GRUNDLAGEN VON INFORMATIONSSYSTEMEN
INDEXSTRUKTUREN I: B-BÄUME UND IHRE VARIANTEN

Leiter des Proseminars: Dr.Thomas Bode
Verfasser dieses Vortrages: Christoph Tornau
Datum: 20.11.2000

Inhalt:

1. Einführung

In diesem Essay geht es um die Art und Weise, in der Daten im Hintergrundspeicher abgelegt werden. Da es bei großen Datenbanken nicht möglich ist, alle Daten zufällig abgreifbar im Hauptspeicher zu halten, müssen wir den Hintergrundspeicher benutzen. Aus dem Hintergrundspeicher können wir nur in Blöcken übertragen. Wir müssen also später darauf achten, daß nur wenige „geschickte" Blockübertragungen aus dem Hintergrundspeicher stattfinden, um eine möglichst schnelle Implementierung für die Grundfunktionen einer Datenstruktur zu erhalten. Es geht dabei im Hauptsächlichen um B^+-Bäume. Wir werden sehen, warum gerade diese Speicherstruktur benutzt wird, um eine schnelle **Suche** zu realisieren. Auf einer Datenstruktur müssen weiterhin, das sei zu Beginn noch gesagt, noch die wichtigen Funktionen **Einfügen** und **Löschen** implementiert sein.

Zu Beginn müssen wir uns auch noch klarmachen, was eine Suche auf einer Datenstrukutur überhaupt ist. Zum einen können wir einen Schlüsselwert suchen. Wenn wir eine Datenstruktur haben, wo immer nur ein Schlüssel einmal vorkommt, so bekommen wir hier auch einen oder keinen Schlüssel zurück. Aber das Problem beginnt schon, wenn wir Datenstrukturen haben, wo der zu suchende Schlüssel mehrfach vorkommt oder vorkommen kann. Wir müssen dann bei der Suche alle Schlüssel zurückgeben und nicht nur einen. Dann können wir auch noch Bereichsabfragen starten, z.B. alle Datensätze die einen Schlüssel < 100 haben. Für Bereichsabfragen sind einige Datenstrukturen besonders schlecht geeignet.

Wichtige Begriffe in dieser Ausarbeitung sind folgende:

Tupel: Ein Tupel ist eine Ansammlung von Werten zu Attributen, die zu zueinander gehören. Ein Tupel entspricht in der tabellarischen Darstellung einer Relation einer Zeile der Tabelle.

In diesem Essay wollen wir die Daten, die in einem Tupel gespeichert sind, zunächst nicht betrachten. Ein Datensatz hat für uns die folgende Form:

Schlüssel: Ein Schlüssel ist das Attribut, nach welchem das Tupel gekennzeichnet wird und nach welchem die Relation sortiert wird. Meistens sind solche Schlüsselattribute Unikate, d.h. ein Schlüsselattributwert kann nur einmal in der gesamten Datenbank vorkommen. Wir betrachten jedoch hier zunächst auch Relationen, in denen Schlüsselwerte doppelt vorkommen.

Datenstruktur: Die Datenstruktur ist die Art und Weise, in welcher Datensätze abgespeichert werden.

Relation: Eine Menge von Tupeln gleichen Typs. Eine Relation kann man in tabellarischer Darstellung darstellen, in dem man alle Tupel mit den dazugehörigen Attributen auflistet.

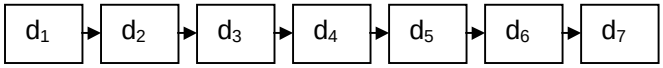
Index: Ein Index ist eine Ansammlung von Schlüsseln und Zeigern, die auf bestimmte Datensätze zeigen, um die Suche zu beschleunigen.

2. Einfache Datenstrukturen

2.1 Sequentielle Struktur

Die Tupel können in einer sortierten Liste abgelegt werden. Bei dieser Liste wird jedoch maximale Zeit verbraucht um Tupel zu suchen. Wenn der zu suchende Schlüssel nicht vorkommt, können wir die Suche unterbrechen, wenn der Schlüssel des aktuellen Tupels größer als der zu suchende Schlüssel ist. Andernfalls kann es sein, daß sogar alle Tupel durchgegangen werden müssen und aus dem Hintergrundspeicher geladen werden müssen, bis der richtige Tupel gefunden worden ist.

Beispiel für diese Datenstruktur:



Wobei $d_1 ... d_7$ die Schlüssel der Tupel darstellen. Es gilt $d_1 < d_2 < ... < d_7$

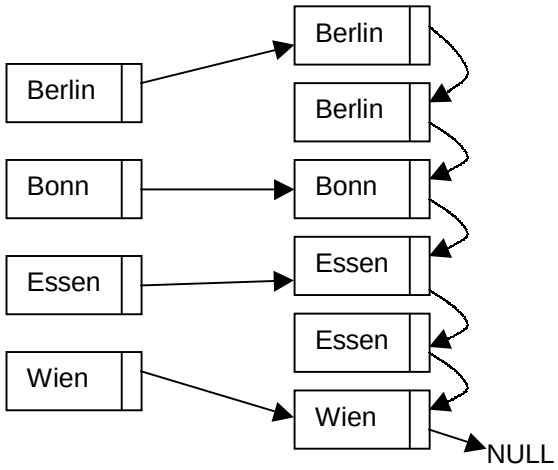
Bei großen Datenbanken ist es so, daß nicht mehr die ganze Liste im Hauptspeicher gehalten werden kann. Es kann sein, daß je ein Tupel in einem Block auf der Festplatte gespeichert wird. Wird es dies, so sind maximal Anzahl der Tupel Festplattenzugriffe nötig. Viel wahrscheinlicher ist es jedoch, daß mehrere Tupel in einem Block liegen. Zunächst sieht es aus, als ob wir dann weniger Blockzugriffe benötigen. Doch wenn diese Tupel nicht hintereinander in der Liste liegen, muß der Block eventuell für jeden gelesenen Tupel neu von der Festplatte geladen werden. Eine solche Struktur kann vor allen Dingen dann aufgebaut werden, wenn auf der Datenbank geschrieben und gelöscht wird. Ein Defragmentieren würde wieder viel Zeit benötigen.

2.2 Dichter Index (Dense Index)

Im Dichten Index gibt es für jeden Wert, den ein vorhandenes Schlüsselattribut angenommen hat einen Indexeintrag. Dieser Indexeintrag zeigt auf den ersten Schlüssel in der Datenbank, der den Wert einnimmt. Will man einen Tupel mit einem Schlüssel, der noch nicht vorhanden ist, einfügen, muß man einen neuen Indexeintrag erzeugen. Wenn man einen Tupel wieder löschen möchte, dessen Schlüssel nur einmal in der Relation vorkommt, muß man den Indexeintrag für diesen Tupel auch löschen, ansonsten tut man es nicht.

Es ist darauf zu achten, daß wenn es viele Tupel gibt, dieser dichte Index extrem groß werden kann. Besonders bei vielen kleinen Tupel lohnt dieses Verfahren nicht, da der Index äußerst groß wird.

Beispiel für einen Dichten Index:

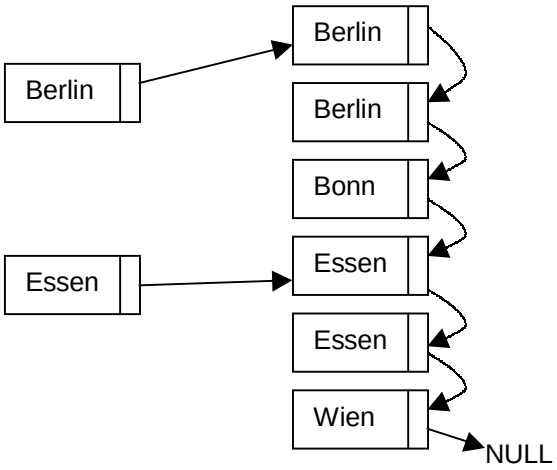


2.3 Gestreuter Index (Sparse Indices)

Im gestreuten Index gibt es weniger Indexeinträge. Nicht mehr jeder Schlüssel kommt im Index vor. Der Index wird dafür benutzt um an einer passenden Stelle in die Liste hineinzuspringen, um dann ab der Stelle den Datensatz zu finden. Man kann an der Größe des Indexes so sparen. Je gestreuter und dünner jedoch der Index wird, desto kostenaufwendiger ist die Suche, da immer mehr Datensätze durchlaufen werden müssen.

Wenn man Datensätze in die Relation mit gestreutem Index einfügen möchte, und deren Schlüssel noch nicht im Index vorhanden ist, so ist abzuwägen, ob dieser in den Index aufgenommen werden soll. Dies muß anhand spezieller Algorithmen in der Einfügemethode geschehen. (Beispielsweise Index einfügen, wenn es seit 3 Datensätzen keinen Index mehr gab) Löschen aus der Relation mit gestreutem Index kann man, indem man einfach wieder den Datensatz löscht. Wenn es einen Indexeintrag auf diesen Datensatz gab, dann wird dieser auch wieder gelöscht. Man kann weiterhin nun wieder neue Indexeinträge erzeugen, damit der Index so gestreut bleibt, wie man es vorher definiert hat. (Wieder das Beispiel, daß nach je 3 Datensätzen ein Datensatz mit Indexeintrag kommt und dies beim Löschen auch beibehalten wird.)

Beispiel:

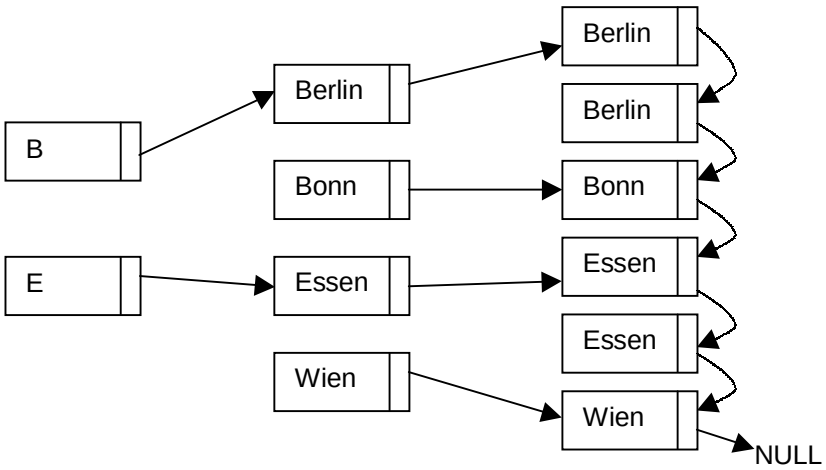


Die Vorteile einer solchen Indexstruktur sind, daß Speicherplatz für den Index eingespart wird. Dieser kann dann schneller durchlaufen werden. Ein weiterer Vorteil ist, daß man jeden Indexeintrag jeweils auf einen Block auf der Festplatte zeigen kann. Das lesen dieses einen Blocks geht langsam von statten, das durchsuchen dieses Blockes allerdings schnell. Zunichte gemacht wird dieser Vorteil wieder, wenn die Tupel, die nacheinander folgen zerstreut in den Blocks liegen. Hier muß immer wieder neu geladen werden. Vielleicht ist sogar dann ein dichter Index besser.

2.4 Multilevel Index

Um das Problem der langen Laufzeiten zu lösen, kann man mehrere Indexe verwenden. Man kann anhand eines ersten Indexes suchen und anhandes des Indexeintrages auf einen zweiten Index springen usw. bis man den richtigen Datensatz gefunden hat. Beim Einfügen und Löschen in eine solche Datenstruktur muß man natürlich auch die Indexeinträge in allen Indexen der nun hergestellten Relation entsprechend verändern.

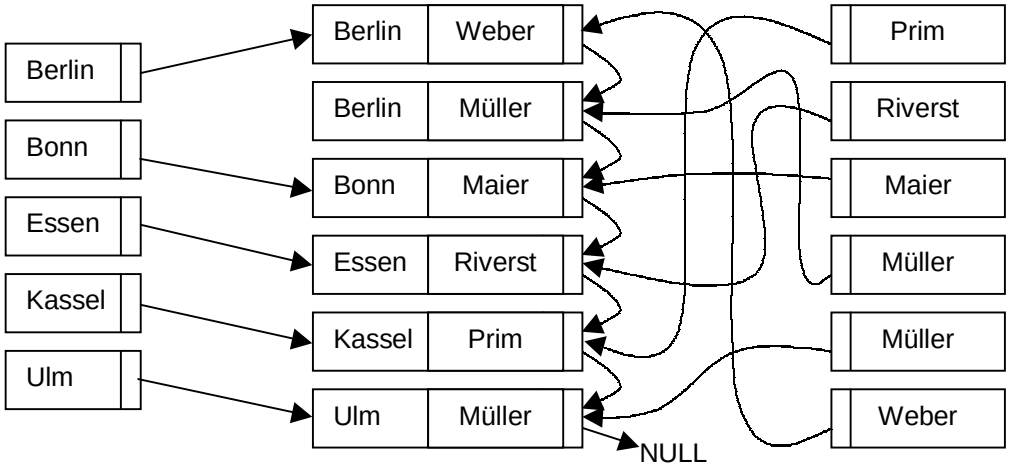
Beispiel für einen Multilevelindex:



2.5 Index nach mehreren Schlüsseln sortiert

Es gibt des weiteren noch Relationen, in denen mehrere Schlüssel vorliegen, nach denen sortiert werden kann. Man kann nun mehrere dichte Indexe anlegen und jeden Index nach einem Schlüssel sortiert haben, um auf diese Relation schnell zugreifen zu können. Beim Einfügen und Löschen müssen dann die entsprechenden Veränderungen an allen Indexen durchgeführt werden.

Beispiel:

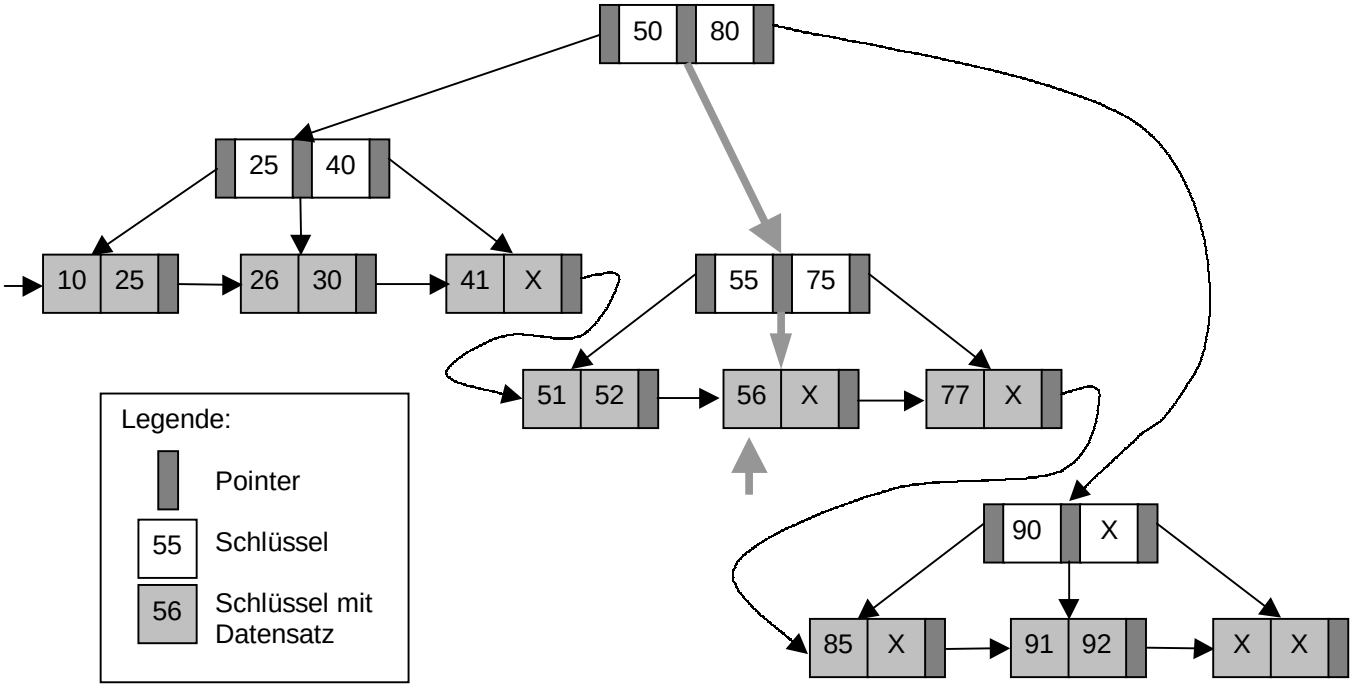


3. B$^+$-Bäume

3.1 Aufbau

Der B$^+$-Baum ist wie der Name schon sagt, ein Baum. Er besteht aus Knoten in denen mehrere Schlüssel enthalten sind. Durch die Schlüssel in den Knoten kann man den „Weg“ zu dem gewünschten Datensatz finden, indem man die Zeiger benutzt, die ebenfalls in den Knoten enthalten sind. Die Blätter, in denen auch mehrere Datensätze vorhanden sein können, sind untereinander sequentiell verknüpft, damit man schneller alle Datensätze zusammen auslesen kann. Das B in B$^+$-Bäumen steht für balanced. Der B$^+$-Baum ist ausbalanciert. Das heißt, er hat bis zu jedem Blatt die gleiche Länge.

Beispiel:



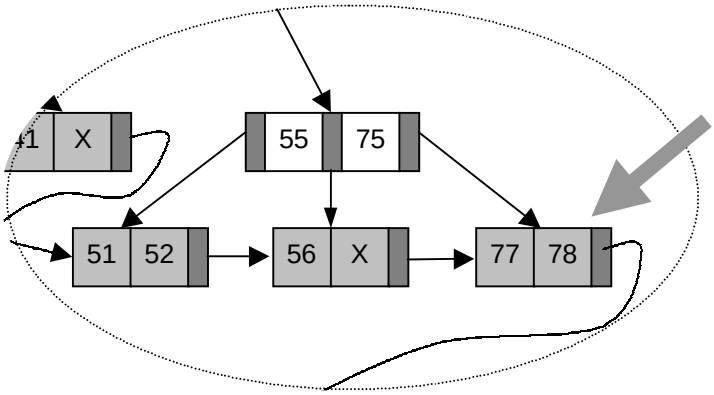
3.2 Suchen

Gesucht wird in einem B^+-Baum, indem man in jedem Knoten nach dem zu suchenden Schlüssel sucht. Ist der Schlüssel größer als ein im Knoten links liegender Schlüssel und kleiner als in einem Knoten rechts liegender Schlüssel, so folgt man dem Pointer zwischen den beiden Schlüsseln. Im obigen Baum ist der Suchweg 56 markiert.

Warum der B^+-Baum so effizient bei der Suche ist, liegt daran, daß ein Knoten möglichst in einem Block liegt. Manchmal liegen über 50 Schlüssel in einem Knoten. Zwar muß auf den Knoten einzeln noch eine sequentielle Suche stattfinden, aber da dies eine Suche im Speicher ist, ist sie sehr schnell.

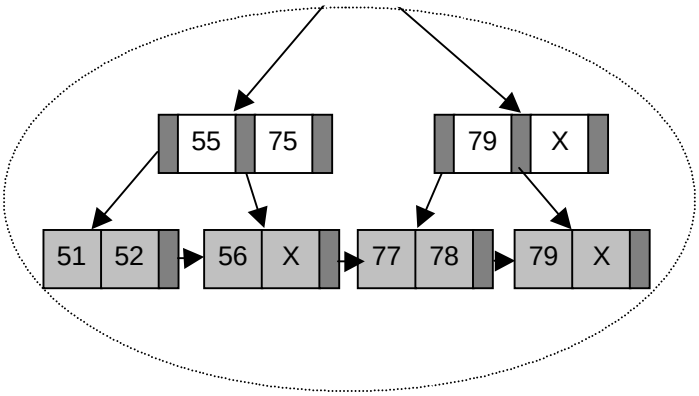
3.3 Hinzufügen

Beim einfachen Hinzufügen wird gemäß dem Suchalgorithmus das Blatt im Baum gesucht, an der der Datensatz eingefügt werden muß. Dann wird er dort hingeschrieben. Fügen wir beispielsweise 78 ein, dann passiert folgendes:



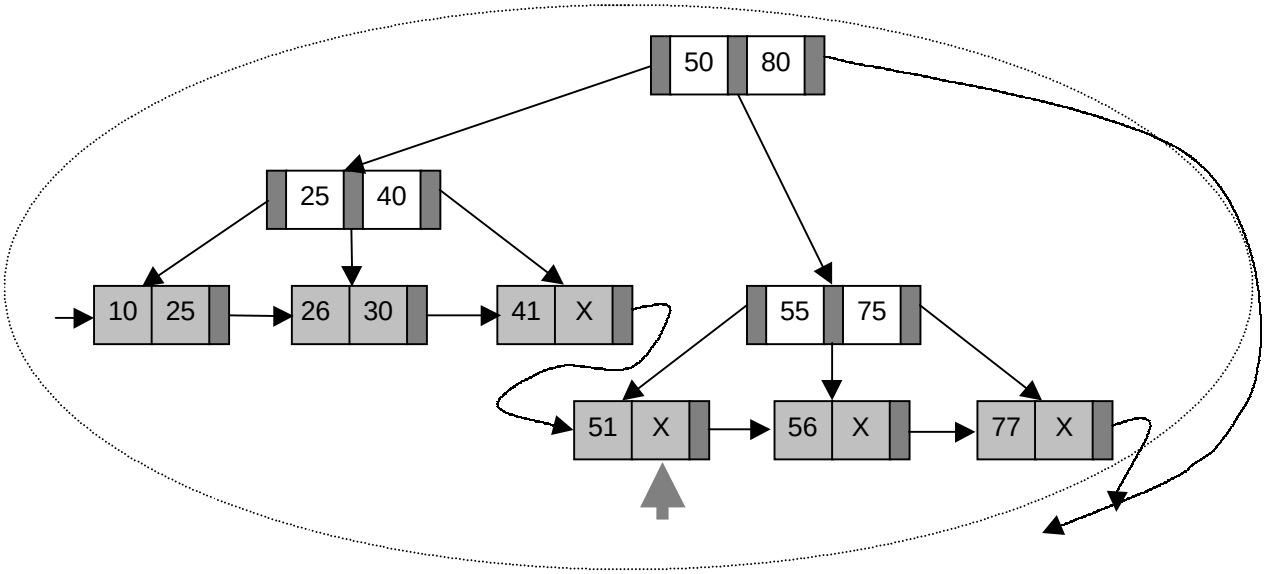
Wenn jedoch für den einzufügenden Datensatz in dem jeweiligen Blatt keinen Platz mehr haben, dann müssen wir diesen schaffen, indem wir in dem darüberliegenden Knoten einen neues Blatt einfügen. Ist dies auch nicht möglich, dann spalten wir diesen Knoten und gehen rekursiv wieder einen Knoten rückwärts und versuchen dort die beiden gespalteten Knoten einzufügen. Dies können wir bis dahin tun, daß die Wurzel gespalten wird, der Baum eine neue Wurzel bekommt und dementsprechend um 1 länger wird.

Beispielsweise fügen wir 79 ein:



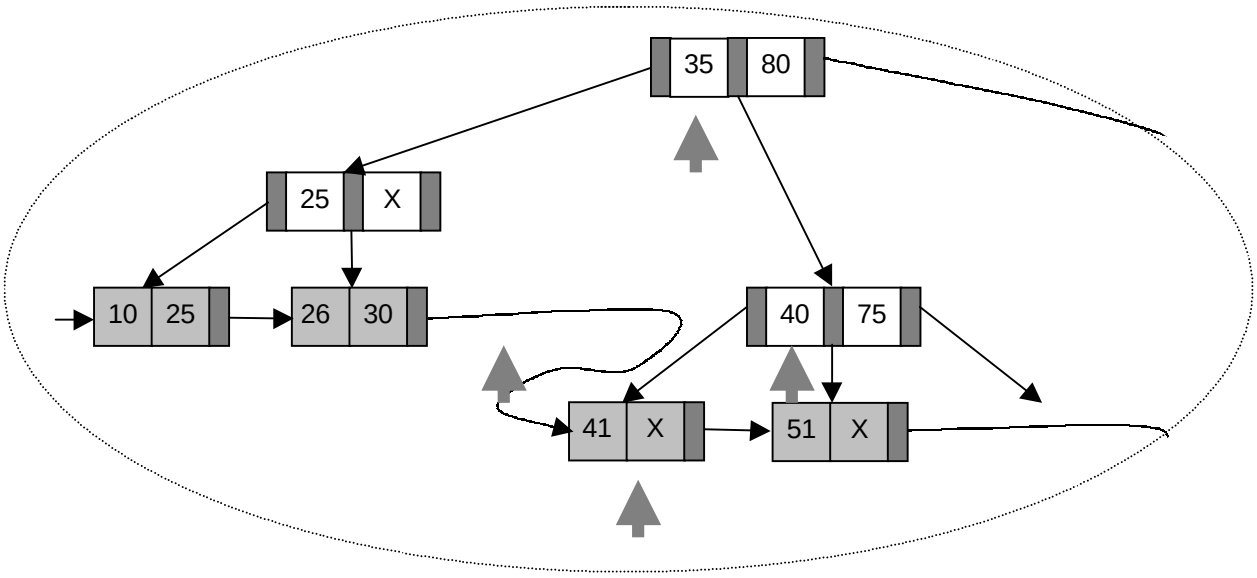
3.4 Löschen

Beim Löschen aus B^+-Bäumen gibt es einmal wieder das ganz normale Löschen, indem man einfach den zu löschenden Datensatz wegnimmt. In dem folgenden Beispiel ist aus dem Baum der Datensatz 52 gelöscht worden.



Nun gibt es aber noch den Fall, daß ein Blatt nicht mehr genügend oder gar keine Datensätze nach dem Löschen enthält. Wir können dann das Blatt ganz löschen. Dies kann jedoch auch soweit gehen, daß ein Knoten zu wenig Zeiger auf Blätter hat, da nur noch ein Blatt existiert. Ein solcher Knoten wäre unsinnig, da diesem Zeiger immer gefolgt würde und so Rechenzeit verschwendet werden würde. Der Baum muß allerdings ausbalanciert sein. Deshalb wenden wir einen Trick an und borgen vom benachbarten Knoten einen Knoten. Dieses borgen kann wieder rekursiv bis zur Wurzel stattfinden. Dies kann bis dahin gehen, daß der Baum nur noch einen Knoten hat, nämlich die Wurzel selbst, in welchem auch die Datensätze gespeichert sind.

Im folgenden Beispiel löschen wir 56 und 77 und borgen uns daraufhin ein Blatt vom linken Nachbarn, nämlich das Blatt 41,X. Den Schlüssel in den Knoten passen wir entsprechend an:

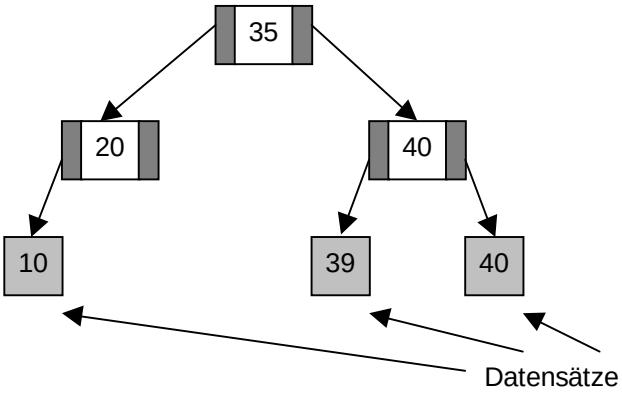


4. Andere Varianten von B-Bäumen

4.1 Binärer Baum

Ein binärer Baum unterscheidet sich vom B^+-Baum, daß es an jedem Knoten immer nur zwei Entscheidungen gibt. Entweder man folgt dem linken Zeiger oder dem rechten Zeiger. Im Gegensatz dazu gibt es in B^+-Bäumen manchmal 50 oder noch mehr Zeiger in einem Knoten. Binäre Bäume sind häufig auch nicht ausbalanciert.

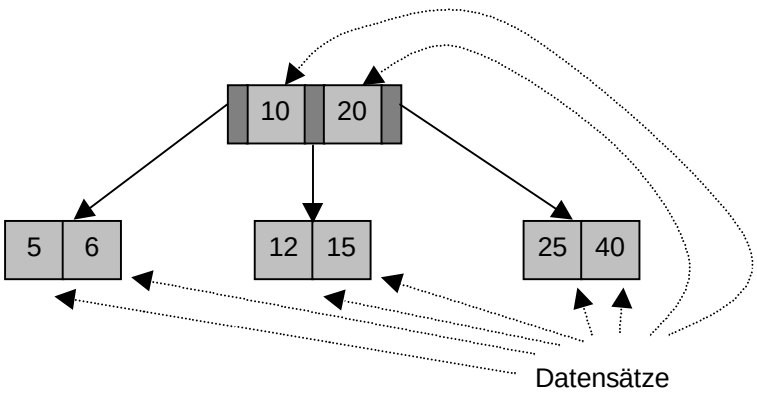
Beispiel für einen binären Baum:



4.2 B-Baum

Bei normalen B-Bäumen stehen die Datensätze nicht nur in den Blättern, sondern auch in den Knoten, die noch keine Blätter sind. Damit wird verhindert, daß eine Information redundant im Baum vorhanden ist. B-Bäume werden jedoch in der Praxis selten verwendet, weil man in B-Bäumen nicht sequentiell alle Datensätze in einem Rutsch lesen kann, wie es in B^+-Bäumen der Fall ist, wo jedes Blatt durch eine Liste noch einmal extra verkettet ist.

Beispiel für einen B-Baum:



6. Quellenverzeichnis

[SiKoSu97]
Abraham Silberschatz; Henry F. Korth; S. Sudarshan: Database System Concepts. Third Edition, Kapitel 11, McGraw-Hill 1997, ISBN 0-07-044756-X

[Comer97]
Douglas Comer: The Ubiquitous B-Tree. In: ACM Computing Surveys, 11:2., pages 121-137, June 1979.