cdot(k+1)^2 + \ldots + 2(k+1)^{h-2} \\ &= 1 + 2\cdot\sum_{i=0}^{h-2}(k+1)^i \\ &= 1 + \frac{2}{k}\cdot((k+1)^{h-1} - 1) \end{aligned} \tag{1}$$

Offensichtlich erhält man einen maximalen Baum, wenn jeder Knoten die größtmögliche Anzahl von Söhnen hat.

$$\begin{aligned} N_{max}(k,h) &= 1 + 2k + 1 + (2k+1)^2 + \ldots + (2k+1)^{h-1} \\ &= \sum_{i=0}^{h-1}(2k+1)^i \\ &= \frac{(2k+1)^h - 1}{2k} \end{aligned} \tag{2}$$

Damit ergibt sich als obere und untere Grenze für die Knotenzahl N(B) eines beliebigen B-Baumes $B \in \tau(k,h)$

$$1 + \frac{2}{k}\cdot((k+1)^{h-1} - 1) \le N(B) \le \frac{(2k+1)^h - 1}{2k} \qquad \text{für } h \ge 1 \tag{3}$$

$$\text{und} \qquad N(B) = 0 \qquad \text{für } h = 0$$

Aus $N_{min}(k,h)$ und $N_{max}(k,h)$ erhält man die minimale und maximale Anzahl von Schlüsseln, die im Baum gespeichert sind. Für $h \ge 1$ gilt:

$$n \ge n_{min} = 2(k+1)^{h-1} - 1 \tag{4}$$

$$\text{und} \qquad n \le n_{max} = (2k+1)^h - 1 \tag{5}$$

Nach h aufgelöst erhalten wir die oben angegebenen Grenzen.

In einem B-Baum der Klasse $\tau(k,h)$ kann also die Anzahl der Schlüssel in den abgeleiteten Grenzen schwanken:

$$2(k+1)^{h-1} - 1 \le n \le (2k+1)^h - 1 \tag{6}$$

Für $\tau(2,3)$ (Bild 4) bedeutet dies $17 \le n \le 124$. Eingangs bezeichneten wir den B-Baum als fast ausgeglichenen Mehrwegbaum. In Bezug auf seine Knotenstruktur ist er zwar ausgeglichen, erreicht aber nicht immer die minimale mögliche Höhe. Der quantitative Unterschied zum (optimalen) ausgeglichenen m-Wege-Suchbaum soll kurz skizziert werden. In Analogie zum ausgeglichenen binären Suchbaum sei er wie folgt definiert:

- h-1 Stufen von der Wurzel her sind voll belegt: b = 2k
- die Belegung der Stufe h (h > 1) erfüllt das B-Baum-Kriterium: $k \le b \le 2k$
- für h=1 gilt Ungleichung (6).

Mit $N_{max}(k,h) = \frac{(2k+1)^{h-1} - 1}{2k} + (2k+1)^{h-1}$

gilt für die optimale Belegung immer

$$\begin{aligned} n_{opt}(k,h) &\ge (2k+1)^{h-1} - 1 + k(2k+1)^{h-1} \\ &= (k+1)\cdot(2k+1)^{h-1} - 1 \qquad \text{für } n > 1. \end{aligned}$$

Mit n_{max} als oberer Grenze ergibt sich

$$(k+1)\cdot(2k+1)^{h-1} - 1 \le n_{opt} \le (2k+1)^h - 1 \quad .$$

Als Schwankungsbereich für $\tau(2,3)$ erhalten wir damit $74 \le n_{opt} \le 124$. Da sich die Bereiche der n_{opt} für verschiedene n nicht überdecken, können solche ausgeglichene m-Wege-Suchbäume nur für bestimmte Schlüsselmengen aufgebaut werden.

2.2 Einfügen in B-Bäumen

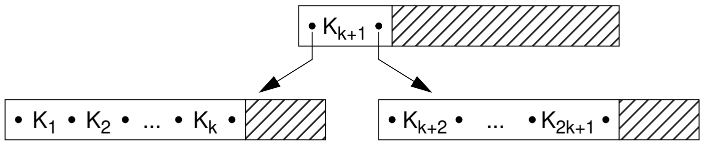
Während die bisher betrachteten Bäume alle von der Wurzel zu den Blättern hin gewachsen sind, ist bei B-Bäumen das Wachstum von den Blättern zur Wurzel hin gerichtet. Man beginnt mit einer leeren Seite als Wurzel. Die ersten 2k Schlüssel werden in die Wurzel unter Beachtung der Ordnungsrelation eingefügt. Durch den nächsten Schlüssel läuft die Wurzel über.

$$\boxed{\bullet\ K_1 \bullet K_2 \bullet \ldots \bullet K_{2k} \bullet} \quad K_{2k+1}$$

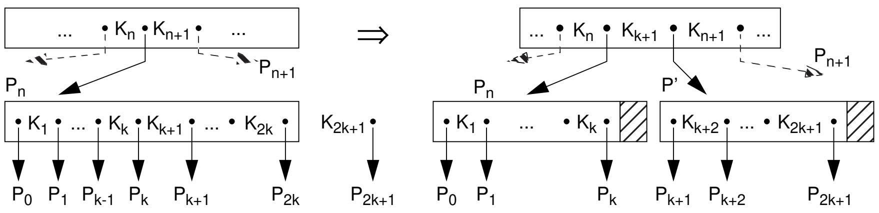
Das Schaffen von neuem Speicherplatz wird durch eine für den B-Baum fundamentale Operation – dem Split-Vorgang – erreicht. Über die Freispeicherverwaltung wird eine neue Seite angefordert. Die Schlüsselmenge wird aufgespalten und nach dem folgenden Prinzip neu aufgeteilt.

$$\boxed{K_1\ K_2\ \ldots\ K_k} \quad \overset{\Uparrow}{K_{k+1}} \quad \boxed{K_{k+2}\ \ldots\ K_{2k+1}}$$

Die ersten k-Schlüssel verbleiben in der ursprünglichen Seite. Der in der Sortierreihenfolge mittlere Schlüssel – der Median – wird zum Vaterknoten gereicht, während die restlichen k Schlüssel in die neue Seite kommen. Da in der obigen Situation kein Vaterknoten vorhanden ist, wird eine weitere neue Seite zugeordnet, so daß der B-Baum folgendes Aussehen erhält:



Der Schlüssel K_{k+1} in der neuen Wurzel dient nun als Wegweiser, ob ein beliebiger Schlüssel K im linken oder im rechten Unterbaum steht oder dort einzufügen ist. Weitere Schlüssel werden in den Blättern in Sortierreihenfolge eingefügt. Sobald ein Blatt überläuft, wird ein Split-Vorgang ausgeführt, wodurch ein weiterer Schlüssel in die Wurzel aufgenommen wird. Nach einer Reihe von Split-Vorgängen wird schließlich die Wurzel selbst überlaufen. Dieses Ereignis erzwingt die Aufteilung des Schlüssels in der Wurzel auf zwei Knoten und einer neuen Wurzel (wie oben gezeigt). Dadurch vergrößert sich die Höhe des Baumes um 1. Dieser Split-Vorgang als allgemeines Wartungsprinzip des B-Baumes läßt sich also solange rekursiv fortsetzen, bis genügend Speicherplatz auf allen Baumebenen geschaffen worden ist. Folgendes allgemeine Schema wird dabei angewendet:



Im Einfügealgorithmus werden also folgende Schritte ggf. rekursiv ausgeführt:

- Suche Einfügeposition
- Wenn Platz vorhanden ist, speichere Element, sonst schaffe Platz durch Split-Vorgang und füge ein.

Für eine Folge von Schlüsseln ist der Einfügealgorithmus bei einem B-Baum aus $\tau(2,h)$ in Bild 5 gezeigt. Bei Einfügen der Schlüssel 33 und 50 verändert der B-Baum seine Höhe h jeweils um 1; dabei fallen jeweils h Split-Vorgänge an (Höhe vor Einfügen).

2.3 Kostenanalyse für Einfügen und Suchen

Wir hatten bereits festgestellt, daß bei B-Baum-Operationen die Anzahl der Externspeicher-Zugriffe als relevante Größe zu berücksichtigen ist. Wir nehmen dabei an, daß jede Seite, die für eine Operation benötigt wird, genau einmal vom Externspeicher geholt werden muß. Bei Veränderung der Seite muß sie dann auch genau einmal zurückgeschrieben werden. Wir bezeichnen die Anzahl der zu holenden Seiten mit f (fetch) und der zu schreibenden Seiten mit w (write).

Für die sequentielle Suche – beispielsweise durch einen Durchlauf in symmetrischer Ordnung – fallen N Lesezugriffe an. Da dabei jeder interne Knoten sehr oft aufgesucht wird, muß gewährleistet sein, daß er während dieses Referenzzeitraums im Hauptspeicher gehalten werden kann. Später werden wir eine Variante des B-Baum kennenlernen, die eine effizientere Durchführung dieser Operation gestattet.

Bei der direkten Suche erhalten wir:

- $f_{min} = 1$: der Schlüssel befindet sich in der Wurzel
- $f_{max} = h$: der Schlüssel ist in einem Blatt

Um eine Aussage über die mittleren Suchkosten f_{avg} zu bekommen, berechnen wir sie für den Fall der maximalen Belegung eines B-Baumes. Wir werden sie dann weiterhin

Einfügereihenfolge:

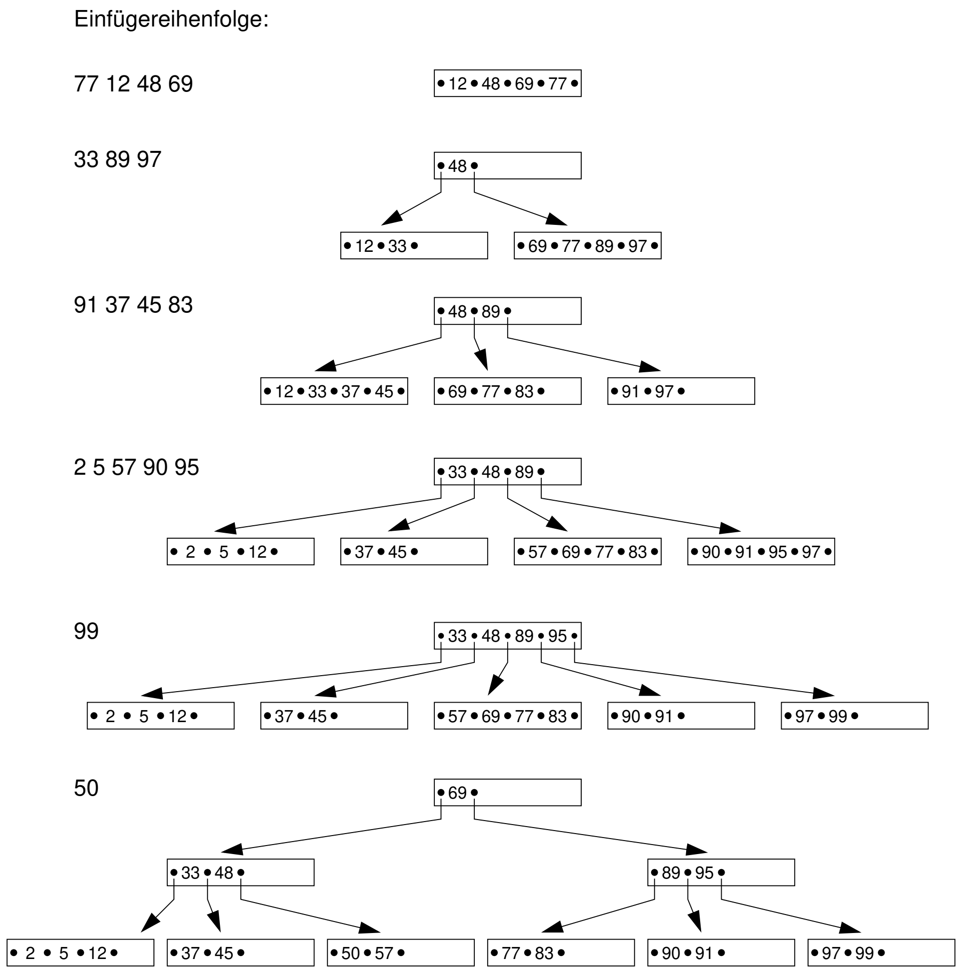


Bild 5: Aufbau eines B-Baumes der Klasse $\tau(2, h)$

für seine minimale Belegung ableiten und haben so die Grenzen des Intervalls, in dem die mittleren Suchkosten zu erwarten sind.

Mit Hilfe von Gleichung (2) erhalten wir als gesamte Zugriffskosten bei maximaler Belegung eines B-Baumes, wenn auf jeden Schlüssel genau einmal zugegriffen wird,

$$
\begin{aligned}
z_{max} &= 2k \cdot \sum_{i=0}^{h-1} (i+1) \cdot (2k+1)^i \\
&= (m-1) \cdot \left(\sum_{i=0}^{h-1} m^i + \sum_{i=0}^{h-1} i \cdot m^i \right) \\
&= (m-1) \cdot \left(\frac{m^h - 1}{m-1} + \frac{(h-1) \cdot m^h}{(m-1)} - \frac{m^h - m}{(m-1)^2} \right) \\
&= h \cdot m^h - 1 - \frac{m^h - m}{m-1} \quad \text{mit } m = 2k+1
\end{aligned}
$$

Die mittleren Zugriffskosten ergeben sich unter Berücksichtigung von (5) durch

$$
\begin{aligned}
f_{avg}(max) &= \frac{z_{max}}{n_{max}} \\
&= \frac{h(m^h - 1) + h - 1}{m^h - 1} - \frac{m^h - 1 - m + 1}{(m-1) \cdot (m^h - 1)} \\
&= h + \frac{h}{m^h - 1} - \frac{1}{m-1} \\
&= h - \frac{1}{2k} + \frac{h}{(2k+1)^h - 1} \qquad (7)
\end{aligned}
$$

Bei minimaler Belegung eines B-Baumes erhalten wir die gesamten Zugriffskosten bei einmaligem Aufsuchen jedes Schlüssels mit Hilfe von Gleichung (1)

$$
\begin{aligned}
z_{min} &= 1 + 2k \cdot \sum_{i=0}^{h-2} (i+2) \cdot (k+1)^i \\
&= 1 + 2(m-1) \cdot \left(\sum_{i=0}^{h-2} 2m^i + \sum_{i=0}^{h-2} i \cdot m^i \right) \\
&= 1 + 2h \cdot m^{h-1} - 4 - 2\frac{m^{h-1} - m}{m-1} \qquad \text{mit } m = k+1 .
\end{aligned}
$$

Unter Berücksichtigung von (4) lassen sich die mittleren Zugriffskosten bestimmen:

$$
\begin{aligned}
f_{avg}(min) &= \frac{z_{min}}{n_{min}} \\
&= \frac{h(2m^{h-1} - 1) + h - 3 - \dfrac{2m^{h-1} - 1 - 2m + 1}{m-1}}{2m^{h-1} - 1}
\end{aligned}
$$

$$= h + \frac{h-3}{2m^{h-1}-1} - \frac{1}{m-1} + \frac{2}{2m^{h-1}-1} + \frac{1}{(2m^{h-1}-1)\cdot(m-1)}$$

$$= h - \frac{1}{k} + \frac{h-1}{2(k+1)^{h-1}-1} + \frac{1}{k(2(k+1)^{h-1}-1)} \qquad (8)$$

In den Gleichungen (7) und (8) lassen sich für h > 1 die 3. und 4. Terme vernachlässigen, da der Parameter k in B-Bäumen typischerweise sehr groß ist $(k \approx 100 - 200)$. Wir erhalten also als Grenzen für den mittleren Zugriffsaufwand

$$f_{avg} = h \qquad \text{für } h = 1$$

$$\text{und } h - \frac{1}{k} \le f_{avg} \le h - \frac{1}{2k} \qquad \text{für } h > 1.$$

Diese Intervallgrenzen sind sehr eng und kaum von h unterschieden.

Bei einer typischen Baumhöhe von h=3 und einem k=100 ergibt sich $2.99 \le f_{avg} \le 2.995$, d. h., beim B-Baum sind die maximalen Zugriffskosten h eine gute Abschätzung der mittleren Zugriffskosten.

Bei der Analyse der Einfügekosten müssen wir ebenso eine Fallunterscheidung machen. Im günstigsten Fall ist kein Split-Vorgang erforderlich. Da immer in einem Blatt eingefügt wird, ergibt sich:

$$f_{min} = h;\ w_{min} = 1;$$

Der ungünstigste Fall tritt dann ein, wenn der Baum seine Höhe ändert. Dabei werden alle Seiten längs des Suchpfades gespalten und außerdem wird eine neue Wurzel zugeordnet. Wenn h die Höhe des B-Baumes vor der Einfügung war, dann gilt:

$$f_{max} = h;\ w_{max} = 2h+1;$$

Dieser Fall verdeutlicht sehr schön, wie der B-Baum von den Blättern zur Wurzel hin wächst. Wegen seiner hohen Kosten tritt er glücklicherweise nur selten auf. Wenn ein B-Baum der Höhe h durch sukzessive Einfügungen aufgebaut wird, ereignet sich dieser Fall nur h-1 mal.

Um die durchschnittlichen Einfügekosten bestimmen zu können, benötigen wir eine Abschätzung der Wahrscheinlichkeit für das Auftreten eines Split-Vorgangs. Unter der Annahme, daß im betrachteten B-Baum nur gesucht und eingefügt wird, erhalten wir eine obere Schranke für seine durchschnittlichen Einfügekosten. Hat der Baum mit n Elementen N(n) Knoten, so gab es bei seinem Aufbau höchstens N(n)-1 Split-Vorgänge. N(n) kann durch Vorgabe einer minimalen Belegung abgeschätzt werden:

$$N(n) \le \frac{n-1}{k} + 1 .$$

Die Wahrscheinlichkeit dafür, daß eine Einfügung eine Spaltung auslöst, ergibt sich aus der Menge der Spaltungen und den gegebenen n Einfügungen höchstens zu

$$\frac{\frac{n-1}{k} + 1 - 1}{n} = \frac{n-1}{n \cdot k} < \frac{1}{k} .$$

Bei jeder Einfügung muß eine Seite geschrieben werden. Bei einem Split-Vorgang müssen zusätzlich zwei Schreibvorgänge – für die neu zugeordnete Seite und die Vaterseite – abgewickelt werden.

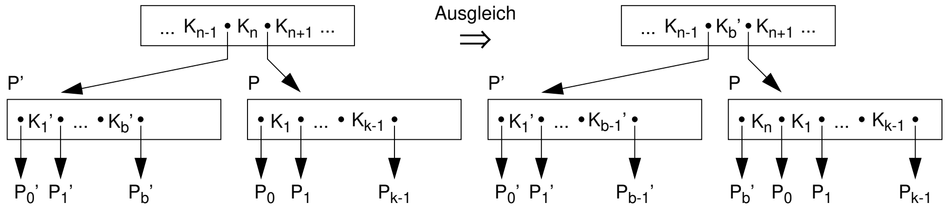
Der durchschnittliche Mehraufwand bei einer Einfügung resultiert aus der Wahrscheinlichkeit einer Spaltung und ihren zusätzlichen Kosten.

$$f_{avg} = h;\ w_{avg} < 1 + \frac{2}{k};$$

Wenn wir ein k=100 unterstellen, kostet eine Einfügung im Mittel w_{avg} < 1 + 2/100 Schreibvorgänge, d. h., es entsteht eine Belastung von 2% für den Split-Vorgang. Bei den in praktischen Fällen üblichen Werten von k werden also Einfügungen durch den Mehraufwand für das Spalten nur unwesentlich belastet.

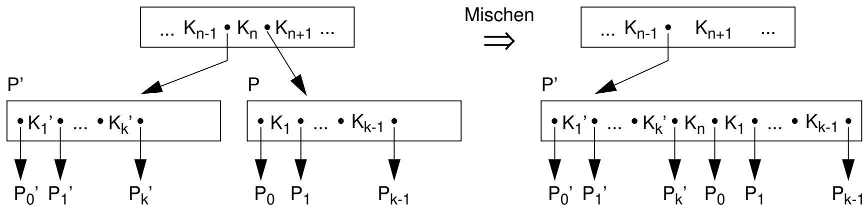
2.4 Löschen in B-Bäumen

Das Löschen eines Schlüssels aus einem B-Baum ist nur geringfügig komplizierter als das Einfügen. Die Hauptschwierigkeit besteht darin, die B-Baum-Eigenschaft wiederherzustellen, wenn die Anzahl der Elemente in einem Knoten kleiner als k wird. Durch Ausgleich mit Elementen aus einer Nachbarseite oder durch Mischen (Konkatenation) mit einer Nachbarseite wird dieses Problem gelöst. Beim Ausgleichsvorgang sind in der Seite P k-1 Elemente und in P' mehr als k Elemente. Er läuft nach folgendem Schema ab:



Ein Ausgleichsvorgang setzt sich im Baum nicht fort. Es wird immer mindestens ein Element zum Nachbarknoten hin verschoben (über den Vaterknoten rotiert). Es ist aber auch eine Implementierung denkbar, bei der ein Ausgleich derart geschieht, daß in jedem Knoten etwa die Hälfte der Elemente ((b+k-1)/2) abgelegt werden. Nachfolgende Löschungen im gleichen Knoten würden nicht sofort wieder einen Ausgleichsvorgang nach sich ziehen.

Ein Mischen geschieht dann, wenn in der Seite P, in der gelöscht wurde, k-1 Elemente und in der Nachbarseite P' genau k Elemente sind. Das allgemeine Löschschema sieht folgendermaßen aus:



Dieser Mischvorgang kann sich im Baum fortsetzen. Er kann im Vaterknoten einen Ausgleich oder wieder ein Mischen mit einem Nachbarknoten anstoßen.

Mit diesen Operationen läßt sich jetzt der Löschalgorithmus folgendermaßen erklären. Dabei bedeutet b Anzahl der Elemente:

1. Löschen in Blattseite
   - Suche x in Seite P
   - Entferne x in P und wenn
     a) $b \geq k$ in P: tue nichts
     b) $b = k-1$ in P und $b > k$ in P': gleiche Unterlauf über P' aus
     c) $b = k-1$ in P und $b = k$ in P': mische P und P'.
2. Löschen in innerer Seite
   - Suche x
   - Ersetze $x = K_i$ durch kleinsten Schlüssel y in $B(P_i)$ oder größten Schlüssel y in $B(P_{i-1})$ (nächstgrößerer oder nächstkleinerer Schlüssel im Baum)
   - Entferne y im Blatt P
   - Behandle P wie unter 1.

In Bild 6 ist der Löschalgorithmus an einem Beispiel für das Löschen in einer Blattseite und in einer inneren Seite gezeigt.

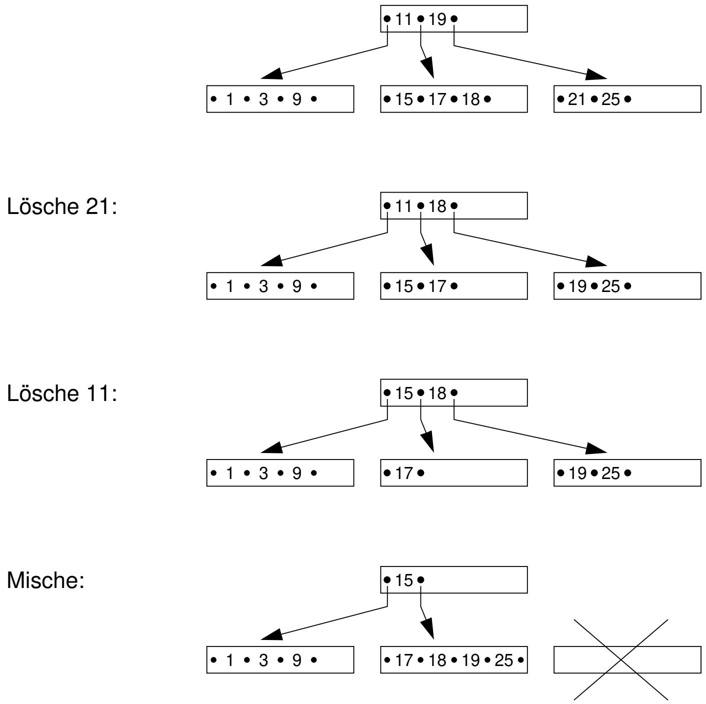


Bild 6: Löschen in einem B-Baum ($\tau(2,2)$)

Kostenanalyse für das Löschen

Der günstigste Fall liegt dann vor, wenn in einem Blatt gelöscht wird, aber kein Mischen oder Ausgleich erfolgt.

$$f_{min} = h \, ; \, w_{min} = 1 \, ;$$

Beim Ausgleich eines Unterlaufes, der sich nicht fortpflanzt, sind 3 Seiten betroffen, die geschrieben werden müssen. Beim Mischvorgang müssen nur die „überlebende" Seite und ihre Vaterseite geschrieben werden. Der schlimmste Fall tritt nun in der pathologischen Situation auf, in der vom betroffenen Blatt her alle Seiten bis auf die ersten zwei in einem Löschpfad gemischt werden; beim Nachfolger der Wurzel tritt ein Unterlauf ein, der über ihre Nachbarseite und über die Wurzel ausgeglichen werden muß:

$$f_{max} = 2h - 1 \, ; \, w_{max} = h + 1 \, ;$$

Für die durchschnittlichen Löschkosten geben wir eine obere Schranke an. Wir berechnen sie unter der Annahme, daß der Reihe nach alle n Schlüssel aus einem Baum herausgelöscht werden. Findet bei einem Löschvorgang weder Mischen noch Ausgleich statt, gilt:

$$f_1 = h;\ w_1 \le 2;$$

Jedes Löschen eines Schlüssels verursacht höchstens einen Unterlauf. Als zusätzliche Kosten kommen bei einem Ausgleichsvorgang hinzu:

$$f_2 = 1;\ w_2 = 2;$$

Die Gesamtzahl der notwendigen Mischvorgänge ist durch N(n)-1, also durch (n-1)/k beschränkt. Jedes Mischen kostet zusätzlich einen Lese- und einen Schreibzugriff. Da nur ein Bruchteil der Löschvorgänge ein Mischen erforderlich macht, belastet es die einzelne Operation mit

$$f_3 = w_3 = \frac{1}{n} \cdot \frac{n-1}{k} < \frac{1}{k}.$$

Durch Summation der Kostenanteile erhalten wir als durchschnittliche Löschkosten

$$f_{avg} \le f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$

$$w_{avg} \le w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$

Der Anteil der Kosten für einen Mischvorgang ist für große k vernachlässigbar.

3 Optimierungsmaßnahmen in B-Bäumen

Nachfolgend diskutieren wir einige Möglichkeiten zur Verbesserung wichtiger Eigenschaften von B-Bäumen. Sie betreffen die Speicherplatzbelegung, die Suche in einer Seite sowie den Einsatz von variabel langen Schlüsseln und Datensätzen.

3.1 Verallgemeinerte Überlaufbehandlung

Im Originalvorschlag des B-Baum-Konzeptes verlangte das Einfügen eines Schlüssels einen Split-Vorgang, sobald die betreffende Seite überlief. Dabei wurde die Schlüsselmenge so auf die ursprüngliche und auf die neu zugeordnete Seite aufgeteilt, daß anschließend in beiden Seiten jeweils k Schlüssel gespeichert waren. Da eine volle Seite den Split-Vorgang auslöste, bezeichnen wir in diesem Fall den Splitfaktor mit m=1 (einfacher Überlauf). Die Speicherplatzbelegung in beiden betroffenen Seiten ist nach einer Spaltung β = m/(m+1) = 1/2. Es läßt sich leicht überlegen, daß Einfügefolgen auftreten können, die in der ganzen Baumstruktur eine Speicherplatzbelegung von 50% (Ausnahme: Wurzel) erzeugen. Ein Beispiel für eine solche Folge ist das sortierte Einfügen der Schlüssel. Die Speicherplatzbelegung unter einem einfachen Überlaufschema bei zufälliger Schlüsselreihenfolge ist $\beta \approx \ln 2$ ($\approx 69\%$).

Zur Verbesserung der Speicherplatzausnutzung kann analog zum Ausgleich des Unterlaufs beim Löschen ein Ausgleich des Überlaufs auf Kosten des freien Platzes in einer Nachbarseite herangezogen werden. Dabei werden ein oder mehrere Schlüssel über den Vater zur Nachbarseite geschafft (rotiert), ohne daß eine neue Seite zugeordnet wird. Die aktuelle Implementierung kann auf unterschiedliche Weise geschehen. Ein geeigneter Vorschlag ist die Neuaufteilung der (2k+1+b) Schlüssel derart, daß etwa (2k+1+b)/2 Schlüssel in jede der beiden Seiten kommen. Der Split-Vorgang wird solange verzögert, bis beide benachbarten Seiten voll sind. Dann erfolgt eine Spaltung bei „doppeltem" Überlauf (m=2), bei der die Schlüsselmenge der beiden Seiten gleichmäßig auf drei Seiten aufgeteilt wird. Ihre Speicherplatzbelegung ist dann β = 2/3. Falls nur Einfügungen abgewickelt werden, liegt die Speicherplatzausnutzung im ungünstigsten Fall bei etwa 66%. Treten jedoch zusätzlich Löschvorgänge auf, kann sie wieder bis auf 50% absinken. Durch eine allerdings recht komplexe Erweiterung des Ausgleichs beim Löschen (über 3 Seiten) läßt sich eine höhere Speicherplatzbelegung auch beim Löschen garantieren. Wir wollen jedoch diese Möglichkeit nicht vertiefen.

Der B-Baum mit doppeltem Überlauf (Splitfaktor m = 2) wird in der Literatur manchmal als B*-Baum bezeichnet [Kn73]. Wir verwenden diese Bezeichnung hier nicht, da wir sie für eine andere Variante des B-Baumes reserviert haben.

In Bild 7 wird das eben diskutierte Ausgleichsprinzip am Beispiel erläutert. Es veranschaulicht, wie die Spaltung einer Seite hinausgeschoben und wie dadurch eine bessere Speicherplatzausnutzung erzielt wird.

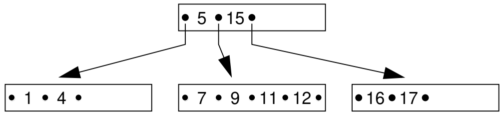
Als Grenzen für die Einfügekosten bei einem Splitfaktor von m = 2 erhalten wir:

$$f_{min} = h; \qquad w_{min} = 1$$

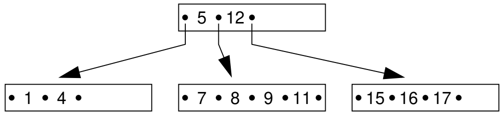
$$f_{max} = 2h - 1;\ w_{max} = 3h.$$

Der schlimmste Fall tritt dann ein, wenn auf jeder Ebene – vom Blatt bis zum Nachfolger der Wurzel – die Schlüssel aus zwei Seiten auf drei Seiten aufgeteilt werden und wenn zusätzlich die Wurzel gespalten wird. Unter Betrachtung eines reinen Einfügeprozesses lassen sich in einfacher Weise Grenzen für die durchschnittlichen Einfügekosten ableiten. Zu den minimalen Kosten kommen bei einem Ausgleich ein Lesezugriff und zwei Schreibzugriffe hinzu. Da bei einem Splitvorgang zusätzlich zwei Seiten gelesen und 3 Seiten geschrieben werden müssen, ergeben sich als durchschnittliche Splitanteile 2/k und 3/k:

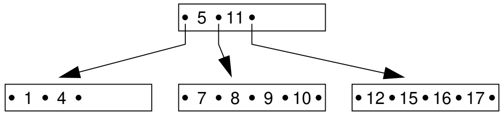
$$f_{avg} \le h + 1 + \frac{2}{k};\ w_{avg} \le 1 + 2 + \frac{3}{k} = 3 + \frac{3}{k}$$



Einfüge 8:



Einfüge 10:



Einfüge 6:

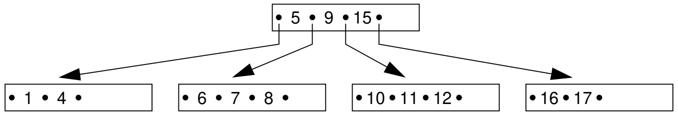


Bild 7: Einfügen in den B-Baum bei doppeltem Überlauf ($\tau(2,2)$)

Der Split-Vorgang läßt sich nun in naheliegender Weise verallgemeinern. Zur Erhöhung der Speicherplatzbelegung könnte bei einem Überlauf erst in der linken Nachbarseite gesucht werden. Wenn sich kein freier Platz lokalisieren läßt, wird in der rechten Nachbarseite gesucht. Erst wenn in den drei zusammenhängenden Seiten kein Platz gefunden wird, kommt es zu einer Spaltung, bei der die Inhalte der drei Seiten auf vier Seiten aufgeteilt werden (Splitfaktor m = 3). Wird nur der Einfügeprozeß betrachtet, so läßt sich hierbei eine minimale Speicherausnutzung von 75% garantieren.

Die Verallgemeinerung des Split-Vorgangs bedeutet, daß erst in m benachbarten Seiten nach freiem Platz gesucht wird, bevor eine Spaltung erfolgt.

Die Suche geschieht dabei abwechselnd im ersten, zweiten, dritten usw. linken und rechten Nachbarn. Es ist leicht einzusehen, daß der Such- und Ausgleichsaufwand in diesem Fall stark ansteigt, so daß im praktischen Einsatz der Splitfaktor auf $m \le 3$ begrenzt sein sollte. Wie aus der Tabelle1 zu entnehmen ist, erhält man dann besonders für die mittlere Speicherplatzbelegung β_{avg} – ermittelt bei Einfügungen in zufälliger

Schlüsselreihenfolge – hervorragende Werte. Die angegebenen Werte für β_{avg} sind untere Grenzwerte, die für $k \to \infty$ erreicht werden.

	Belegung		
Splitfaktor	β_{min}	β_{avg}	β_{max}
1	1/2 = 50%	ln 2 ≈ 69%	1
2	2/3 = 66%	2 · ln (3/2) ≈ 81%	1
3	3/4 = 75%	3 · ln (4/3) ≈ 86%	1
m	$\frac{m}{m+1}$	$m \cdot \ln\left(\frac{m+1}{m}\right)$	1

Tabelle1: Speicherplatzbelegung als Funktion des Splitfaktors

3.2 Suche in der Seite eines Mehrwegbaumes

Neben den Kosten für Seitenzugriffe muß beim Mehrwegbaum der Suchaufwand innerhalb der Seiten als sekundäres Maß berücksichtigt werden. Ein Suchverfahren erfordert eine Folge von Vergleichsoperationen im Hauptspeicher, die bei künftigen Seitengrößen (> 8 K Bytes) sowie 500 oder mehr Schlüssel-Verweis-Paaren pro Seite durchaus ins Gewicht fallen können. Eine Optimierung der internen Suchstrategie erscheint deshalb durchaus gerechtfertigt. Folgende Verfahren lassen sich einsetzen.

Systematische Suche

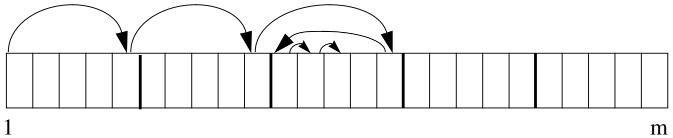
Die Seite wird eintragsweise sequentiell durchlaufen. Bei jedem Schritt wird der betreffende Schlüssel mit dem Suchkriterium verglichen. Unabhängig von einer möglichen Sortierreihenfolge muß im Mittel die Hälfte der Einträge aufgesucht werden. Bei 2k Einträgen sind k Vergleichsschritte erforderlich.

Sprungsuche

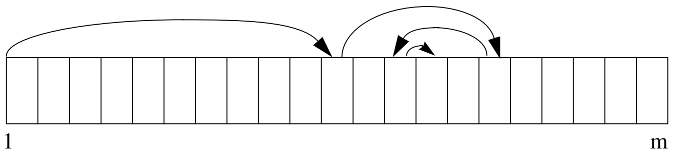
Die geordnete Folge von 2k Einträgen wird in j gleich große Intervalle eingeteilt. In einer ersten Suchphase werden die Einträge jedes Intervalles mit den höchsten Schlüsseln überprüft, um das Intervall mit dem gesuchten Schlüssel zu lokalisieren. Anschließend erfolgt eine systematische Suche im ausgewählten Intervall (Bild 8a). Bei dieser Suchstrategie fallen durchschnittlich (j/2 + k/j) Vergleichsschritte an. Für $j = \sqrt{2k}$ erzielt man hierbei ein Optimum, weshalb sie oft auch als Quadratwurzel-Suche bezeichnet wird.

Binäre Suche

Die binäre Suche setzt wiederum eine geordnete Folge der Einträge voraus. Bei jedem Suchschritt wird durch Vergleich des mittleren Eintrags entweder der gesuchte Schlüssel gefunden oder der in Frage kommende Bereich halbiert (Bild 7.8b). Eine ideale Halbierung läßt sich bei $2k = 2^j - 1$ ($j > 1$) erreichen. Die Anzahl der im Mittel benötigten Vergleichsschritte beträgt angenähert $\log_2(2k) - 1$.



a) Sprungsuche



b) binäre Suche

Bild 8: Suche in einer Seite

Während eine systematische Suche auf Einträgen fester und variabler Länge sowie bei ihrer Komprimierung ausgeführt werden kann, setzen Sprungsuche und binäre Suche Einträge fester Länge voraus. Sie sind lediglich im Falle einer zusätzlichen Indextabelle in der Seite ggf. indirekt einsetzbar. Die Verwendung eines solchen zusätzlichen Index ist jedoch fragwürdig, weil dadurch der für Einträge nutzbare Speicherplatz verkleinert wird.

3.3 Einsatz von variabel langen Schlüsseln

Die Definition des B-Baumes geht von minimal k und maximal 2k Schlüsseln (Einträgen) pro Seite aus, was bei Schlüsseln gleicher Länge besagt, daß jede Seite mindestens halb voll und höchstens voll ist. Weiterhin ist bei Split-Operationen eine gleichförmige Neuverteilung der Schlüssel gewährleistet, während bei Mischoperationen immer garantiert werden kann, daß die betroffenen 2k Einträge Platz in der zugeordneten Seite finden.

Da sich bei variabel langen Einträgen weder ein für alle Seiten gültiger Parameter k noch eine gleichförmige Neuverteilung bei Split-Operationen gewährleisten läßt, ist es naheliegend, das B-Baum-Konzept weiterzuentwickeln und anstelle der strikten Split-Aufteilung Split-Intervalle einzuführen. Bei jedem Split-Vorgang wird ein Split-Intervall um die Mitte der Seite (z. B. 35-65%) festgelegt. Die Neuaufteilung der Einträge im Rahmen des Split-Intervalls erfolgt so, daß eine Seite nur > 35% und die zweite Seite entsprechend < 65% der Einträge aufnehmen kann.

Dieses Verfahren läßt sich durch Parameter für die Größe der Split-Intervalle für Blätter und Nicht-Blätter steuern und separat optimieren. Die Vergrößerung der Split-Intervalle tendiert zwar einerseits durch die zu erzielende Verkürzung der Einträge zur Verringerung der Baumhöhe, erzeugt aber andererseits durch einen geringeren Belegungsgrad in jeweils einer der am Split-Vorgang beteiligten Seiten mehr Seiten als nötig und damit mehr Einträge in den inneren Knoten. Zur Erhöhung des Belegungsfaktors können deshalb bei der Neuaufteilung – ähnlich wie bei der verallgemeinerten Überlaufbehandlung – auch bei variablen Eintragslängen benachbarte Seiten einbezogen werden.

4 B*-Bäume

Die für den praktischen Einsatz wichtigste Variante des B-Baums ist der B*-Baum. Seine Unterscheidungsmerkmale lassen sich am besten von der folgenden Beobachtung her erklären. In B-Bäumen spielen die Einträge (K_i, D_i, P_i) in den inneren Knoten zwei ganz verschiedene Rollen:

- Die zum Schlüssel K_i gehörenden Daten D_i werden gespeichert.
- Der Schlüssel K_i dient als Wegweiser im Baum.

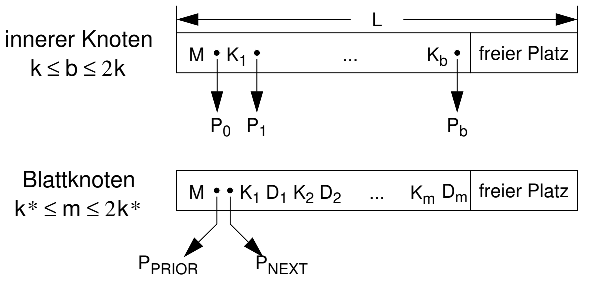
Für diese zweite Rolle ist D_i vollkommen bedeutungslos. In B*-Bäumen wird in inneren Knoten nur die Wegweiser-Funktion ausgenutzt, d. h., es sind nur (K_i, P_i) als Einträge zu führen. Die zu speichernden Informationen (K_i, D_i) werden in den Blattknoten abgelegt. Dadurch ergibt sich für einige K_i eine redundante Speicherung. Die inneren Knoten bilden also einen Index (index part), der einen schnellen direkten Zugriff zu den Schlüsseln gestattet. Die Blätter enthalten alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge. Durch Verkettung aller Blattknoten (sequence set) läßt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte.

Definition: Seien k, k* und h* ganze Zahlen, $h^* \geq 0$, $k, k^* > 0$.
Ein B*-Baum B der Klasse τ (k,k*,h*) ist entweder ein leerer Baum oder ein geordneter Suchbaum, für den gilt:

i. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge.
ii. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.
iii. Jeder innere Knoten hat höchstens 2k+1 Söhne.
iv. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k* und höchstens 2k* Einträge.

Hinweis: Der so definierte Baum wird in der Literatur gelegentlich als B^+-Baum [Co79] bezeichnet.

Es sind also zwei Knotenformate zu unterscheiden:



Die Zeiger P_{PRIOR} und P_{NEXT} dienen zur Verkettung der Blattknoten, damit alle Schlüssel direkt sequentiell aufsteigend und absteigend verarbeitet werden können. Das Feld M enthalte eine Kennung des Seitentyps sowie die Zahl der aktuellen Einträge. Da die Seiten eine feste Länge L besitzen, läßt sich aufgrund der obigen Formate k und k* bestimmen:

$$L = l_M + l_P + 2 \cdot k(l_K + l_P) \; ; \qquad k = \left\lfloor \frac{L - l_M - l_P}{2 \cdot (l_K + l_P)} \right\rfloor$$

$$L = l_M + 2 \cdot l_P + 2 \cdot k^*(l_K + l_D) \; ; \qquad k^* = \left\lfloor \frac{L - l_M - 2l_P}{2 \cdot (l_K + l_D)} \right\rfloor$$

Da in den inneren Knoten weniger Information pro Eintrag zu speichern ist, erhält man beim B*-Baum einen im Vergleich zum B-Baum beträchtlich höheren k-Wert. Ein B*-Baum besitzt also einen höheren Verzweigungsgrad (fan-out), was bei gegebenem n im Mittel zu Bäumen geringerer Höhe führt.

Im Bild 9 ist ein B*-Baum mit derselben Schlüsselmenge wie der B-Baum in Bild 4 dargestellt. Es wird deutlich, daß die Schlüssel im Indexteil redundant gespeichert sind. Als allgemeine Regel gilt, daß jeder Schlüssel im Indexteil eine Kopie des höchsten Schlüssels in seinem linken Unterbaum ist. Nach Löschungen kann diese Regel durchbrochen sein. In jedem Fall gilt jedoch, daß jeder Schlüssel im linken Unterbaum von K_i kleiner oder gleich K_i ist.

4.1 Höhe des B*-Baumes

Die Höhe des B*-Baumes läßt sich in ähnlicher Weise wie beim B-Baum bestimmen. Die Anzahl der Blattknoten bei minimaler Belegung eines B*-Baumes ergibt sich zu

$$B_{min}(k, h^*) = \begin{cases} 1 & \text{für } h^* = 1 \\ 2(k+1)^{h^*-2} & \text{für } h^* \geq 2 \end{cases}$$

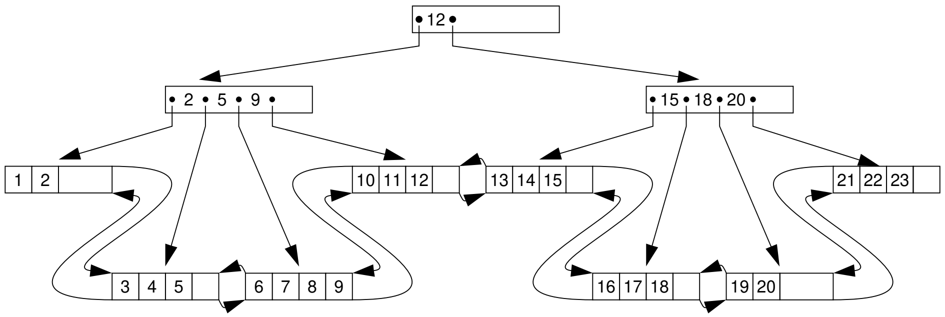


Bild 9: B*-Baum der Klasse $\tau(3,2,3)$

Die Anzahl von Elementen erhalten wir $n_{min}(k, k^*, h^*) = 1$ für h* = 1 und

$$n_{min}(k, k^*, h^*) = 2k^* \cdot (k+1)^{h^*-2} \qquad \text{für } h^* \geq 2 \qquad (9)$$

Bei maximaler Belegung gilt für die Anzahl der Blattknoten

$$B_{max}(k, h^*) = (2k+1)^{h^*-1} \qquad \text{für } h^* \geq 1$$

und für die Anzahl der gespeicherten Elemente

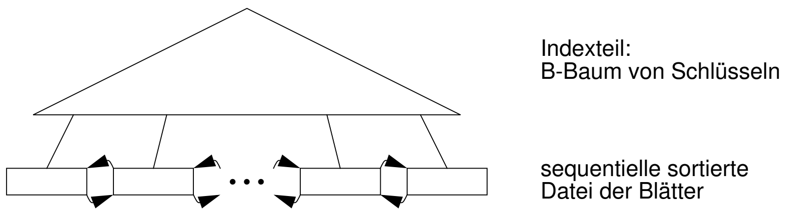
$$n_{max}(k, k^*, h^*) = 2k^* \cdot (2k+1)^{h^*-1} \qquad \text{für } h^* \geq 1 \qquad (10)$$

Mit Hilfe von (9) und (10) läßt sich leicht zeigen, daß die Höhe h* eines B*-Baumes mit n Datenelementen begrenzt ist durch

$$1 + \log_{2k+1}\frac{n}{2k^*} \leq h^* \leq 2 + \log_{k+1}\frac{n}{2k^*} \qquad \text{für } h^* \geq 2 .$$

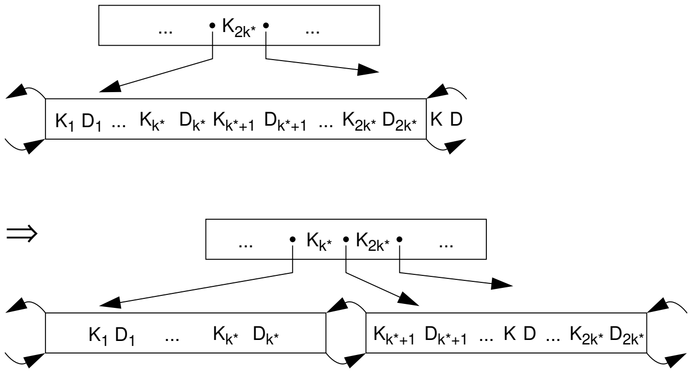
4.2 Grundoperationen beim B*-Baum

Der B*-Baum kann aufgefaßt werden als eine gekettete sequentielle Datei von Blättern, die einen Indexteil besitzt, der selbst ein B-Baum ist. Im Indexteil werden insbesondere beim Split-Vorgang die Operationen des B-Baums eingesetzt. Diese besondere Sicht des B*-Baumes läßt sich wie folgt veranschaulichen:



Wie bereits erwähnt, ist h* im Mittel kleiner als h in B-Bäumen. Da alle Schlüssel in den Blättern sind, kostet jede direkte Suche h* Zugriffe. Da f_{avg} beim B-Baum in guter Näherung mit h abgeschätzt werden kann, erhält man also durch den B*-Baum eine effizientere Unterstützung der direkten Suche. Die sequentielle Suche erfolgt nach Aufsuchen des Linksaußen der Struktur unter Ausnutzung der Verkettung der Blattseiten. Es sind zwar ggf. mehr Blätter als beim B-Baum zu verarbeiten, doch da nur h*-1 innere Knoten aufzusuchen sind, wird die sequentielle Suche ebenfalls effizienter ablaufen.

Das Einfügen ist von der Durchführung und vom Leistungsverhalten her dem Einfügen in einen B-Baum sehr ähnlich. Bei inneren Knoten wird die Spaltung analog zum B-Baum durchgeführt. Der Split-Vorgang einer Blattseite läuft nach folgendem Schema ab:



Beim Split-Vorgang muß gewährleistet sein, daß jeweils die höchsten Schlüssel einer Seite als Wegweiser in den Vaterknoten kopiert werden. Außerdem muß die neue Seite in die verkettete Liste der Blattknoten aufgenommen werden. Die Verallgemeinerung des Split-Vorgangs läßt sich analog zum B-Baum einführen.

Der Löschvorgang in einem B*-Baum ist einfacher als im B-Baum. Datenelemente werden immer von einem Blatt entfernt. Die komplexe Fallunterscheidung zum Löschen eines Elementes aus einem inneren Knoten entfällt also beim B*-Baum. Weiterhin muß beim Löschen eines Schlüssels aus einem Blatt dieser Schlüssel nicht aus dem Indexteil entfernt werden; er behält seine Funktion als Wegweiser. Ein Schlüssel K_i im Indexteil muß nur als Separator zwischen seinem linken und rechten Unterbaum dienen. Deshalb ist jeder beliebige String, der diese Funktion erfüllt, zulässig. Einige Einfüge- und Löschoperationen sind für einen B*-Baum in Bild 10 gezeigt.

4.3 Vergleich von B- und B*-Baum

Zusammenfassend seien noch einmal die wichtigsten Unterschiede des B*-Baums zum B-Baum und seine Vorteile aufgelistet:

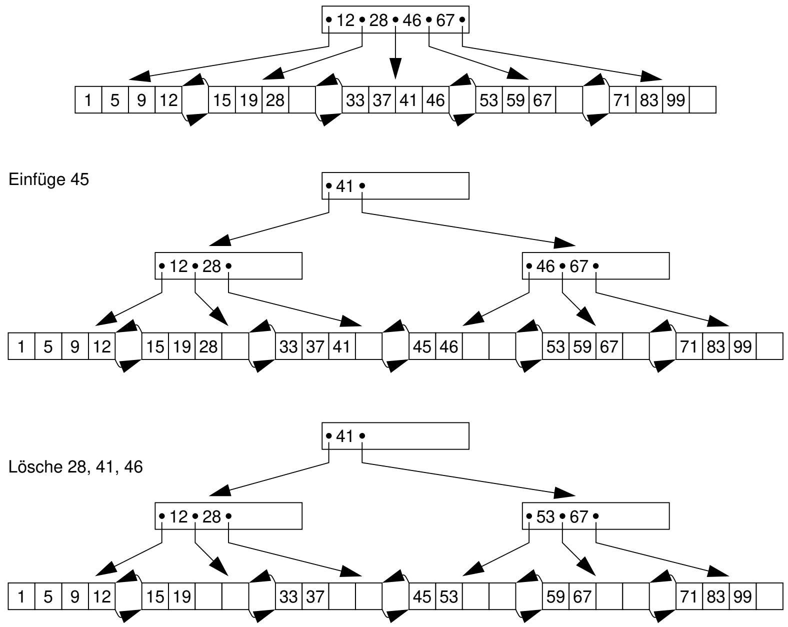


Bild 10: Einfügen und Löschen im B*-Baum: $\tau(2,2,h^*)$

- Es wird eine strikte Trennung zwischen Datenteil und Indexteil vorgenommen. Datenelemente stehen nur in den Blättern des B*-Baumes.
- Schlüssel im inneren Knoten haben nur Wegweiserfunktion. Sie können auch durch beliebige Trenner ersetzt oder durch Komprimierungsalgorithmen verkürzt werden.
- Kürzere Schlüssel oder Trenner in den inneren Knoten erhöhen den Verzweigungsgrad des Baumes und verringern damit seine Höhe.
- Die redundant gespeicherten Schlüssel erhöhen den Speicherplatzbedarf nur geringfügig (< 1%).
- Der Löschalgorithmus ist einfacher.
- Durch die Verkettung der Blattseiten wird eine schnellere sequentielle Verarbeitung erzielt.

Anhand eines praktischen Beispiels sollen für B- und B*-Baum die Mengen der Elemente, die sich in einem Baum vorgegebener Höhe abspeichern lassen, gegenübergestellt werden. Die Seitengröße sei L= 2048 Bytes. Jeder Zeiger P_i, jede Hilfsinformation und jeder Schlüssel K_i seien 4 Bytes lang. Bezüglich der Daten wird folgende Fallunterscheidung gemacht:

- bei eingebetteter Speicherung ist l_D = 76 Bytes
- bei separater Speicherung ist l_D = 4 Bytes, d. h., im Baum wird nur ein Zeiger auf den Datensatz gespeichert.

Für den B-Baum erhalten wir

- bei eingebetteter Speicherung: k = 12;
- bei separater Speicherung: k = 85;

Als Parameterwerte für den B*-Baum ergeben sich

- bei eingebetteter Speicherung: k = 127; k* = 12;
- bei separater Speicherung: k = 127; k* = 127;

Die Gleichungen (4), (5), (9) und (10), die für minimale und maximale Belegung gelten, seien nochmals zusammengestellt:

	B-Baum	B*-Baum
n_{min}	$2 \cdot (k+1)^{h-1} - 1$	$2k^* \cdot (k+1)^{h^*-2}$
n_{max}	$(2k+1)^h - 1$	$2k^* \cdot (2k+1)^{h^*-1}$

Damit ergeben sich folgende Zahlenwerte für den B-Baum:

	Datensätze separat (k=85)		Datensätze eingebettet (k=12)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	170	1	24
2	171	29.240	25	624
3	14.791	5.000.210	337	15.624
4	1.272.112	855.036.083	4.393	390.624

Die entsprechenden Werte für den B*-Baum sind in der folgenden Tabelle zusammengefaßt:

	Datensätze separat (k=127, k *= 127)		Datensätze eingebettet (k=12, k *= 127)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	254	1	24
2	254	64.770	24	6.120
3	32.512	16.516.350	3.072	1.560.600
4	4.161.536	4.211.669.268	393.216	397.953.001

Es ist klar, daß sich die Wertebereiche von B- und B*-Baum für eine bestimmte Höhe überlappen. Jedoch ist die deutliche Überlegenheit des B*-Baumes zu erkennen. Besonders deutlich tritt sie im Fall eingebetteter Datensätze hervor. Trotz der Redundanz im Indexteil stellt der B*-Baum die effizientere Struktur dar.

Literatur

BMc72 R. Bayer, McCreigtht, E.M.: Organization and maintenance of large ordered indexes, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173-189.

Co79 D. Comer: The ubiquitous B-tree, ACM Computer Surveys, Vol. 11, No. 2, 1979, pp. 121-137.

Kn73 D.E. Knuth: The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley Publishing Co., Reading, Mass., 1973.