B-Bäume*

Hans Jürgen Ohlbach

18. April 2018

Keywords: Suchbäume, B-Bäume,B+-Bäume

Empfohlene Vorkenntnisse: Bäume, siehe Baeume.pdf

Inhaltsverzeichnis

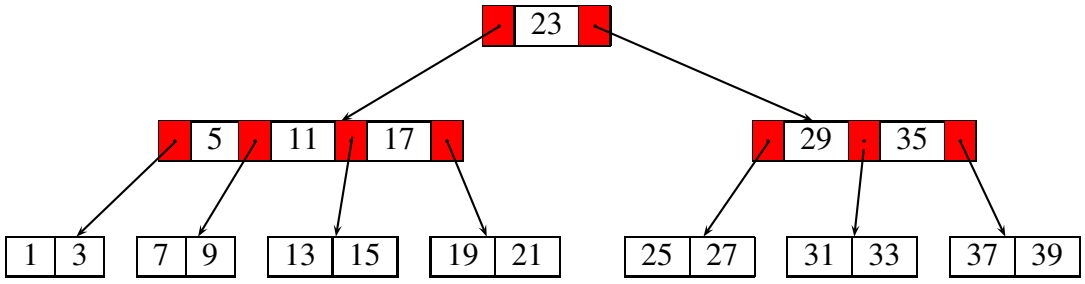
*Dieser Text ist Teil einer Sammlung von Miniskripten zur Einführung in die Informatik. Er ist in erster Linien für Nichtinformatiker gedacht, kann aber natürlich auch als erste Einführung für Informatiker nützlich sein.

1 Struktur und Zweck von B-Bäumen

Suchbäume sind Bäume, deren Knoten Daten aus einer Menge mit einer Totalordnung enthalten. Ihr Hauptzweck ist die Optimierung der Suche nach einem konkreten Wert. Sie sind eine Alternative zu Arrays, deren Einträge sortiert sind, so dass *binäre Suche* sehr schnell zu einem Ergebnis führt. Allerdings ist bei Arrays das Einfügen von neuen Elementen und Löschen von Einträgen sehr aufwendig. Sind das häufige Operationen, dann sind Suchbäume deutlich effizienter als Arrays.

B-Bäume sind Suchbäume, allerdings nicht unbedingt binäre Suchbäume. Im Gegensatz zu AVL-Bäumen oder Rot-Schwarz-Bäumen können sie in den einzelnen Knoten mehr als einen Schlüssel haben.

Ein B-Baum könnte so aussehen:



Ein B-Baum

Die Motivation für diese ausgedehnten Knoten kommt aus den Datenbankanwendungen. Dabei liegt die Datenbank, sowie ein ganzer B-Baum zunächst in dem externen Speicher. Weil in vielen Anwendungen riesige Datenmengen verwaltet werden müssen, sind auch die Bäume riesig und werden nicht am Stück in den Hauptspeicher geladen. Es bietet sich dabei an, den Baum knotenweise in den Hauptspeicher zu laden.

Die Daten können nur Clusterweise in den Hauptspeicher gelesen werden. Daher packt man soviel Information in einen Knoten, dass ein Cluster voll wird. Wenn der Knoten im Hauptspeicher ist, dann ist die Suche innerhalb des Knotens schnell. Das Nachladen eines Kindknotens aus dem externen Speicher ist dagegen viel aufwendiger.

Die Struktur eines B-Baumes muss dabei folgenden Bedingungen genügen:

- die Schlüssel innerhalb eines Knotens sind aufsteigend sortiert
- die Schlüssel der Kindknoten, auf die ein Zeiger links eines Knotens mit Schlüssel n zeigt, sind kleiner als n, sowie
  die Schlüssel der Kindknoten, auf die ein Zeiger rechts eines Knotens mit Schlüssel n zeigt, sind größer als n.
- Alle Blattknoten befinden sich in gleicher Tiefe

- Es wird ein *Verzweigungsgrad* t festgelegt mit den Bedingungen:
  - alle Knoten außer der Wurzel haben mindestens $t-1$ und höchstens $2t-1$ Schlüssel und mindestens t und höchstens $2t$ Kindverweise, wenn es sich um innere Knoten handelt.
  - Die Wurzel hat mindestens 1 und höchstens $2t-1$ Schlüssel, wenn der B-Baum nicht leer ist, und mindestens 2 und höchstens $2t$ Kindverweise, wenn die Höhe des Baumes größer als 0 ist.

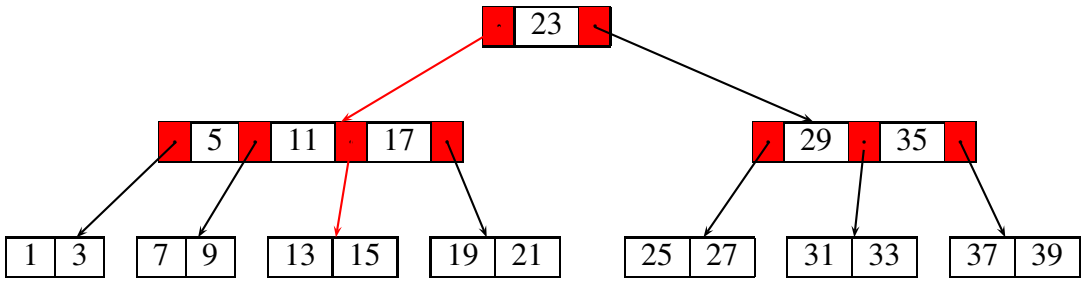
Der begrenzte Verzweigungsgrad t bewirkt, dass die Knoten nicht beliebig breit sein können. $t = 1024$ könnte aber nichtsdestotrotz eine sinnvolle Größe sein.

Bei $t = 2$ gibt es für jeden Knoten 2-4 Kindknoten. Dieser Spezialfall wird auch *2-3-4-Baum* genannt und speziell implementiert.

Die Tiefe des Baumes mit n Knoten ergibt sich dann zu $\mathcal{O}(\log_t(n))$. Im Vergleich zu einem Binärbaum ist ein B-Baum mit $t = 1024$ dann 10 mal flacher. Der Abstieg in die Tiefe ist damit 10 mal schneller.

2 Suche im B-Baum

Die Suche nach einem bestimmten Schlüssel in einem B-Baum geht geht ganz analog zur Suche in binären Suchbäumen. Der einzige Unterschied ist, dass man für einen Wert n innerhalb eines Knotens den Schlüssel n selbst, oder den richtigen Zeiger am Anfang oder Ende oder zwischen den Schlüsseln mit $n_1 < n < n_2$ zu dem Kindknoten suchen muss. Im folgenden Baum wird der Pfad zum Schlüssel 15 rot angezeigt.



Suche nach 15

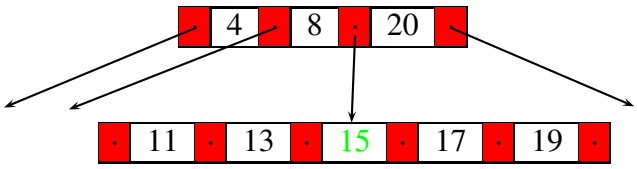
Da die Baumtiefe $\mathcal{O}(\log(n))$ ist, und die Knotenbreite durch den Verzweigungsgrad t begrenzt ist, ist auch der Zeitaufwand für die Suche $\mathcal{O}(\log(n))$. Ist t sehr groß, kann man die Suche innerhalb eines Knotens mit binärer Suche auf einen Zeitaufwand von $\mathcal{O}(\log(t))$ reduzieren.

3 Einfügen eines neuen Schlüssels

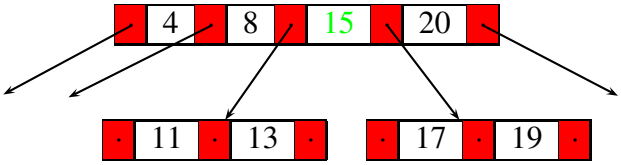
Zunächst kann jeder neue Schlüssel an die richtige Position *in einem Blattknoten* eingebaut werden. Dabei kann es jedoch passieren, dass der Knoten breiter wird als der Verzweigungsgrad erlaubt. In diesem Fall muss der Knoten aufgespalten werden.

Im nächsten Beispiel wird angenommen, der untere Knoten ist zu breit. Wegen der Bedingung, dass die Breite $2t-1$ nicht überschreitet, muss die Anzahl der Schlüssel ungerade sein. In diesem Fall kann man den mittleren Schlüssel, egal ob dieser oder ein anderer eingefügt wurde, nach oben verschieben.

Am mittleren Schlüssel, der 15, wird der Knoten getrennt.



Die 15 wird nach oben verschoben:



Durch das Verschieben nach oben kann der obere Knoten jetzt zu breit werden, so dass dieser ebenfalls getrennt werden muss. Dies kann sich rekursiv bis zur Wurzel fortsetzen.

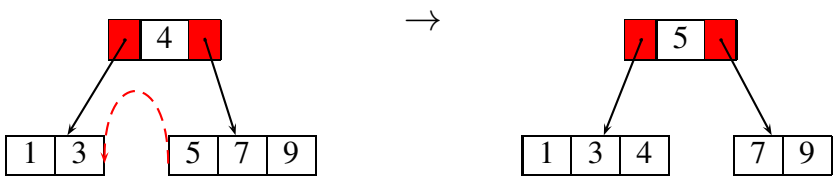
4 Löschen eines Schlüssels

Je nachdem, ob der Schlüssel in einem Blattknoten oder in einem inneren Knoten ist, müssen verschiedene Operationen durchgeführt werden.

4.1 Löschen eines Schlüssels aus einem Blattknoten

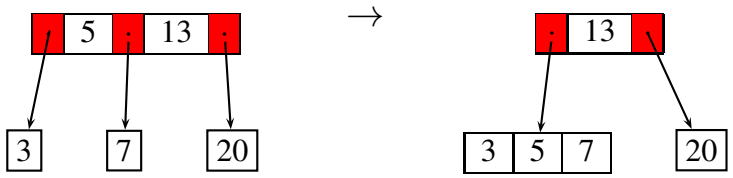
Hierbei wird die Bedingung relevant, dass Knoten *mindestens* $t-1$ Schlüssel haben müssen. Bleiben nach Löschen eines Schlüssels noch genügend viele übrig, muss nichts getan werden. Bleiben allerdings zu wenige übrig, werden die Knoten durch Verschieben und Verschmelzen so umgebaut, dass der Schlüssel anschließend ohne Verletzung der $t-1$-Bedingung gelöscht werden kann.

Das erste Beispiel zeigt, wie eine Zelle von rechts nach links verschoben wird. Der Schlüssel, die 5, wird von unten nach oben verschoben, und der Schlüssel oben, die 4 nach links unten.



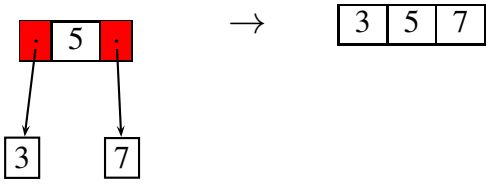
Ganz analog kann man auch eine Zelle von links nach rechts verschieben.

Ein Sonderfall ist, wenn beim Verschieben der Knoten ganz leer wird.
Im Beispiel unten wandert nicht nur die 5 von oben nach links unten, sondern die 7 schließt sich dem linken Blattknoten an.



Jetzt könnte man z.B. die 7 löschen.

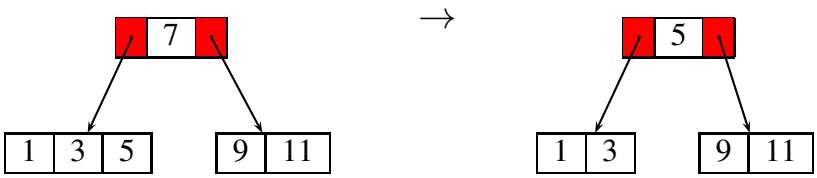
Wenn im Beispiel oben der obere Knoten nur einen Schlüssel enthält, dann rutscht der Blattknoten eine Ebene höher.



4.2 Löschen eines Schlüssels aus einem inneren Knoten

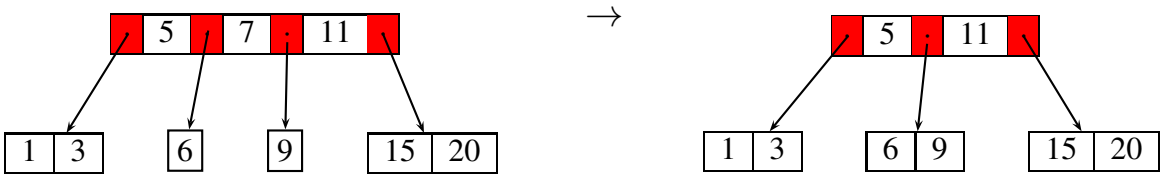
Ein Schlüssel in einem inneren Knoten kann gelöscht werden, indem man einen Schlüssel von einem unteren Knoten an die Stelle des gelöschten Schlüssels schiebt.

Im Beispiel wird die 7 gelöscht, indem man die 5 an ihre Stelle schiebt.



Man könnte auch die 9 an die Stelle der 7 schieben, nimmt aber i.A. die Seite, wo mehr Schlüssel drin sind. Das geht natürlich nur, wenn in einem der Kindknoten noch genügend viele Schlüssel übrig bleiben. Werden beide Kindknoten zu klein, muss man sie verschmelzen.

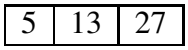
Im folgenden Beispiel wird die 7 gelöscht, und daher 6 und 9 verschmolzen.



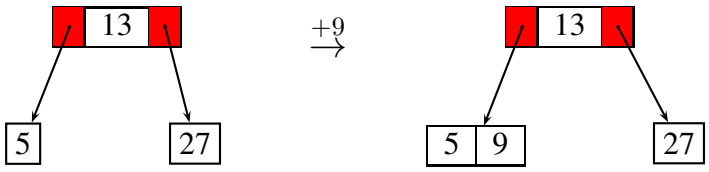
5 Ein größeres Beispiel

Wir entwickeln einen Baum mit $t = 2$. Ein solcher Baum kann minimal einen und maximal drei Schlüssel in einem Knoten speichern.

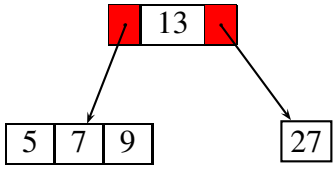
Die ersten drei Schlüssel, 5, 13 und 27 werden in den Wurzelknoten eingebaut:



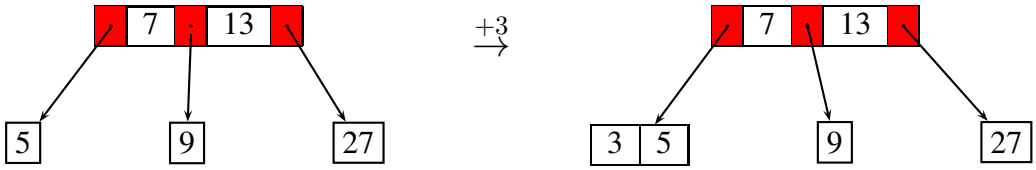
9 wird hinzugefügt:
In dem Knoten ist kein Platz mehr. Er muss erst getrennt werden:



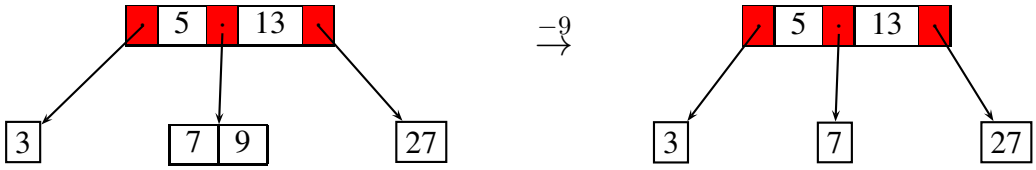
7 wird hinzugefügt:



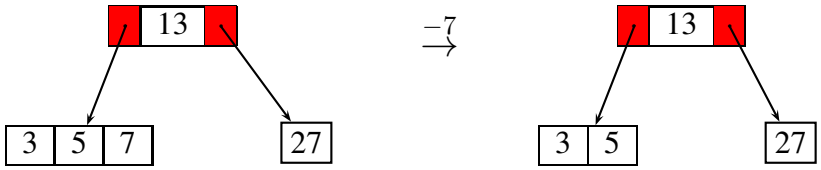
3 wird hinzugefügt:
Der linke Blattknoten würde zu groß werden. Er muss erst getrennt werden.



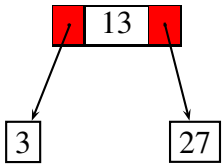
9 wird gelöscht:
Der Knoten würde leer werden. Daher wird die rechte Zelle aus dem linken Blattknoten nach rechts geschoben, indem die 5 nach oben wandert und die 7 nach unten. Dann kann die 9 gelöscht werden.



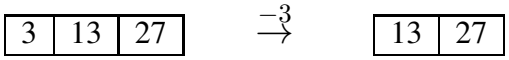
7 wird gelöscht:
Da der Blattknoten mit der 7 leer würde, werden die beiden Nachbarknoten erst verschmolzen. Dann kann die 7 gelöscht werden.



5 wird gelöscht:
Das geht direkt



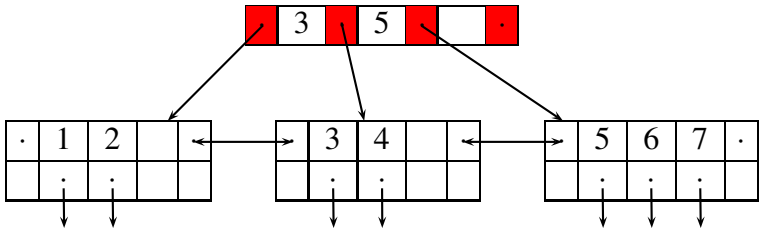
3 wird gelöscht:
Zunächst müssen die Knoten verschmolzen werden. Dann kann die 3 gelöscht werden.



6 B+-Bäume

Eine Variante der B-Bäume sind die B+-Bäume, bei der die Schlüssel nur in den Blattknoten gespeichert sind. Die inneren Knoten haben auch Schlüssel. Diese sind aber nur für die Suche gedacht. Zusätzlich haben die Blattknoten noch Verweise auf die Nachbarknoten. Damit wirken die Blattknoten wie verkettete Listen und können effizient durchlaufen werden.

Der folgende B+-Baum hat die Schlüssel 1,2,3,4,5,6,7. Jeder Knoten hat eine feste Anzahl Zellen, die auch leer sein können. Damit kann man die Knoten als Arrays implementieren, deren Größe sich nicht ändert. Die Schlüssel in dem Wurzelknoten sind nicht Bestandteil der Daten. Die Blattknoten sind als doppelt verkettete Liste implementiert. Jeder Schlüsseln in den Blattknoten hat auch noch einen Zeiger auf weitere Daten, die zu dem Schlüssel gehören. Damit kann man auch komplexe Datentypen mit dem B+-Baum organisieren. Man könnte auch noch weitere B+-Bäume haben, die auf dieselben Daten zeigen, aber eine andere Ordnung benutzen.



Ein Beispiel mit mehreren B+-Bäumen für die gleichen Daten wären Studentendaten. Ein *primärer Schlüssel* könnten die Matrikelnummern sein. Dann hätte man einen B+-Baum, wo die Schlüssel Zahlen sind (Matrikelnummern), mit den Verweisen auf die eigentlichen Studentendaten. Ein weiterer *sekundärer Schlüssel* könnten die Namen sein. Dann hätte man einen zweiten B+-Baum, mit Strings als Schlüssel, und der alphabetischen Ordnung auf Strings. Die Verweise zeigen dann ebenfalls auf die eigentlichen Studentendaten.

Die Suche in B+-Bäumen geht fast genauso wie in B-Bäumen, nur dass sie immer bis zu den Blattknoten laufen muss. Einfügen und Löschen von neuen Knoten ist ähnlich wie bei B-Bäumen, muss aber im Detail der neuen Struktur angepasst werden.

Stichwortverzeichnis