TU-München

Hauptseminar Informatik

Database Hall of Fame

Wintersemester 2001/2002

Prof. R. Bayer:

B-Bäume

Verfasser: Moritz Theile
Betreuer: Prof. Dr. D. Kossmann
Vortragstermin: 27.11.01

1 Inhaltsverzeichnis

1 Inhaltsverzeichnis 2
2 Motivation 3
3 B-Bäume 5
3.1 Definition eines B-Baumes 5
3.2 Die Operation find(K) 6
3.3 Die Operation insert(K) 7
3.4 Die Operation delete(K) 8
3.5 Die Kosten der Operationen 11
3.6 Die Wahl von k 12
4 B*-Bäume 13
4.1 Unterschied zwischen B- und B*-Bäumen 13
4.2 Vor- und Nachteile von B*-Bäumen 13
5 B+-Bäume 13
5.1 Punktsuche und Bereichssuche 13
5.2 Merkmal eines B+-Baumes 14
5.3 Die next()-Operation auf einem B+-Baum 14
6 Parallele Operationen auf einem B-Baum 15
6.1 Das Problem 15
6.2 Naiver Ansatz 16
6.3 Sichere und unsichere Knoten 17
6.4 Lock-Coupling 18
7 Literaturverweise 21

2 Motivation

Die Aufgabe von Datenbanken ist die Verwaltung von großen Datenmengen. D.h. man will verschiedene Operationen auf diesen Daten möglichst schnell ausführen können. Man muss die Informationen also in einer Datenstruktur abspeichern, die die entsprechenden Operationen in einer möglichst effizienten Weise unterstützen.

Um ein greifbares Beispiel zu haben, stelle man sich einen Datensatz vor, der alle Bundesbürger mit Telefonnummer und Adresse erfasst. (siehe Abb.1) Es handelt sich also quasi um ein bundesweites Telefonbuch.

Um diese Datenbank nun auf aktuellem Stand zu halten, sind einige Operationen notwendig. Wird z.B. ein Anschluss abgemeldet, muss der entsprechende Eintrag gelöscht werden. Angenommen der Datensatz liegt in einer Liste alphabetisch nach Namen geordnet vor und man kennt den Namen des zu löschenden Eintrages, dann könnte man die Liste relativ schnell nach dem gesuchten Element durchsuchen und es dann löschen. Wenn man aber bloß die Telefonnummer gegeben hätte, müsste man den ganzen Datensatz sequentiell durchsuchen, was in einer völlig inakzeptablen Suchzeit resultieren würde.

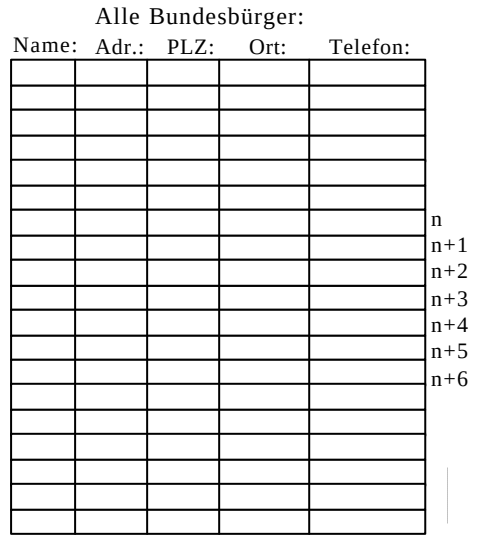
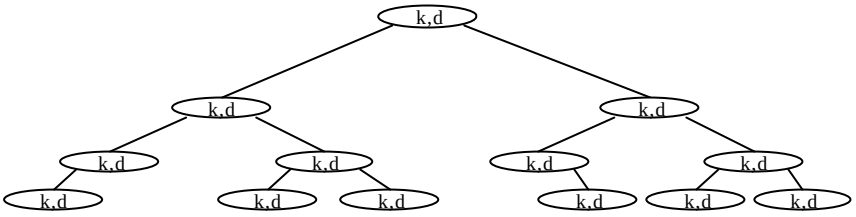


Abbildung 1

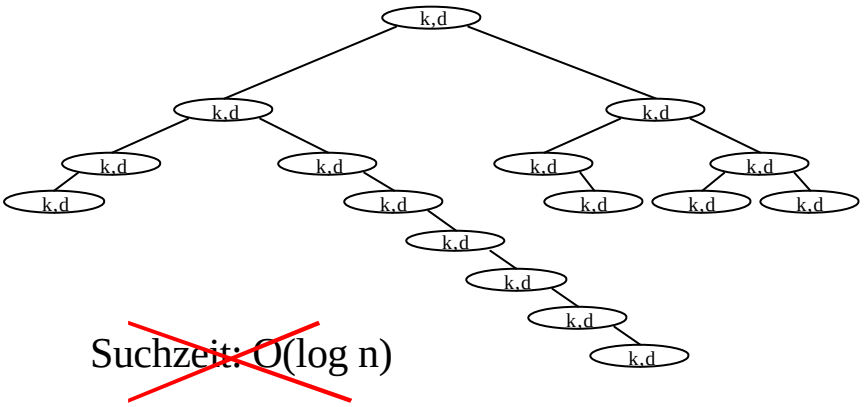
Um für beide Fälle gerüstet zu sein, legt man für jede Spalte, in der evtl. gesucht werden muss, einen Index an. D.h. man generiert für jeden Record ein Tupel (k,d) wobei k der Schlüssel und d der Zeiger auf den Record ist. In unserem Fall könnte z.B. die Telefonnummer als Schlüssel dienen.
Jetzt muss man nicht mehr den Datensatz selbst durchsuchen, sondern muss nur aus der Menge der Tupel das richtige Element finden. Damit dies schnell geht, speichert man die Tupel in geeigneter Form ab.

Ein naiver Ansatz wäre, sie in einem binären Suchbaum abzulegen. (siehe Abb.2) Hiermit wäre eine logarithmische Suchzeit erreicht.
Wenn man aber Update-Funktionen wie delete(x) oder insert(x) ausführt, kann der Baum schnell entarten und eine schnelle Suche ist nicht mehr gewährleistet.
In unserem Beispiel könnte die Telekom z.B. ein paar hintereinanderliegende Nummern vergeben, und der Baum könnte bald ungefähr so aussehen wie in Abb.3.



Suchzeit: O(log n)

Abbildung 2



~~Suchzeit: O(log n)~~

Abbildung 3

Es ist also naheliegend, einen höhenbalancierten Baum zu nutzen. Eine Ausprägung von einem höhenbalancierten Baum stellt der AVL-Baum dar. Man hätte das Problem der Entartung umgangen, wenn man ihn zur Verwaltung der Index-Tupel verwendet hätte. Die Rebalancierungsmaßnahmen im AVL-Baum erzeugen aber leider Adressfolgen, die keinerlei Lokalität aufweisen. Dies ist nicht weiter schlimm, wenn sich der gesamte Baum im Hauptspeicher des Computers befindet. Es wirkt sich aber katastrophal auf die Laufzeit aus, wenn häufig Daten aus dem langsamen Hintergrundspeicher nachgeladen werden müssen, weil nicht vorauszusehen ist, welche Daten für die nächste Operation wahrscheinlich gebraucht werden.[INFOHANDB]
B-Bäume sind eine Datenstruktur, die für die skizzierte Problemstellung optimal sind.

3 B-Bäume

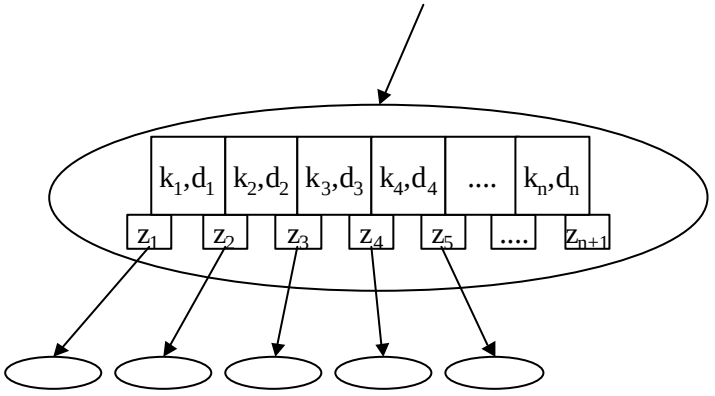
B-Bäume wurden von Prof. Rudolf Bayer explizit für die Plattenspeicherverwaltung entworfen. Für seine Arbeiten rund um den B-Baum und andere Verdienste erhielt Prof. Rudolf Bayer dieses Jahr (2001) den „SIGMOD Innovations award“.

Wesentliche Merkmale eines B-Baumes ist seine Höhenbalanciertheit und die Lokalität von Operationen.

3.1 Definition eines B-Baumes

Alle Wege von der Wurzel zu den Blättern sind gleich lang.

1.) Jeder Knoten hat mindestens k und maximal 2k Einträge. Ausnahme: die Wurzel kann zwischen 1 und 2k Einträge haben.
2.) Wenn ein Knoten n Einträge hat, hat er n+1 Verweise auf seine Söhne. Ausnahme: Blätter haben keine bzw. undefinierte Verweise.
3.) Ein Blatt muss folgendermaßen organisiert sein: (siehe Abb.4)
    - Wenn k_1 bis k_n sind Schlüssel sind, sind z_1 bis z_{n+1} Zeiger auf Söhne.
    - z_1 zeigt auf Teilbaum mit Schlüsseln kleiner k_1
    - z_i (i=1...n) zeigt auf Teilbaum mit Schlüsseln zwischen k_i und $k_{i+1}$$z_{i+1}$ zeigt auf Teilbaum mit Schlüsseln grösser



k_{i+1}

Abbildung 4 Ein B-Baum-Blatt.

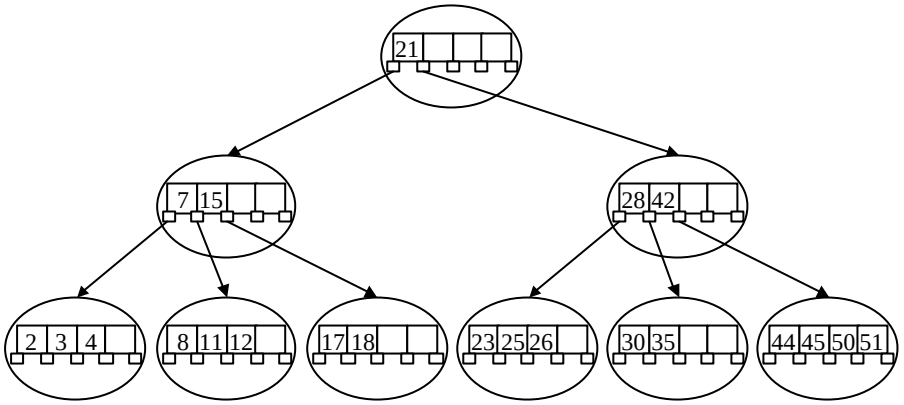


Abbildung 5 Ein Beispiel für einen B-Baum

Wie man an der Definition sieht, kann jedes Blatt eine unterschiedliche Anzahl von Einträgen haben. Dies führt dazu, dass meistens nur ein sehr lokaler Aufwand getrieben werden muss, um den Baum in Balance zu halten. Warum das so ist, kann man besser begreifen, wenn man sich über die Operationen ein paar Gedanken gemacht hat, die wir im folgenden besprechen werden.

3.2 Die Operation find(k)

Der Suchalgorithmus auf einem B-Baum ist nicht sehr kompliziert. Er fängt an der Wurzel an zu suchen. Wenn man das gewünschte Element nicht findet, durchsucht man den in Frage kommenden Kind-Knoten usw.

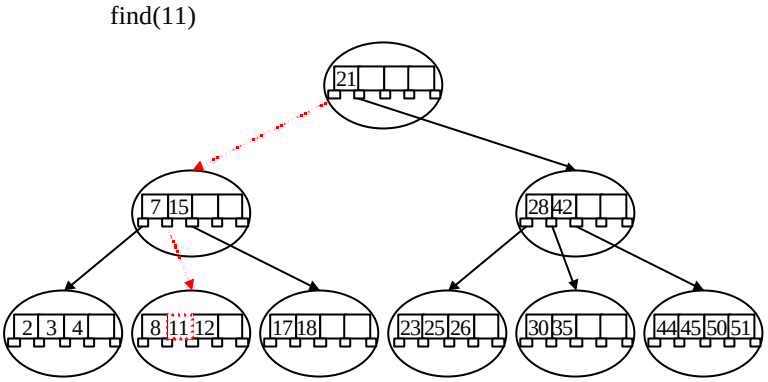


Abbildung 6 Der Suchpfad von einem find(11).

3.3 Die Operation insert((k,d))

Der Einfügealgorithmus lautet wie folgt:

```
INSERTION OF KEY 'K'
   find the correct leaf node 'L';
   if ( 'L' overflows ){
      split 'L', by pushing the middle key
         upstairs to parent node 'P';
      if ('P' overflows){
         repeat the split recursively;
      }
   else{
      add the key 'K' in node 'L';
   }
```

[CMU]

Zuerst sucht man sich den richtige Stelle zum Einfügen des Schlüssels. Wenn in dem Knoten weniger als 2k Einträge waren ist man fertig.
Sind vor dem Einfügen schon 2k Einträge in dem Knoten, muss man ein sog. „splitting“ von dem „überfüllten“ Knoten durchführen. Diese Situation ist exemplarisch in Abb.7 u. 8 dargestellt. Bei einem „splitting“ nimmt man die k kleinsten Einträge (44,45) und die k größten Einträge (50,51) aus dem „überfüllten“ Knoten und bildet damit 2 neue Knoten. Den verbleibenden Schlüssel (47) fügt man in den Vaterknoten ein. Wenn dieser Schlüssel zum Eintrag k_i wird, zeigt der Zeiger z_i auf den Knoten mit den kleineren Schlüsseln und der Zeiger z_{i+1} auf den mit den größeren.
Wenn der Vaterknoten auch schon voll gewesen wäre, hätte sich dieses Splitting rekursiv in Richtung Wurzel fortgesetzt. Wenn also alle Knoten auf dem Pfad von der Wurzel bis zum gesuchten Knoten voll besetzt wären, würde die Höhe des Baumes um eins zunehmen.[COMER]

insert(45)

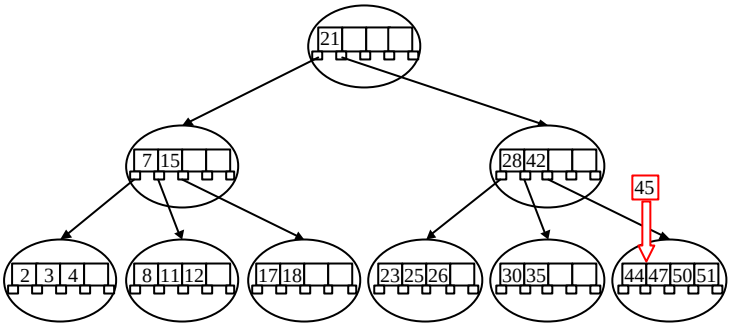


Abbildung 7 Baum A vor einem insert(45).

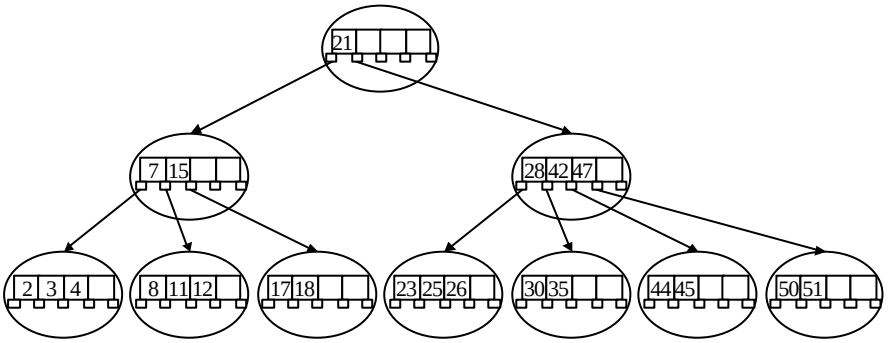


Abbildung 8 Baum A nach einem insert(45)

3.4 Die Operation delete(k)

Der Löschalgorithmus ist hier beschrieben:

```
locate key 'K', in node 'N'
if( 'N' is a non-leaf node){
   delete 'K' from 'N';
   find the immediately largest key 'K1';
      /* which is guaranteed
         to be on a leaf node 'L' */
   copy 'K1' in the old position of 'K';
   invoke this DELETION routine on 'K1' from the leaf node 'L';
else{
   /* 'N' is a leaf node */
   if( 'N' underflows ){
      let 'N1' be the sibling of 'N';
      if( 'N1' is "rich"){
         /* ie., N1 can lend us a key */
```

```
            borrow a key from 'N1'
              THROUGH the parent node;
         }else{
            /* N1 is 1 key away from underflowing */
            MERGE: pull the key from the parent 'P',
               and merge it with the keys of 'N' and 'N1'
               into a new node;
            if( 'P' underflows){ repeat recursively }
         }
   }
```

[CMU]

Bei einem Löschvorgang muss man erst einmal unterscheiden, ob das zu löschende Element in einem Blatt oder in einem anderen Knoten ist. Wenn es in einem Knoten ist, kann man den nun fehlenden Eintrag ersetzen, indem man z.B. den kleinsten Schlüssel aus dem rechten Unterbaum nimmt.
Nachdem man jetzt einen Schlüssel aus einem Blatt entfernt hat, muss man überprüfen, ob das Blatt noch mindestens k Einträge hat. Falls dies nicht der Fall ist, muss man die Nachbarblätter zu Hilfe nehmen, um wieder zu einem legalen B-Baum zu gelangen.
Es gibt zwei Möglichkeiten. Wenn in dem Blatt und seinem Nachbarn zusammen mehr als 2k Einträge vorhanden sind, kann man eine „redistribution“ durchführen. D.h. man teilt die Einträge in den beiden Blättern einfach gleichmäßig auf. Dieser Fall ist in Abb. 9, 10, 11 illustriert.

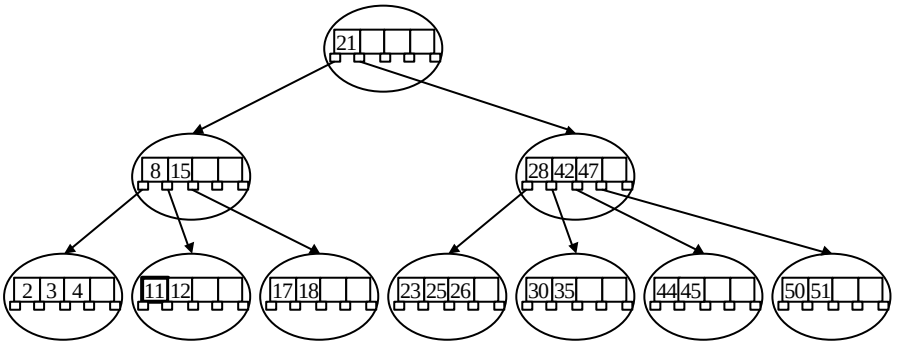


Abbildung 9 Baum B vor einem delete(8**)**

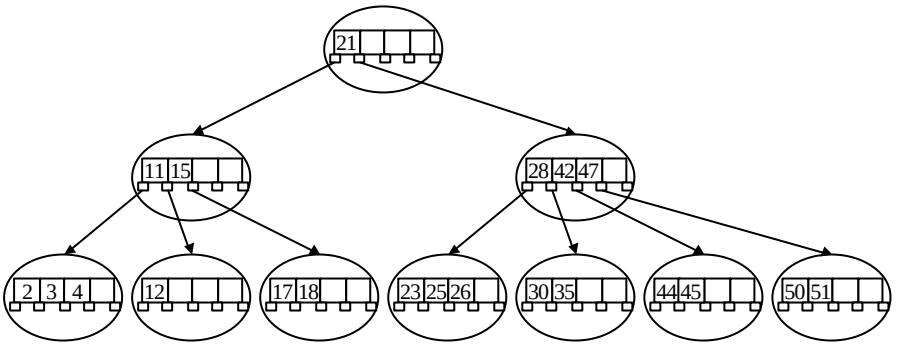


Abbildung 10 Nachdem der leere „slot“ durch den 11er-Schlüssel aufgefüllt wurde, verletzt das 2. Blatt von links die B-Baum-Definition

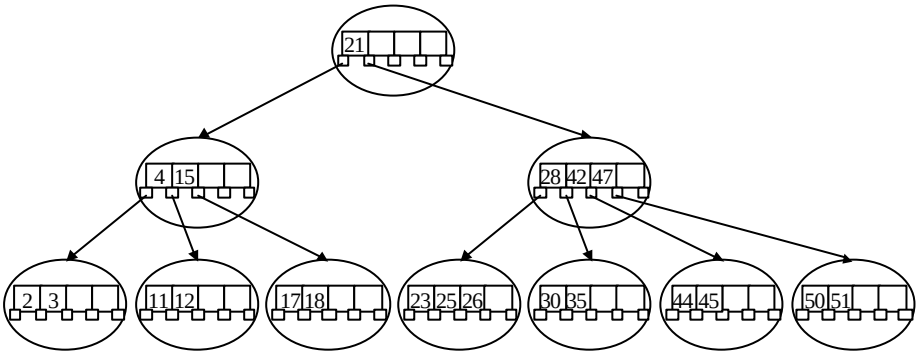


Abbildung 11 Baum B nach der „redistribution“

Wenn weniger als 2k Einträge vorhanden sind, werden die beiden Blätter miteinander verschmolzen. Das ist dann die sog. „concatenation“. Dabei werden die Einträge aus zwei benachbarten Blättern in eines geschrieben. Da der Vaterknoten nun über einen Sohn weniger verfügt, hat er natürlich auch einen Eintrag weniger. Würde bei ihm nun wiederum die Mindestanzahl der Einträge unterschritten, würde eine erneute „concatenation“ erfolgen.
Die „concatenation“ ist das Gegenstück zum „splitting“ welches bei einer Einfügeoperation auftreten kann.
Die „concatenation“ ist anhand er Abb. 12 und 13 illustriert.

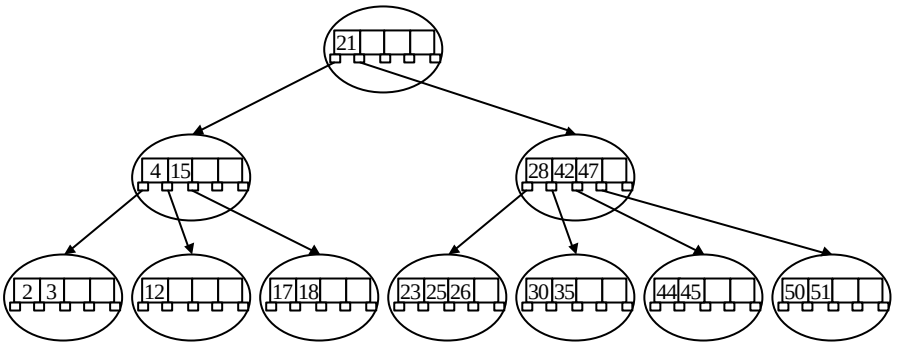


Abbildung 12 Bei diesem Baum verletzt wiederum das 2. Bl v. links die B-Baum-Def. Diesmal ist aber keine „redistribution“ möglich.

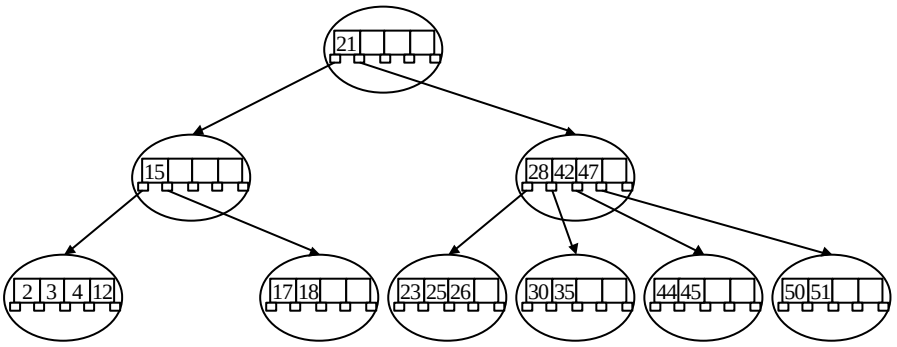


Abbildung 13 Dies ist der Baum aus obiger Abb. nach einer „concatenation“. Allerdings verletzt jetzt der Knoten mit Schlüssel 15 die Bedingungen. Der Baum wäre erst nach einer „redistribution“ wieder ein B-Baum.

3.5 Die Kosten der Operationen

Die Kosten von den Operationen spiegeln sich v.a. in der Anzahl der Zugriffe auf den Hintergrundspeicher wieder. Man geht davon aus, dass mindestens ein Knoten zur Zeit in den Hauptspeicher geladen werden kann. Die Suche im Knoten selbst (Suche auf einer sortierten Liste) kann man vernachlässigen.
Für eine find()-Operation im „worst-case“ (gesuchtes Element ist in einem Blatt) würde das also bedeuten, dass ihre Kosten linear zur Höhe des Baumes steigen würde. Diese ist logarithmisch.(Abb. 15)
Betrachtet man Update-Funktionen, so stellt man fest, dass sie zwar teurer sein können, da sie den Baum wieder hinauflaufen, aber sie steigen nicht überproportional schnell zur Höhe h.

Warum man k nicht einfach unendlich wählen sollte, wird im folgenden erläutert.

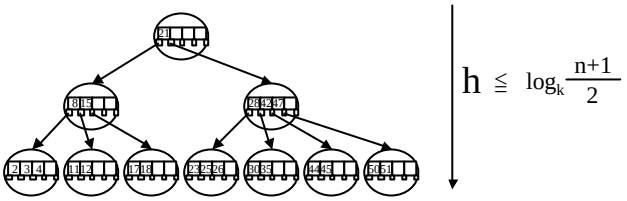


Abbildung 14

3.6 Die Wahl von k

Die Wahl von k ist abhängig von der verwendeten Hardware. Sie richtet sich danach, wie auf den Hintergrundspeicher zugegriffen wird und wie groß der Arbeitsspeicher ist.
Ziel ist es, möglichst wenig Zeit mit dem Zugriff auf den langsamen Hintergrundspeicher zu verbringen.
Je mehr Einträge in einem Knoten sind, desto flacher wird der Baum und desto weniger Zugriffe auf den Hintergrundspeicher sind notwendig. Allerdings dauern die Zugriffe auf den Hintergrundspeicher mit größerem k auch entsprechend länger. Außerdem muss man bedenken, dass die Größe der Knoten durch die Größe des Arbeitsspeichers begrenzt ist. Wenn die Operationen nicht total lokal in einem Blatt ablaufen, kann es auch sinnvoll sein, noch ein paar vorhergehende Knoten im Speicher zu haben, um sie nicht wieder neu laden zu müssen. Es gibt also eine Vielzahl von Faktoren, die die optimale Größe von k beeinflussen.
Um herauszubekommen wie groß k nun tatsächlich für ein bestimmtes System gewählt werden muss, gibt es Richtlinien, die u.a. von Herrn Bayer vorgeschlagen wurden.

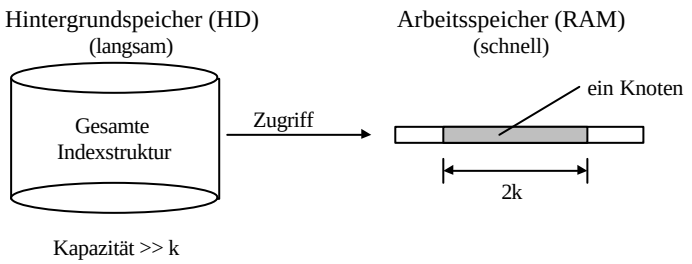


Abbildung 15 Die richtige Wahl von k ist für die spätere Performance des Systems von wesentlicher Bedeutung.

4 B*-Bäume

4.1 Unterschied zwischen B- und B*-Bäumen

B*-Bäume unterscheiden sich von B-Bäumen in der Definition nur in Punkt 2.
Während bei B-Bäumen jedes Blatt zwischen k und 2k Einträge haben muss, müssen die Knoten von B*-Bäume mindestens zu 2/3 aufgefüllt sein.

4.2 Vor- und Nachteile von B*-Bäumen

Wenn Knoten minimal besetzt sind, haben Knoten von B*-Baum mehr Söhne als Knoten von B-Bäumen. Wenn alle Knoten minimal besetzt sind („worst-case"), führt das natürlich dazu, dass die B*-Bäume flacher als die normalen B-Bäume.
Außerdem garantieren sie eine bessere Speicherauslastung.
Es hat aber auch Nachteile, wenn man die Knoten immer sehr voll macht.
Angenommen man würde bei einem B-Baum vorschreiben, dass immer mindestens 2k-1 Einträge vorhanden sein müssen, dann heißt das, dass z.B. eine Einfügeoperation mit einer sehr hohen Wahrscheinlichkeit (etwa 50%) auf einen vollen Knoten trifft. Dies hätte ein „splitting" zur Folge, welches sich wiederum mit hoher Wahrscheinlichkeit rekursiv in Richtung Wurzel fortsetzen würde.
Man kann also erahnen, dass bei B*-Bäume die Operationen im Schnitt weniger lokal ablaufen als bei B-Bäumen.
Als weiteren Nachteil könnte man vielleicht auch noch die etwas aufwändigeren Operationen nennen.

5 B+-Bäume

5.1 Punktsuche und Bereichssuche

Um einen wesentlichen Vorteil von B+-Bäumen besser erläutern zu können, müssen erst einmal die Begriffe Punktsuche und Bereichssuche erläutert werden.
Unter Punktsuche versteht man eigentlich genau die Funktionalität, die durch die find(key) Operation gegeben ist.
Bei der Bereichssuche will man allerdings nicht nur ein Element finden sondern Records, die bezüglich ihrer Schlüssel hintereinander liegen.
Eine Bereichsanfrage -bezogen auf das Bundesbürger-Beispiel vom Anfang- wäre z.B.: „Gib mir alle Adressen, denen eine Telefonnummer zugeordnet ist, die mit 089 beginnt!"
Diese Aufgabe kann nur gelöst werden, wenn auf der Datenstruktur auch eine Funktion next() definiert ist.

5.2 Merkmal eines B+-Baumes

Bei einem B+-Baum unterscheidet sich die Struktur von den Knoten die keine Blätter sind von denen, die Blätter sind. Ich nenne die beiden hier einfach Blätter und Knoten.
Die Einträge in den Knoten enthalten keine Verweise auf die Records. Sie müssen nicht einmal den Schlüsseln entsprechen, aber sie müssen es der find()-Operation ermöglichen, zu dem richtigen Blatt zu finden. Diese enthalten dann das Tupel aus Schlüssel und Zeiger.
Die Verkettung der Blätter von links nach rechts ist ein weiteres wichtiges Merkmal von B+-Bäumen.
Der schematische Aufbau ist in Abb. 16 dargestellt.

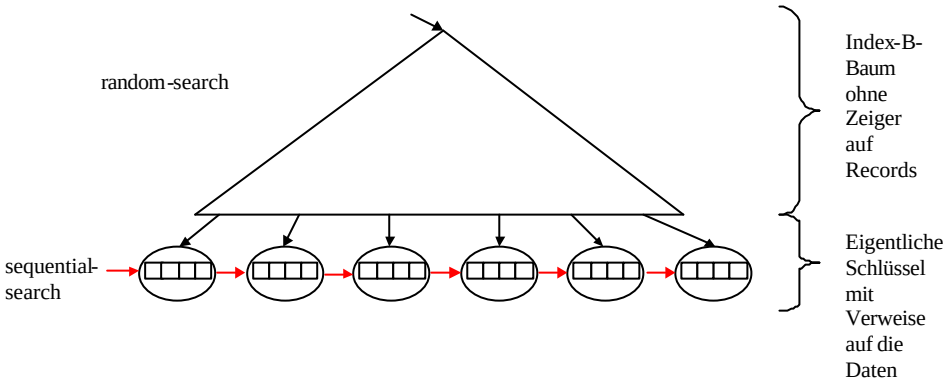


Abbildung 16 Schematische Darstellung eines B+-Baumes.

5.3 Die next()-Operation auf einem B+-Baum

Der Vorteil dieser Datenstruktur kommt vor allem bei der sequentiellen Suche/Ausgabe von Datensätzen zum tragen. Während bei einem B-Baum ein Knoten bis zu 2k mal eingelesen wird, ist bei einem B+-Baum garantiert, dass jeder Knoten max. einmal geladen wird. Wenn die next-Operation am Ende eines Blattes angekommen ist, kann sie sich einfach zum nächsten Blatt „vorhangeln“ und die Einträge dort sequentiell auslesen.
Bei einem B-Baum hingegen springt die Funktion am Ende eines Blattes wieder in den Vaterknoten, liest dort den nächsten Eintrag, springt dann zum nächsten Blatt, um danach wieder zum Vaterknoten zu springen. Der Vaterknoten wurde also mehr als einmal besucht.

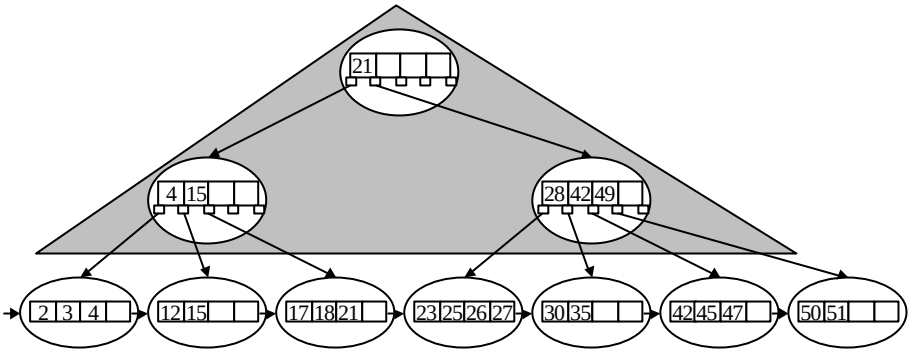


Abbildung 17 Beispiel für einen B+-Baum.

6 Parallele Operationen auf einem B-Baum

6.1 Das Problem

In einer Multiuserumgebung kann es bei gleichzeitigem Zugriff auf einen Baum zu Problemen kommen. Ein exemplarischer Problemfall ist in Abb. 18 zu sehen.
Zwei Prozesse, einer lesend (T1), der andere einfügend(T2), greifen parallel auf einen Baum zu. Ein Problem entsteht, weil in dem Moment als T1 den Zeiger auf Knoten B bekommen hat, T2 aktiv wird und den Knoten B in 2 Knoten aufteilt. Danach wird T1 wieder aktiv und wird melden, dass er das gesuchte Element nicht gefunden hat, obwohl es in dem Baum eigentlich vorhanden gewesen wäre.

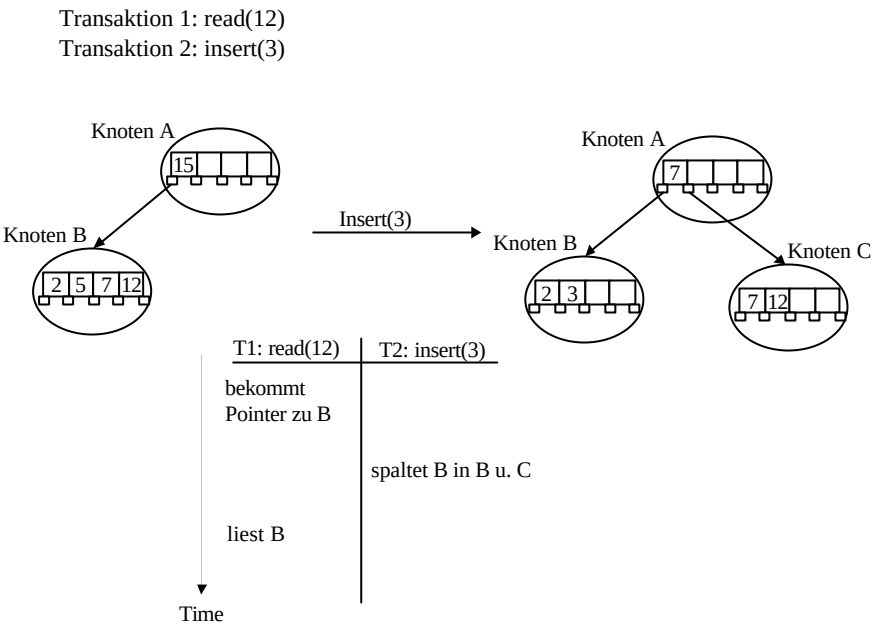


Abbildung 18

6.2 Naiver Ansatz

Auch hier betrachten wir zuerst wieder einen naiven Ansatz für das Verständnis.
Die einfachste Möglichkeit ist, alle Zugriffe streng zu serialisieren, z.B. indem jeder Prozess eine exklusive Sperre auf die Wurzel setzt. Diese würde alle anderen Prozesse davon abhalten, den Baum zu verändern oder auch nur zu lesen.
Ein wenig besser ist die Methode, die zumindest parallele Schreibzugriffe erlaubt. Ein Lesezugriff sperrt die Wurzel mit einem S (für die Bedeutung siehe Abb.19), was anderen Prozessen signalisiert, dass keine Schreibfunktionen ausgeführt werden dürfen. Ein schreibender Zugriff setzt ein X, was überhaupt keinem anderen Prozess Zugriff auf die Datenstruktur erlaubt.(siehe Abb. 20)
Wenn auf dem Baum jetzt nur Leseoperationen ausgeführt werden ist diese Methode sehr gut, da alle Leseoperationen parallel lesen können. Allerdings hätte man dann auch kein Protokoll gebraucht...
Wenn nun Schreibprozesse mit ins Spiel kommen, sind diese immer noch streng serialisiert. Außerdem tritt das Problem des Verhungerns auf. Wenn sich immer mindestens ein Leseprozess im Baum befindet, hat ein Schreibprozess nie die Chance seine Operation durchzuführen.

Beantragter Sperr-Modus	Momentaner Sperr-Modus	
	S	X
S	OK	
X		

S: „shared“ lock
X: „exclusive“ lock

Abbildung 19 Kompatibilitätsmatrix

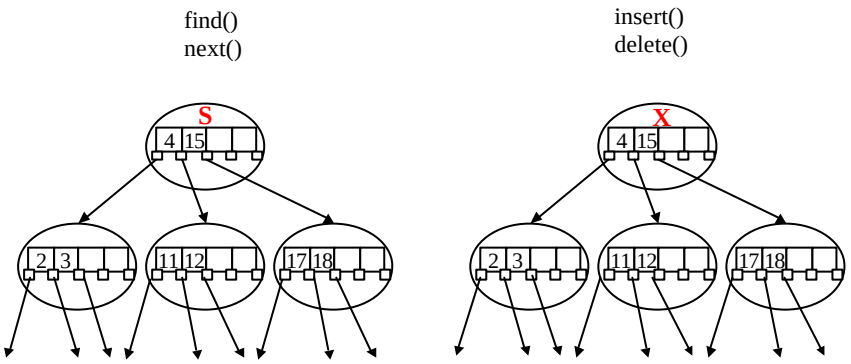


Abbildung 20 Einfache Möglichkeit: Man setzt die Sperren einfach an die Wurzel.

6.3 Sichere und unsichere Knoten

Die Protokolle, die einen sicheren parallelen Zugriff auf B-Bäume gewährleisten, basieren häufig auf der Erkennung von sicheren Knoten. Wenn ein Knoten bezüglich eines Prozesses sicher ist, darf nur noch er und seine Unterbäume von diesem Prozess verändert werden. D.h. andere Prozesse können den nicht „abgeschlossenen“ Teil ohne Konflikte bearbeiten.
Ob ein Knoten sicher ist, hängt von der Anzahl der enthaltenen Elemente und der Art des Prozesses ab.

Ein Knoten ist bezüglich einer Einfügeoperation dann sicher, wenn die Anzahl seiner Einträge kleiner als 2k ist.Handelt es sich wie in Abb. 21 um einen insert()-Prozess, ist das graue Blatt nicht sicher, da ein Insert von z.B. 45 ein Splitting zur Folge hätte, welches auch den Inhalt des Vaterknotens verändern würde. Alle anderen Knoten sind sicher, weil ein weiteres Element keine Auswirkungen auf die Knoten hätte, die sich auf dem Weg zur Wurzel befinden.

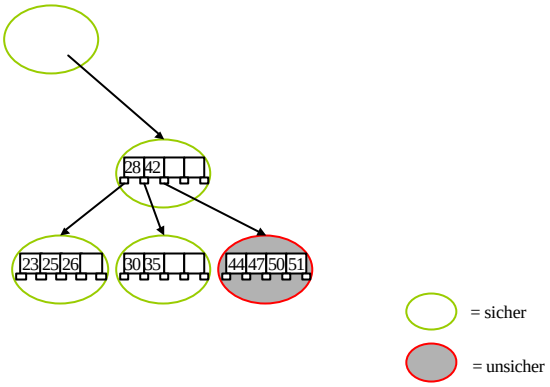


Abbildung 21 Sichere und unsichere Knoten bezüglich Einfügeoperationen.

Ein Knoten ist bezüglich einer Löschoperation dann sicher, wenn die Anzahl seiner Einträge größer als k ist.Handelt es sich wie in Abb. 22 um einen delete()-Prozess, sind die grauen Blätter nicht sicher, da ein delete() von z.B. 30 eine „concatenation“ zur Folge hätte, welche auch den Inhalt der Vaterknoten verändern würde. Alle anderen Knoten sind sicher, weil ein Element weniger keine Auswirkungen auf die Knoten hätte, die sich auf dem Weg zur Wurzel befinden.

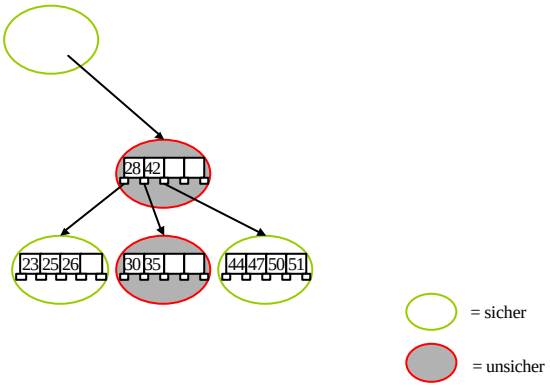


Abbildung 22 Sichere und unsicher Knoten bezüglich Löschoperationen.

6.4 Lock-Coupling

In [BAYERCONC] sind 3 verschiedene Protokolle zur Gewährleistung der Sicherheit bei parallelen Zugriffen gegeben. Jedes hat in bestimmten Situationen seine Vorteile. Wir werden hier nur die erste der Lösungen besprechen. Die anderen sind sehr ähnlich und beruhen auch auf dem Erkennen von sicheren Knoten.

Leser und Schreiber haben ein unterschiedliches Protokoll zu befolgen. Wie diese funktionieren, ist in folgendem Pseudocode gegeben:

```
Protokoll für Leser:
   0) Place S-lock on root;
   1) Get root and make it the current node;
   2) While current node is not a leaf do
            begin
                  3)Place S-lock on appropriate son of current node;
                  4)Release S-lock on current node;
                  5)Get son of current node and make it current;
            end mainloop

Protokoll für Schreiber:
   0) Place X-lock on root;
   1) Get root and make it the current node;
   2) While current node is not a leaf do
            begin
                  3)Place X-lock on appropriate son of current node;
                  4)Get son and make it the current node;
                  5)If current node is safe
                    then release all locks held on ancestors of
                    current node;
            end
```

[BAYERCONC]

Beim Lock-Coupling ist es nun so, dass sich auf einem Pfad von der Wurzel zum Zielknoten mit 2 Sperren „vorgehangelt“ wird.
Betrachten wir das Protokoll für den Schreiber:
Am Anfang wird gleich eine Sperre an die Wurzel gelegt. Nun wird der Kindknoten auf seine Sicherheit hin untersucht. Ist er sicher, wird er mit einer Sperre belegt und die Sperre von dem vorher gesperrten Knoten wieder entfernt. Wäre er nicht sicher gewesen, hätte sich der Prozess den nächsten Knoten auf dem Pfad angeschaut. Sobald es wieder einen sicheren Knoten auf dem Pfad zum Zielknoten gibt, kann der Prozess den vorher gesperrten Knoten wieder entsperren. Gibt es keinen sicheren Knoten mehr auf dem Pfad, bleibt die Sperre bis zum Ende der Schreib-Aktion erhalten und wird erst dann aufgehoben.

Ein Schreibprozess P der das Blatt D aus Abb. 22 verändern wird, würde also folgendermaßen vorgehen:
Zunächst würde er eine X-Sperre auf die Wurzel setzen. Eine Sperre kann natürlich immer nur dann gesetzt werden, wenn der Knoten nicht schon mit einer inkompatiblen Sperre belegt ist. Danach würde er eine Sperre auf den Knoten B beantragen. Sobald er diese genehmigt bekommen hat, untersucht er B auf seine Sicherheit. Da B sicher ist, kann er die Sperre von der Wurzel wieder entfernen, weil gewährleistet ist, dass sich seine Operation nicht mehr auf die Wurzel oder andere Teile des Baumes auswirken werden, die nicht Unterbäume des Knotens B sind (=abgeschlossener Bereich).
Durch das entfernen der Sperre von der Wurzel sind nun andere Prozesse wieder in der Lage, nicht abgeschlossene Bereiche des Knotens zu lesen oder zu ändern.
Jetzt beantragt der Prozess eine Sperre auf dem Knoten C. Nachdem er sie gesetzt hat, stellt er fest, dass C nicht sicher ist. Das bedeutet, er kann die Sperre auf Knoten B nicht lösen, weil er nicht weiß, ob sich B noch ändern wird. Wenn P eine Sperre auf D erhält und feststellt, dass D sicher ist, kann er die Sperren auf B und C wieder lösen. Während

der eigentlichen Operation blockiert der Prozess also wirklich bloß den Knoten, der geändert wird.

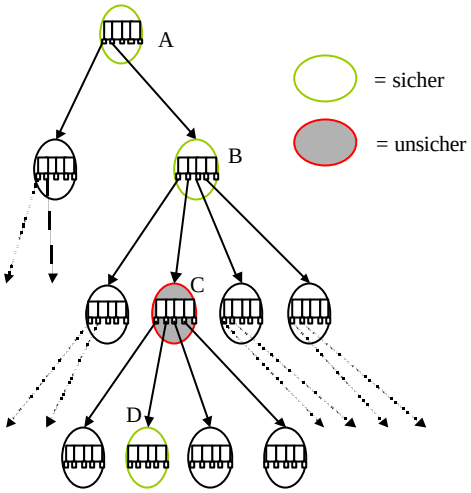


Abbildung 23

Dieser Ansatz ist schon einigermaßen brauchbar. Man kann aus diesem Algorithmus aber ein wesentlich höheres Maß an Parallelisierung herausholen, wenn man ihn noch ein wenig verändert und mehrere unterschiedliche Sperren verwendet. Außerdem muss man zwischen Lese- und Schreibprozessen unterscheiden.
Literatur [BAYERCONC]

7 Literaturverweise

[INFOHANDB]
Rechenberg/Pomberger, Informatik-Handbuch, Hanser-Verlag

[COMER]
Douglas Comer: The Ubiquitious B-Tree. ACM Computing Surveys 11(2):121-137 (1979)
[BAYERCONC]
Rudolf Bayer, Mario Schkolnick: Concurrency of Operations on B-Trees. Acta Informatica 9: 1-21 (1977)

[CMU]
Carnegie Mellon University
15-415 - Database Applications
Addendum on B-trees
http://www-2.cs.cmu.edu/afs/cs.cmu.edu/user/christos/www/courses/dbms.F00/HANDOUTS/btreeAlgos.txt