Wintersemester 2000/2001

Proseminar:

Grundlagen von Informationssystemen

- Indexstrukturen I : B-Bäume und ihre Varianten -

--- Christoph Tornau ---

--- ctornau@yahoo.de ----

Wichtige Begriffe (1)

- Tupel
  - Eine Ansammlung von Werten, die zu zueinander gehörigen Attributen gehören.
  - Entspricht in der tabellarischen Darstellung einer Zeile.
- Relation
  - Ansammlung von Tupeln gleichen Typs
  - Entspricht in tabellarischer Darstellung einer Tabelle.
- Datenstruktur
  - Die Datenstruktur ist die Art und Weise in der die Tupel einer Relation im Speicher abgespeichert werden. Die Datenstruktur bestimmt, wie gesucht wird und wie die Tupel abgegriffen werden. B-Bäume sind Datenstrukturen.
- Schlüssel
  - Das Attribut einer Relation, nach welchem eine Sortierung stattfindet
  - Wir gehen in diesem Vortrag davon aus, daß die Schlüssel diesmal nicht Unikate sind, sondern auch mehrere gleiche Schlüssel vorkommen können.

Wichtige Begriffe (2)

Wir betrachten in dem ersten Teil dieses Vortrages vorwiegend nur die Schlüssel der Tupel. Wir behalten im Hinterkopf, daß hinter den Schlüsseln auch noch Daten vorhanden sind:

Schlüssel

- Index
  - Sortierte Ansammlung von Schlüsseln und dazugehörige Zeiger, die auf
    - bestimmte Tupel in der Relation zeigen
    - oder auf bestimmte Indexeinträge in einem nachfolgenden Index zeigen

© 2000, 2001 Institut für Informatik III, Universität Bonn

Motivation und Grundlegendes (1)

- Suchvorgänge sind für die Datenbank essentiell wichtig. Fast bei jeder Anfrage wird auf der Datenbank gesucht.
- Es wird angestrebt ein **schnelles Suchen** auf einer Relation möglich zu machen.
- Was ist Suchen überhaupt?
  - Suchen nach einem bestimmten Schlüsselwert: Die Ausgabe sind die Tupel, deren Schlüsselwerte mit dem gesuchten Schlüsselwert übereinstimmen. Dies kann kein Tupel sein, dies kann ein Tupel sein und wenn es erlaubt ist, daß auch Schlüssel mehrmals vorkommen können, können es viele Tupel sein.
  - Suche nach einem Bereich: Hierbei muß ein bestimmter Bereich in der Relation eingegrenzt werden und ausgegeben werden. Ausgabe kann auch wieder kein Tupel sein oder mehrere Tupel.

© 2000, 2001 Institut für Informatik III, Universität Bonn

Motivation und Grundlegendes (2)

- Für eine effiziente Suche müssen wir die **Hardware** kennen:
  - Der größte Teil einer Relation kann nicht im Hauptspeicher sein. Wir müssen auf den Hintergrundspeicher zugreifen.
  - Hintergrundspeicherzugriffe dauern ewig. Deshalb minimieren.
  - Aus dem Hintergrundspeicher wird immer in Blöcken übertragen. Blockzugriffe sind zu minimieren.
  - Ein Tupel kann über mehrere Blöcke gehen. Es können auch mehrere Tupel in einem Block stehen.
- Folgende Funktionen müssen auf einer Datenstruktur implementiert sein:

Suchen **Einfügen** **Löschen**

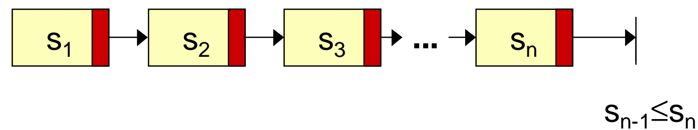
© 2000, 2001 Institut für Informatik III, Universität Bonn

Teil 1: Indexe

© 2000, 2001 Institut für Informatik III, Universität Bonn

Einfache Datenstrukturen: Liste

- Älteste und einfachste Methode
- Alle Datensätze sortieren
- In jedem Datensatz einen Zeiger auf den nachfolgenden Datensatz setzen

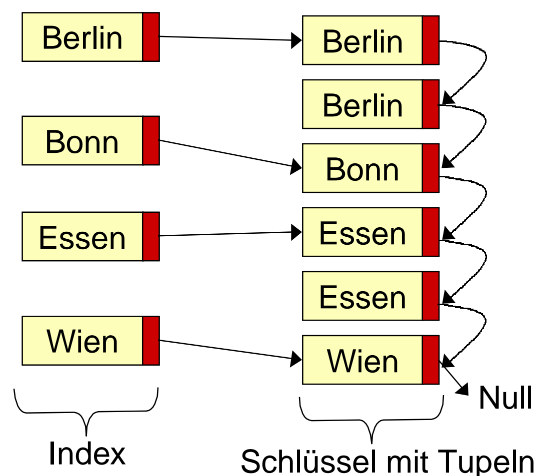


$s_{n-1} \leq s_n$

- **Suche**: Datensätze werden nacheinander betrachtet. Suche abbrechen, wenn der aktuelle Schlüssel größer ist als der gesuchte.
- **Einfügen**: Einfügen an der richtigen Position durch verbiegen der Zeiger
- **Löschen**: Zeiger vom Datensatz davor auf den nächsten Datensatz zeigen lassen.
- **Extrem lange Laufzeit**. Im schlimmsten Fall müssen sogar alle Datensätze aus dem Hintergrundspeicher geholt werden.

Dichter Index (Dense Indices) (1)

- Für jeden Wert, den ein vorhandener Schlüsseleintrag angenommen hat, gibt es einen Indexeintrag



Dichter Index (Dense Indices) (2)

- Suchen
  - Wir gehen am Index entlang.
    - Finden wir einen passenden Indexeintrag, springen wir zu dem entsprechenden Tupel.
    - Finden wir keinen, ist dieser Schlüssel nicht enthalten
- Einfügen
  - Ist der Schlüssel schon einmal vorhanden, fügen wir nach dem letzten Tupel mit diesem Schlüssel ein.
  - Ist der Schlüssel noch nicht vorhanden, fügen wir den Tupel an der richtigen Position (Sortierung) ein und fügen ebenso im Index an der richtigen Position einen Indexeintrag ein.
- Löschen
  - Wir suchen den richtigen Tupel. Dann entfernen wir ihn. Bei mehreren Tupeln muß man den richtigen entfernen
  - Wir entfernen den Indexeintrag, wenn es nur noch einen Tupel mit diesem Schlüssel gab

© 2000, 2001 Institut für Informatik III, Universität Bonn

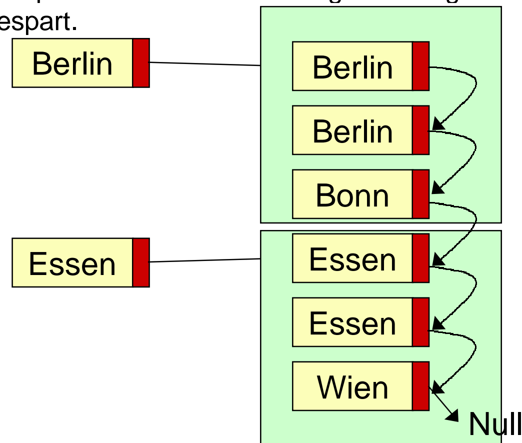
Dichter Index (Dense Indices) (3)

- Vorteile
  - Nur die Schlüssel müssen aus dem Hintergrundspeicher gelesen werden. Deshalb schneller als sequentielle Suche.
- Nachteile
  - Besonders bei kleinen Tupeln hinter dem Schlüssel und dementsprechend vielen Tupeln, wird der Index groß, so daß es lange dauert, in diesem zu suchen.
    - Beispiel: Eine Relation für eine Klausur in der die Attribute „Martrikelnummer“ und „bestanden“ gespeichert sind.
  - Der Index belegt viel Speicherplatz

© 2000, 2001 Institut für Informatik III, Universität Bonn

Gestreuter Index (Sparse Indices) (1)

- Nicht jedes Tupel hat einen Indexeintrag. Bei einigen Tupeln wird dieser eingespart.



- Besonders günstig ist diese Strategie, wenn die Tupel die hinter einem Indexeintrag stehen, in einem Block von der Festplatte kommen. Nur Blockzugriffe sind zeitfressend.

© 2000, 2001 Institut für Informatik III, Universität Bonn

Gestreuter Index (Sparse Indices) (2)

- Suchen
  - Wir suchen, indem wir den Indexeintrag suchen, der kleiner oder gleich dem Schlüssel ist. An dieser Position springen wir in die Relation und suchen dort weiter.
- Einfügen
  - Wir fügen den Tupel an der richtigen Stelle in der Relation ein.
  - Es ist abzuwägen, ob wir einen Indexeintrag einfügen.
    - ➡ Dies hängt von unseren Vorgaben ab, wie oft wir einen Indexeintrag haben wollen.
- Löschen
  - Wir löschen das Tupel.
  - Wenn wir einen Indexeintrag auf diesen Tupel zeigend hatten, löschen wir ihn auch oder schreiben in diesen das nächste Tupel hinein.
  - Es ist wieder abzuwägen, ob wir vielleicht einen anderen Indexeintrag wieder löschen und den Index gestreuter machen.
    - ➡ Dies hängt wieder von unseren Vorgaben ab, wie oft wir einen Indexeintrag haben wollen.

© 2000, 2001 Institut für Informatik III, Universität Bonn

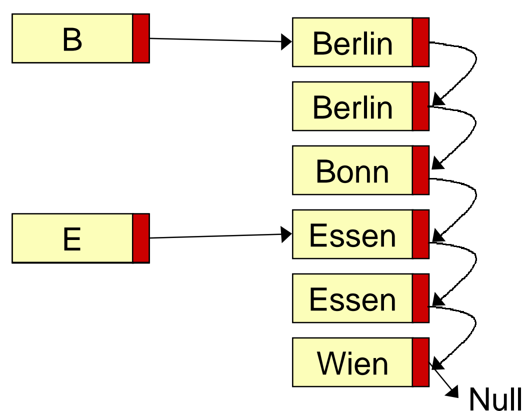
Gestreuter Index (Sparse Indices) (3)

- Vorteile
  - Der Index ist kleiner
- Nachteile
  - Je kleiner („gestreuter“) der Index ist, desto mehr Datensätze müssen wieder durchlaufen werden.
  - Speicherplatzersparnis geht auf Performance

© 2000, 2001 Institut für Informatik III, Universität Bonn

Präfixindex (1)

- Beim Präfixindex steht nur ein Teil des Schlüssels im Index. Dadurch wird der Index wieder um einiges kleiner



© 2000, 2001 Institut für Informatik III, Universität Bonn

Präfixindex (2)

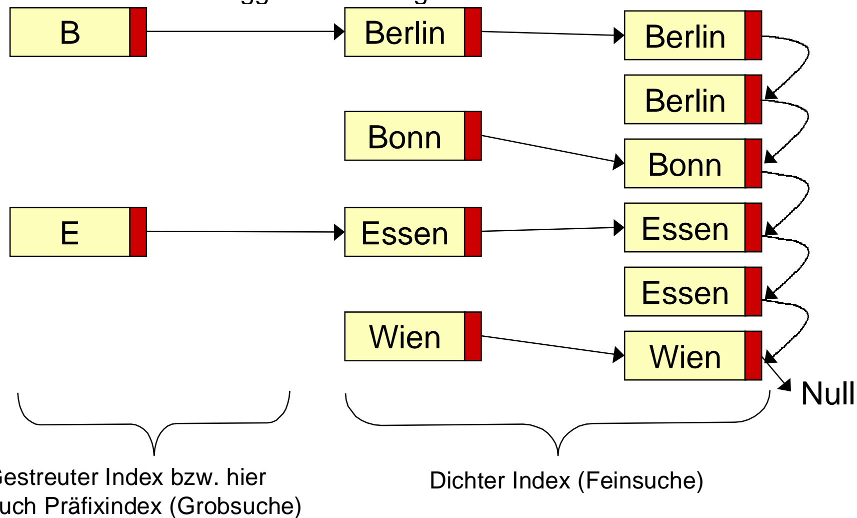
- Suchen, Einfügen und Löschen funktionieren ähnlich wie beim Gestreuten Index.

- Vorteile:
  - kleinere Indexe
  - Weil die Indexe kleiner sind kann man sich wieder einen längeren Index erlauben.
- Nachteile:
  - Es sind eigentlich keine wesentlichen Verbesserungen bezüglich der beiden anderen Formen gemacht worden

© 2000, 2001 Institut für Informatik III, Universität Bonn

Multilevel Index (1)

- Abhilfe verschafft der Multilevelindex. Hier wird der Index auf mehreren Ebenen geschaltet. Der erste Index realisiert die Grobsuche, der zweite die Feinsuche ggf. auf beliebig vielen Ebenen.



© 2000, 2001 Institut für Informatik III, Universität Bonn

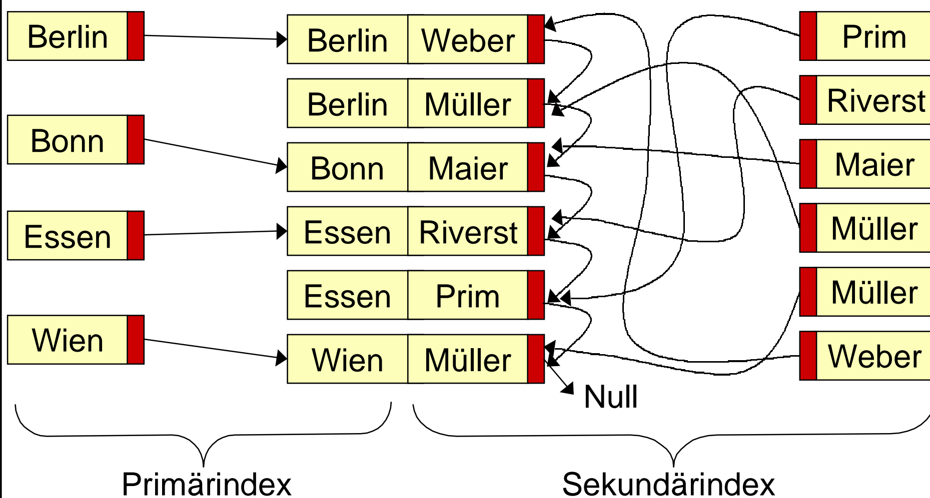
Multilevel Index (2)

- Suchen, Einfügen und Löschen gehen ähnlich wie bei normalen Indexen, nur muß hier auf mehren Ebenen gearbeitet werden.

- Vorteile
  - Aufgrund der mehreren Ebenen lassen sich die richtigen Tupel schnell finden.
- Nachteile
  - Mehr Speicher wird verbraucht.

Relation nach mehreren Schlüsseln sortiert (1)

- Bis jetzt haben wir die Tupel nur nach einem Schlüssel sortiert. Doch wir können auch nach mehreren Schlüsseln sortieren:



Relation nach mehreren Schlüsseln sortiert (2)

- Bei dieser Indexart treten massive Probleme mit den Zeigern in einem Index auf, wenn wir über den anderen Index Hinzufügen und Löschen
  - Es ist zu bedenken, daß wir auch im anderen Index die Zeiger ändern müssen.
  - Wenn wir das tun wollen müssen wir Zeiger haben, die aus der Relation wieder zurück auf den entsprechenden Indexeintrag zeigen, was Speicherplatz frißt oder wir müssen entsprechend Suchzeit in Kauf nehmen.

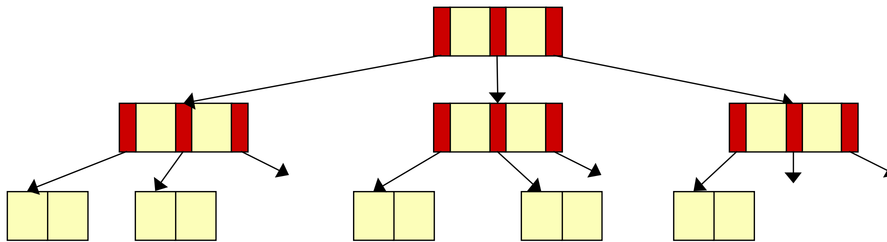
© 2000, 2001 Institut für Informatik III, Universität Bonn

Teil 2: B-Bäume

© 2000, 2001 Institut für Informatik III, Universität Bonn

B-Bäume Aufbau (1)

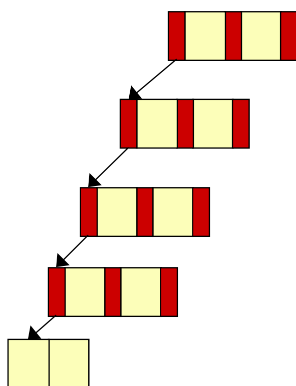
- B-Bäume sind keine Binären Bäume. B steht für balanced, d.h. ausbalanciert. Es ist erwünscht, daß ein B-Baum in jedem Blatt die gleiche Tiefe hat.



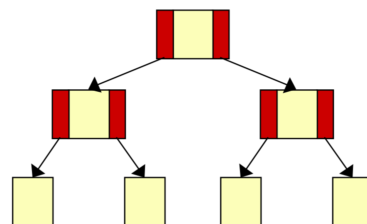
© 2000, 2001 Institut für Informatik III, Universität Bonn

B-Bäume Aufbau (2)

Gegenbeispiele:



hochgradig nicht ausbalancierter Baum



Binärer Baum (B-Baum, wenn er in jedem Blatt die gleiche Tiefe hat)

© 2000, 2001 Institut für Informatik III, Universität Bonn

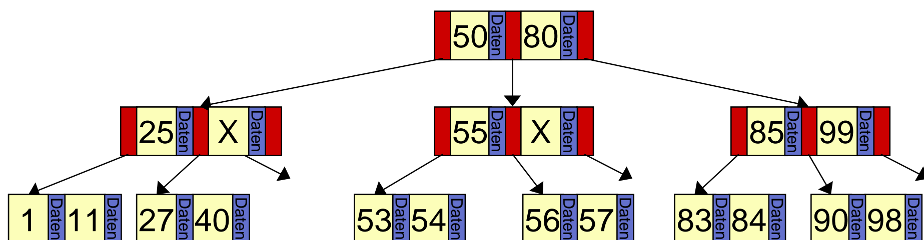
B-Bäume Aufbau (3)

- Knoten
  - In den Knoten sind Schlüssel gespeichert. Vor und nach jedem Schlüssel gibt es einen Zeiger, der auf einen Sohn zeigen kann.
  - Ist ein Schlüssel
    - kleiner als der Schlüssel im Knoten, so können wir ihn finden, wenn wir dem linken Zeiger folgen.
    - größer als der Schlüssel im Knoten, so können wir ihn finden, wenn wir dem rechten Zeiger folgen.
  - Beim B-Baum werden an den Schlüssel auch die zu dem Schlüssel gehörenden Daten gespeichert. Finden wir also den passenden Schlüssel, so können wir direkt auf die Daten zugreifen.
  - In einem Knoten können beliebig viele Schlüssel enthalten sein. Sind zwei Schlüssel enthalten, so ist es ein binärer Baum. Aber auch z.B. 50 Schlüssel sind durchaus in der Anwendung.
  - Ein Knoten muß immer zur Hälfte bis ganz gefüllt sein. Dies ist Vorgabe für den B-Baum, da ansonsten Speicherplatz verschwendet wird.
- Blätter
  - haben keine Zeiger auf andere Knoten mehr.

© 2000, 2001 Institut für Informatik III, Universität Bonn

B-Bäume Aufbau (4)

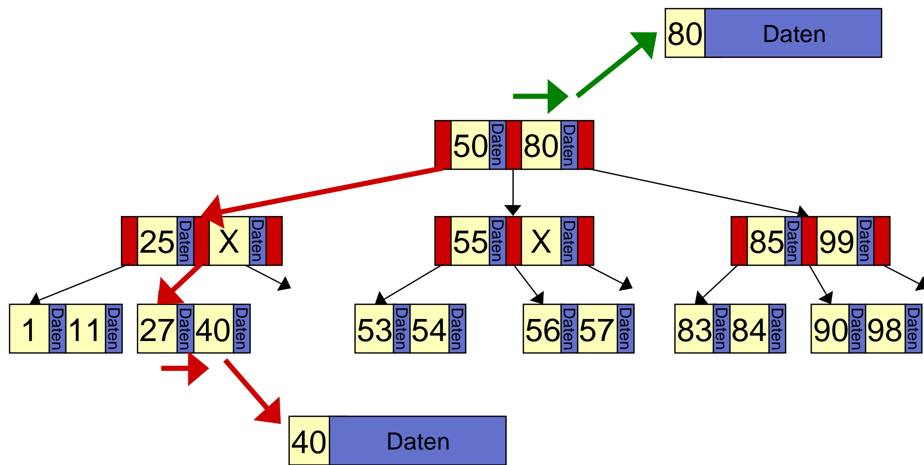
Beispiel:



© 2000, 2001 Institut für Informatik III, Universität Bonn

B-Bäume Suche

Beispiel:



© 2000, 2001 Institut für Informatik III, Universität Bonn

B-Bäume Einfügen und Löschen

- Einfügen
  - Wir fügen ein, indem wir den Schlüssel suchen, der kleiner als der einzufügende Schlüssel ist und am nächsten dem Schlüsselwert des einzufügenden Schlüssels liegt.
  - Wenn wir keine Position finden können, müssen wir uns eine schaffen, indem wir einen Knoten splitten und in zwei Knoten aufteilen.
- Löschen
  - Wir löschen, indem wir den Schlüssel suchen. Dann entfernen wir ihn.
  - Es kann passieren, daß gerade der Datensatz gelöscht wird, der in einem Knoten steht, welches kein Blatt ist. Hier sind große Umkopiervorgänge im Baum von nöten.
  - Es kann sein, daß der entsprechende Knoten zu klein wird.
    - ➡ Wir können Tupel vom Nachbarn ausleihen.
    - ➡ Ist der Nachbar jedoch auch zu klein müssen wir ihn mit einem Knoten aus dem benachbarten Zweig verschmelzen lassen. Dabei müssen viele Datensätze im Baum umkopiert werden, um die Richtigkeit des Suchpfades zu gewährleisten.

© 2000, 2001 Institut für Informatik III, Universität Bonn

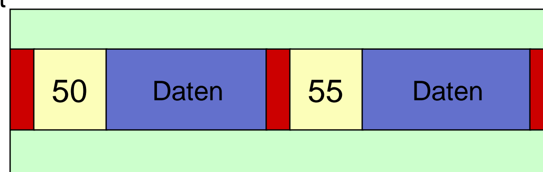
B-Bäume Vorteile und Nachteile

- Vorteile
  - Es gibt keinen Overhead durch den Index mehr. Nur noch einige kleinere Pointer müssen verwaltet werden.
  - Wir haben eine schnelle Suche realisiert. Es muß kein langer Index oder gar eine lange Relation mehr durchgegangen werden, sondern man kann in einem Baum den richtigen Weg finden.
  - Man kann schon früh auf den richtigen Tupel treffen, da schon in der Wurzel welche enthalten sind. Dies ist jedoch eher unwahrscheinlich.

- Nachteile
  - Das eben angesprochene Problem, daß wenn man ein Tupel löscht, welches nicht in einem Blatt steht, man viel umkopieren muß.
  - Das sequentielle Auslesen ist nur schlecht möglich.

Der Vorteil von Bäumen beim Lesen in Blöcken

- Aus dem Hintergrundspeicher, von der Festplatte, wird immer in Blöcken gelesen. Das Holen der Blöcke beansprucht sehr viel Zeit. Die Bearbeitung im Hauptspeicher allerdings wenig.
- B-Bäume nutzen dies, indem ein Knoten möglichst ein Block auf der Festplatte ist



- Es können auch mehrere Knoten in einem Block sein. Hierbei empfiehlt es sich, Sohnknoten teilweise oder ganz mit in einen Block zu nehmen, da diese wahrscheinlich als nächste aufgerufen werden. Ebenso ist es möglich, einen Knoten auf mehrere Blöcke zu verteilen.
- Bei dynamischer Länge von Tupeln kommt man in Schwierigkeiten die Blöcke gut auszunutzen, denn auch nach dem Aufbau wird gelöscht und eingefügt. (Fragmentierung setzt ein)

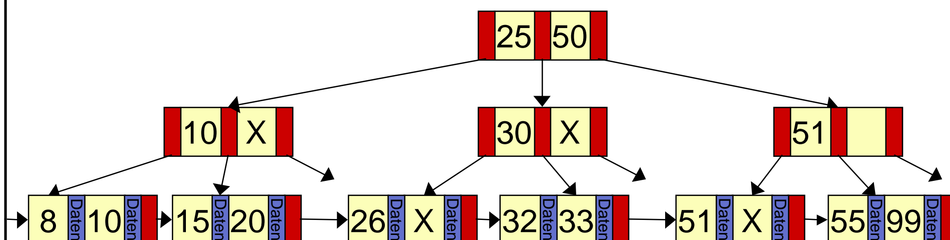
Virtueller Speicher

- Man kann Knoten in den virtuellen Speicher auslagern. Man lagert geschickt, so daß ein Knoten möglichst beim Beginn einer virtuellen Speicherseite liegt.
- Viele Rechner verfügen über eine spezielle Hardwaremethode um den virtuellen Speicher schnell zu schreiben und zu lesen. Dies können wir mit diesem Trick auch für die B-Bäume benutzen.

© 2000, 2001 Institut für Informatik III, Universität Bonn

B^+-Bäume Aufbau

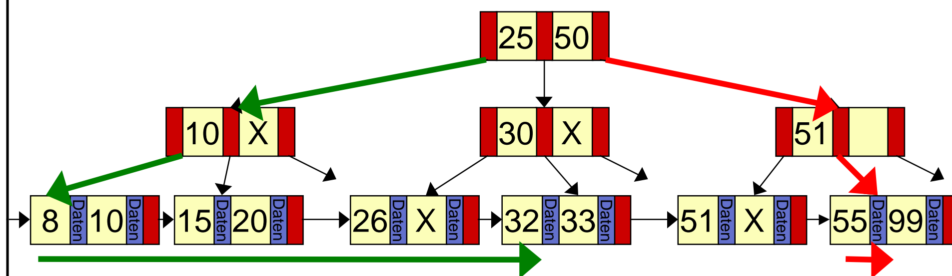
- Abhilfe zu dem Löschproblem und dem Speicherproblem schaffen die B^+-Bäume
- Bei B^+-Bäumen werden die eigentlichen Daten nur in den Blättern gespeichert. Alle anderen Knoten enthalten Schlüssel. Die Blätter werden untereinander verkettet.



© 2000, 2001 Institut für Informatik III, Universität Bonn

B+-Bäume Suchen

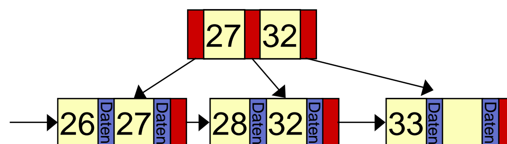
- Suchen wir nach 99 durchlaufen wir den B⁺-Baum indem wir den Zeiger benutzen, dessen zugehöriger Schlüsselwert kleiner oder gleich dem zu suchenden Schlüsselwert ist und dessen Nachfolger größer als der zu suchenden Schlüsselwert ist. Wir finden dann im Blatt den gewünschten Schlüssel oder auch nicht.
- Suchen wir nach allen Schlüsseln zwischen 8 und 32 durchlaufen wir den B⁺-Baum nach dem ersten. Dann können wir die sequentielle Verknüpfung benutzen. Dies ist ein großer Vorteil des B⁺-Baumes.



© 2000, 2001 Institut für Informatik III, Universität Bonn

B+-Bäume Einfügen (1)

- Wir fügen ein, indem wir die richtige Position finden. Wenn wir eine freie passende Position in einem Blatt gefunden haben: Einfügen
- Gibt es keine freie Position mehr, dann schauen wir, ob im Vaterknoten noch Platz frei ist und erzeugen einen neuen Knoten. Wir verteilen in diesem Fall die Schlüssel neu



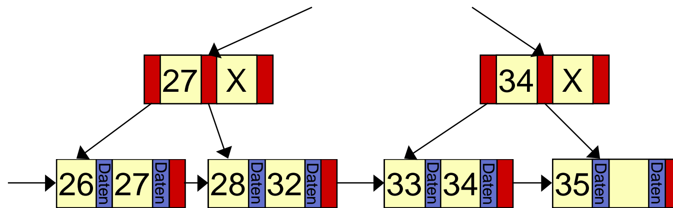
Fügen wir 27 ein

Fügen wir 28 ein

© 2000, 2001 Institut für Informatik III, Universität Bonn

B^{+}-Bäume Einfügen (2)

- Gibt es im Vaterknoten keinen Platz mehr um einen weiteren Knoten anzuhängen, müssen wir den Knoten splitten



Fügen wir 34 ein

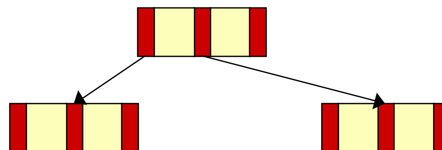
Fügen wir 35 ein

- Gibt es im Knoten über den Vaterknoten einen weiteren freien Platz, so wird dieser benutzt. Wenn es keinen weiteren Platz gibt, so wird das Verfahren rekursiv angewand. Es kann sein, daß dies bis zu einer Wurzelsplittung führt.

© 2000, 2001 Institut für Informatik III, Universität Bonn

B^{+}-Bäume Einfügen (3)

- Wurzelsplittung

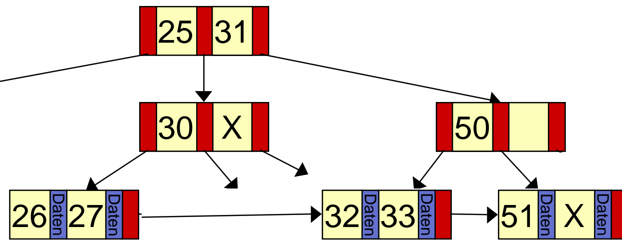


- Die Baumtiefe hat sich um 1 erhöht.

© 2000, 2001 Institut für Informatik III, Universität Bonn

B$^+$-Bäume Löschen (1)

- Wir suchen das zu löschende Tupel und entfernen es
- Haben wir nach dem gelöschten Tupel einen Vater, der auf zuwenige Schlüssel mit Daten in Blättern zeigt, müssen wir innerhalb derselben Ebene Schlüssel austauschen. Wir borgen ein Blatt vom Nachbarn und passen dementsprechend auch den Suchpfad an.



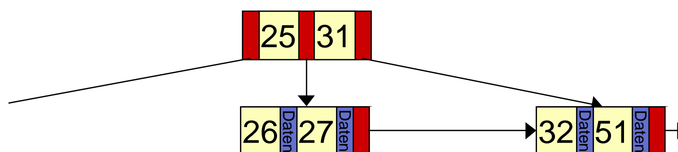
Löschen wir 99

Löschen wir 55

© 2000, 2001 Institut für Informatik III, Universität Bonn

B$^+$-Bäume Löschen (2)

- Es kann sein, daß es keinen Knoten gibt, von dem wir ein Blatt ausborgen können. In diesem Falle verringern wir die Baumhöhe um 1, indem wir die Schlüssel mit den Datensätzen in den Vater umschreiben.



Wir löschen 33

© 2000, 2001 Institut für Informatik III, Universität Bonn

B^+-Bäume Vorteile und Nachteile

- Vorteile
  - Die einzelnen Tupel sind auch sequentiell abgreifbar.
  - Das Löschproblem entfällt.
    - ➡ Es kann nicht mehr vorkommen, daß wir ein Tupel löschen möchten und wir den gesamten Baum verändern müssen, da es gerade in der Wurzel steht.
  - Das Problem mit der variablen Größe der Tupel läßt sich einfacher lösen.
    - ➡ Es können Blätter variabler Länge gewählt werden. In den Blocks, in denen diese Blätter enthalten sind, kann einfacher defragmentiert werden.
  - Da in den Suchknoten nur die Schlüssel gespeichert werden, kann man in einem Block einen relativ großen Knoten unterbringen.
    - ➡ Es gibt Knoten mit über 50 Schlüsseln.
    - ➡ Die Suche beschleunigt sich enorm, da man mit wenigen Blockzugriffen am Ziel ist.
- Nachteile:
  - Wir haben einiges an Overhead. Es ist der Speicher, der durch die Suchknoten belegt ist.

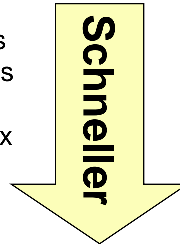
B^+-Bäume Nochmalige Optimierung

- Man kann in den Knoten weiter oben im Baum nur den Präfix des Schlüssels listen.
  - Da dort die einzelnen Schlüssel noch sehr weit von einander entfernt liegen, kann man dies sehr gut anwenden.
  - So kann man noch mehr Schlüssel in einem Knoten unterbringen, welcher in einen Block paßt und auf dem Baum noch schneller suchen.

Zusammenfassung

- Es gibt Datenstrukturen, die eine Suche auf einer Relation erheblich schneller machen.

Liste
Dense Indices
Sparse Indices
Präfixindex
Multilevelindex
B-Baum
B^{+}-Baum



- Durch geschickte Ausnutzung der Hardware (Blockzugriffe, virtueller Speicher) kann die Relation noch schneller durchsucht werden, da der Hintergrundspeicher effizient und schnell zu statten geht.

© 2000, 2001 Institut für Informatik III, Universität Bonn