B-Bäume

Weiterführende Literatur:

- *Wikipedia-Artikel „B-Baum“*
- Saake und Sattler, *Algorithmen und Datenstrukturen*, Kapitel 14.4.3, Seite 386-399 (PDF 402-415)
- Schneider, *Taschenbuch der Informatik*, Kapitel 13.5.4.2 Balancierte Mehrwegbäume, Seite 464, wird nur erwähnt, nicht beschrieben
- Kemper und Eickler, *Datenbanksysteme*, 7.8 B-Bäume Seite 224-228

`https://www.cs.usfca.edu/~galles/visualization/BTree.html` Max. Degree = 5 entspricht k = 2

Eine ausgeglichene Baumstruktur ist der von R. Bayer und E. McCreight entwickelte B-Baum. Hierbei *steht der Name „B“ für balanciert*, breit, buschig oder auch Bayer, nicht jedoch für binär. Die Grundidee des B-Baumes ist es gerade, dass der Verzweigungsgrad variiert, während die Baumhöhe vollständig ausgeglichen ist.[1]

steht der Name „B“ für balanciert

Definition

Ein Baum heißt genau dann B-Baum, wenn gilt:

(a) Jeder Knoten außer der Wurzel enthält zwischen k *und* $2k$ *Elemente* (Schlüsselwerte), k wird als *Ordnung* des B-Baums bezeichnet.

k und $2k$ Elemente

Ordnung

(b) Jeder *Knoten ist entweder ein Blatt* (*ohne Kinder*) oder hat *mindestens* $k+1$ *und höchstens* $2k+1$ *Kind-Knoten*.

Knoten ist entweder ein Blatt (ohne Kinder)

mindestens $k+1$ und höchstens $2k+1$ Kind-Knoten

(c) Der Wurzelknoten ist *entweder ein Blatt oder hat mindestens 2 Nachfolger*.

entweder ein Blatt oder hat mindestens 2 Nachfolger

(d) Alle Blätter haben die *gleiche Tiefe*, d. h. alle Wege von der Wurzel bis zu den Blättern sind gleich lang. Pfade haben die Länge $h-1$, wobei h die Höhe des gesamten Baums ist.

gleiche Tiefe

2

Einfügen

Das Einfügen in einen B-Baum erfolgt *nur in den Blattknoten*. Wenn in einem Blattknoten die *maximale Anzahl* von Elementen ($2k$) erreicht ist, findet ein *Split* statt, d. h. die Elemente werden aufgeteilt und ein neuer Knoten entsteht. Das *mittlere Element* des ursprünglichen Knotens wird dabei *in den Elternknoten integriert*.[3]

nur in den Blattknoten

maximale Anzahl

Split

mittlere Element

in den Elternknoten integriert

[1] Saake und Sattler, *Algorithmen und Datenstrukturen*, Seite 386.

[2] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 32.

[3] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 32.

Suchen

von links nach rechts
Stimmt
überein
gefunden
Element größer
links
fortgesetzt
Element kleiner
nächsten Element der Wurzel wiederholt
rechten Unterbaum

Beginnend mit dem Wurzelknoten werden die Knoten jeweils *von links nach rechts* durchsucht: *Stimmt* ein Element mit dem gesuchtem Schlüsselwert *überein*, ist der Satz *gefunden*. Ist das *Element größer* als der gesuchte Wert, wird die Suche im *links* hängenden Unterbaum *fortgesetzt*. Ist das *Element kleiner* als der gesuchte Wert, wird der Vergleich mit dem *nächsten Element der Wurzel wiederholt*. Ist auch das letztes Element der Wurzel noch kleiner als der gesuchte Wert, dann wird die Suche im *rechten Unterbaum* des Elements fortgesetzt. Falls ein weiterer Abstieg in den Unterbaum nicht möglich ist (d. h. Blattknoten), wird die Suche abgebrochen. Dann ist kein Satz mit dem gewünschten Schlüsselwert vorhanden.[4]

Löschen

Suche den Knoten, in dem das zu löschende Element liegt.[5]

Löschen aus Blattknoten

Verschiebung / Rotation / Ausgleichen

Enthält der für den Abstieg ausgewählte Unterbaum nur die minimale Schlüsselanzahl, aber ein vorausgehender oder nachfolgender Geschwisterknoten hat genügend Schlüssel, wird ein Schlüssel in den ausgewählten Knoten verschoben.[6]

Verschmelzung / Mischen

genau die minimale Schlüsselanzahl

Vaterknoten

Enthalten sowohl der für den Abstieg ausgewählte Unterbaum als auch sein unmittelbar vorausgehender und nachfolgender Geschwisterknoten *genau die minimale Schlüsselanzahl*, ist eine Verschiebung nicht möglich. In diesem Fall wird eine Verschmelzung des ausgewählten Unterbaumes mit dem vorausgehenden oder nachfolgenden Geschwisterknoten durchgeführt. Dazu wird der Schlüssel aus dem *Vaterknoten*, welcher die Wertebereiche der Schlüssel in den beiden zu verschmelzenden Knoten trennt, als mittlerer Schlüssel in den verschmolzenen Knoten verschoben. Die beiden Verweise auf die jetzt verschmolzenen Kindknoten werden durch einen Verweis auf den neuen Knoten ersetzt.[7]

Ein Unterlauf entsteht auf Blattebene. Der Unterlauf wird durch Mischen des Unterlaufknotens mit seinem Nachbarknoten und dem darüberliegenden Elemen durchgeführt, dabei wird sozusagen ein Splitt rückwärts durchgeführt. Wurde einmal mit dem Mischen auf Blattebene begonnen, muss das evtl. nach oben fortgesetzt werden. Mischen wird so lange fortgesetzt, bis kein Unterlauf mehr existiert oder die Wurzel erreicht ist. Wird die Wurzel erreicht, kann der

[4] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 37.
[5] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 39.
[6] *Wikipedia-Artikel „B-Baum“*.
[7] *Wikipedia-Artikel „B-Baum“*.

Baum in der Höhe um 1 schrumpfen. Beim Mischen kann auch wieder ein Überlauf entstehen, d. h. es muss wieder gesplittet werde.[8]

Löschen aus inneren Knoten

Wird der zu löschende Schlüssel bereits in einem inneren Knoten gefunden, kann dieser nicht direkt gelöscht werden, weil er für die Trennung der Wertebereiche seiner beiden Unterbäume benötigt wird. In diesem Fall wird sein *symmetrischer Vorgänger* (oder sein symmetrischer *Nachfolger*) gelöscht und *an seine Stelle kopiert*. Der symmetrische Vorgänger ist der größte Blattknoten im linken Unterbaum, befindet sich also dort ganz rechts außen. Der symmetrische Nachfolger ist entsprechend der kleinste Blattknoten im rechten Unterbaum und befindet sich dort ganz links außen. Die *Entscheidung*, in welchen Unterbaum der Abstieg für die Löschung stattfindet, wird davon abhängig gemacht, *welcher genügend Schlüssel enthält*. Haben beide nur die minimale Schlüsselanzahl, werden die Unterbäume verschmolzen. Damit wird keine Trennung der Wertebereiche mehr benötigt und der Schlüssel kann direkt gelöscht werden.[9]

symmetrischer Vorgänger
Nachfolger
an seine Stelle kopiert
Entscheidung
welcher genügend Schlüssel enthält

Falls das Element E in einem inneren Knoten liegt, dann untersuche den linken und rechten Unterbaum von E:

- Betrachte den Blattknoten mit dem direkten Vorgänger E' von E und den Blattknoten mit direktem Nachfolger E'' von E.
- Wähle den Blattknoten aus, der mehr Elemente hat. Falls beide Blattknoten gleich viele Elemente haben, wähle zufällig einen der beiden aus.
- Ersetze das zu löschende Element E durch E' bzw. E'' aus dem gewählten Blattknoten.
- Lösche E' bzw. E'' im gewählten Blattknoten und behandle ggf. entstehenden Unterlauf in diesem Blattknoten.[10]

```
3  import java.util.Vector;
4
5  /**
6   * Eine Implemenation eines B-Baum (nach Saake Seite 388-389).
7   */
8  @SuppressWarnings({ "rawtypes", "unchecked" })
9  public class BBaum {
10
11   /**
12    * Ein Seite (bzw. ein Knoten) eines B-Baums. Eine Seite enthält mehrer
13    * Schlüsselwerte und mehrere Verweise auf andere Seiten.
14    */
15   public class BBaumSeite {
16
17     /**
18      * Eine Konstante für eine Blatt-Seite, d. h. die Seite hat keine Kinder.
19      */
20     public static final int BLATT_SEITE = 0;
21
22     /**
```

[8] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 40.
[9] *Wikipedia-Artikel „B-Baum“*.
[10] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5*, Seite 39.

```
23        * Eine Konstante für eine Knoten-Seite, d. h. die Seite hat Kinder.
24        */
25       public static final int INNERE_SEITE = 1;
26
27       /**
28        * Diese Konstante wird verwendet, wenn der Schlüsselwert gefunden wird.
29        */
30       public static final int SCHLÜSSEL_GEFUNDEN = -1;
31
32       /**
33        * Diese Konstante wird verwendet, wenn der Schlüsselwert nicht gefunden
    ↪  wird.
34        */
35       public static final int SCHLÜSSEL_NICHT_GEFUNDEN = -2;
36
37       /**
38        * Ein Verweis zum Elternknoten.
39        */
40       BBaumSeite eltern = null;
41
42       /**
43        * Eine Liste zum Speichern der Schlüssel.
44        */
45       Vector<Comparable> schlüsselListe;
46
47       /**
48        * Eine Liste zum Speichern von Verweisen.
49        */
50       Vector<BBaumSeite> kinderListe;
51
52       /**
53        * Der Type einer Seite (Blatt-Seite (0) oder innere Seite (1)).
54        */
55       int seitenTyp;
56
57       public BBaumSeite(int seitenTyp) {
58         this.seitenTyp = seitenTyp;
59         schlüsselListe = new Vector<Comparable>();
60         // Zeiger werden nur bei inneren Seite angelegt.
61         kinderListe = (seitenTyp == INNERE_SEITE ? new Vector<BBaumSeite>() :
     ↪  null);
62       }
63
64       public int gibAnzahlSchlüssel() {
65         return schlüsselListe.size();
66       }
67
68       /**
69        * Gib den Schlüssel durch die Index-Nummer zurück. 0 gibt den ersten
    ↪  Schlüssel
70        * in der Liste, 1 den zweiten etc.
71        *
72        * @param index Eine Index-Nummer von 0 weg gezählt.
73        *
74        * @return Den Schlüsselwert.
75        */
76       public Comparable gibSchlüssel(int index) {
77         return schlüsselListe.get(index);
78       }
79
80       /**
81        * Gib die Anzahl der Kinder-Seiten zurück.
```

```
82        *
83        * @return Die Anzahl der Kinder-Seiten.
84        */
85       public int gibAnzahlKinder() {
86         if (kinderListe != null)
87           return kinderListe.size();
88         return 0;
89       }
90
91       /**
92        * Gib Kinder dieser Seite als Vector zurück.
93        *
94        * @return Instanzen des Klasse {@link BBaumSeite} als Vector.
95        */
96       public Vector gibKinder() {
97         return kinderListe;
98       }
99
100      public BBaumSeite gibKindDurchIndex(int index) {
101        return kinderListe.get(index);
102      }
103
104      public BBaumSeite gibEltern() {
105        return eltern;
106      }
107
108      public void setzeSeitenType(int seitenTyp) {
109        this.seitenTyp = seitenTyp;
110        if (kinderListe == null && seitenTyp == INNERE_SEITE)
111          kinderListe = new Vector<BBaumSeite>();
112      }
113
114      /**
115       * Suche den Schlüssel in der Seite (nach Saake Seite 391).
116       *
117       * @param schlüssel Der Schlüsselwert, nach dem gesucht wird.
118       * @param ergebnis  In diesem Feld wird das Ergebnis gespeichert.
119       *
120       * @return Wird der Schlüssel gefunden, geben wird SCHLÜSSEL_GEFUNDEN (-1)
121       *         zurück. Wenn es ein innerer Knoten ist, geben wir den letzten
    ↪  Verweis
122       *         zurück, im Fall eines Blattes SCHLÜSSEL_NICHT_GEFUNDEN (-2).
123       */
124      public int findeSchlüsselInSeite(Comparable schlüssel, Comparable[] ergebnis)
       ↪  {
125        for (int i = 0; i < schlüsselListe.size(); i++) {
126          int erg = schlüsselListe.get(i).compareTo(schlüssel);
127          if (erg == 0) {
128            ergebnis[0] = schlüsselListe.get(i);
129            return SCHLÜSSEL_GEFUNDEN;
130          } else if (erg > 0)
131            return seitenTyp == INNERE_SEITE ? i : SCHLÜSSEL_NICHT_GEFUNDEN;
132        }
133        return (seitenTyp == INNERE_SEITE ? schlüsselListe.size() :
         ↪  SCHLÜSSEL_NICHT_GEFUNDEN);
134      }
135
136      /**
137       * Füge einen Schlüsselwert in eine Seite ein. (Vergleiche Saake Seite
    ↪  395-396.)
138       *
139       * @param schlüssel          Der Schlüsselwert, der eingefügt werden soll.
```

```
140         * @param linkesGeschwister  Die linke Geschwister-Seite.
141         * @param rechtesGeschwister Die rechte Geschwister-Seite.
142         *
143         * @return Wahr, wenn der Schlüsselwert in die Seite eingefügt werden konnte.
144         */
145        public boolean fügeInSeiteEin(Comparable schlüssel, BBaumSeite
            ↪  linkesGeschwister, BBaumSeite rechtesGeschwister) {
146          boolean erledigt = false;
147          // Position für Schlüssel suchen
148          for (int i = 0; i < schlüsselListe.size(); i++) {
149            int ergebnis = schlüsselListe.get(i).compareTo(schlüssel);
150            if (ergebnis == 0) {
151              // Schlüssel existiert schon -> ignorieren
152              erledigt = true;
153              break;
154            } else if (ergebnis > 0) {
155              // Stelle gefunden -> einfügen
156              schlüsselListe.insertElementAt(schlüssel, i);
157              if (rechtesGeschwister != null) {
158                kinderListe.insertElementAt(rechtesGeschwister, i + 1);
159                rechtesGeschwister.eltern = this;
160              }
161              erledigt = true;
162              break;
163            }
164          }
165          if (!erledigt) {
166            // Schlüssel muss am Ende eingefügt werden
167            schlüsselListe.add(schlüssel);
168            if (linkesGeschwister != null && kinderListe.isEmpty()) {
169              kinderListe.add(linkesGeschwister);
170              linkesGeschwister.eltern = this;
171            }
172            if (rechtesGeschwister != null) {
173              kinderListe.add(rechtesGeschwister);
174              rechtesGeschwister.eltern = this;
175            }
176          }
177          // Knoten zu groß
178          return schlüsselListe.size() > ordnung * 2;
179        }
180
181        /**
182         * Teile eine Seite (Nach Saake Seite 396)
183         *
184         * @return Die neu erzeugte Geschwisterseite.
185         */
186        public BBaumSeite teile() {
187          int pos = gibAnzahlSchlüssel() / 2;
188          // Geschwisterknoten erzeugen
189          BBaumSeite geschwister = new BBaumSeite(seitenTyp);
190          for (int i = pos + 1; i < gibAnzahlSchlüssel(); i++) {
191            // die obere Hälfte der Schlüssel und Verweise kopieren
192            geschwister.schlüsselListe.add(this.gibSchlüssel(i));
193            if (seitenTyp == BBaumSeite.INNERE_SEITE)
194              geschwister.kinderListe.add(this.gibKindDurchIndex(i));
195          }
196          // es gibt einen Verweis mehr als Schlüssel
197          if (seitenTyp == BBaumSeite.INNERE_SEITE)
198
               ↪  geschwister.kinderListe.add(this.gibKindDurchIndex(gibAnzahlSchlüssel()));
199          // und anschließend im Originalknoten löschen
```

```
200         for (int i = gibAnzahlSchlüssel() - 1; i >= pos; i--) {
201           schlüsselListe.remove(pos);
202           if (seitenTyp == BBaumSeite.INNERE_SEITE)
203             kinderListe.remove(pos + 1);
204         }
205         return geschwister;
206       }
207     }
208
209     private BBaumSeite wurzel = null;
210
211     private int ordnung;
212
213     /**
214      * Dieser Konstruktur erzeugt einen B-Baum mit einer bestimmten Ordnung.u
215      *
216      * @param ordnung Die Ordnung des B-Baums. Ist die Ordnung beispielsweise 2,
217      *                dann muss jede Seite mindestens 2 Knoten und maximal 4 Knoten
218      *                aufweisen.
219      */
220     public BBaum(int ordnung) {
221       this.ordnung = ordnung;
222       wurzel = new BBaumSeite(BBaumSeite.BLATT_SEITE);
223     }
224
225     /**
226      * Gib die oberste Seite, die sogenannte Wurzel zurück.
227      *
228      * @return Die Wurzelseite.
229      */
230     public BBaumSeite gibWurzel() {
231       return wurzel;
232     }
233
234     /**
235      * Finde einen Schlüsselwert im B-Baum. (nach Saake Seite 391)
236      *
237      * @param schlüssel Der Schlüsselwert, der gesucht wird.
238      *
239      * @return Den gefunden Schlüsselwert.
240      */
241     public Comparable finde(Comparable schlüssel) {
242       BBaumSeite seite = wurzel; // Startknoten
243       boolean beendet = false;
244       Comparable[] ergebnis = { null };
245       do {
246         // Suche Schlüssel im aktuellen Knoten
247         int index = seite.findeSchlüsselInSeite(schlüssel, ergebnis);
248         if (index == BBaumSeite.SCHLÜSSEL_GEFUNDEN || index ==
          ↪  BBaumSeite.SCHLÜSSEL_NICHT_GEFUNDEN)
249           // Schlüssel gefunden oder auf einem
250           // Blattknoten nicht gefunden -> fertig
251           beendet = true;
252         else
253           // anderenfalls Verweis verfolgen
254           seite = seite.gibKindDurchIndex(index);
255       } while (!beendet);
256       return ergebnis[0];
257     }
258
259     /**
260      * Füge einen Schlüsselwert in den B-Baum ein. (nach Saake Seite 393-394)
```

```
261       *
262       * @param schlüssel Der Schlüsselwert, der eingefügt werden soll.
263       */
264      public void fügeEin(Comparable schlüssel) {
265        BBaumSeite linkesGeschwister = null, rechtesGeschwister = null;
266        // Suche Blattknoten, der den Schlüssel aufnimmt
267        BBaumSeite seite = findeBlattSeite(schlüssel);
268        // Schlüssel einfügen
269        while (seite.fügeInSeiteEin(schlüssel, linkesGeschwister,
           ↪  rechtesGeschwister)) {
270          // Split erforderlich
271          int pos = seite.gibAnzahlSchlüssel() / 2;
272          schlüssel = seite.gibSchlüssel(pos);
273          BBaumSeite eltern = seite.gibEltern();
274          if (eltern == null)
275            // ein neuer Elternknoten muss angelegt werden
276            eltern = new BBaumSeite(BBaumSeite.INNERE_SEITE);
277          // Split durchführen
278          linkesGeschwister = seite;
279          rechtesGeschwister = seite.teile();
280          // Wurzel anpassen
281          if (wurzel == seite)
282            wurzel = eltern;
283          // der aktuelle Knoten ist jetzt der Elternknoten
284          seite = eltern;
285          // und der muss ein innerer Knoten sein
286          seite.setzeSeitenType(BBaumSeite.INNERE_SEITE);
287        }
288      }
289
290      /**
291       * Füge mehrere Schlüsselwerte auf einmal ein.
292       *
293       * @param mehrereSchlüssel Ein Feld mit mehreren Schlüsselwerten.
294       */
295      public void fügeEin(Comparable... mehrereSchlüssel) {
296        for (Comparable schlüssel : mehrereSchlüssel) {
297          fügeEin(schlüssel);
298        }
299      }
300
301      /**
302       * Finde die Blatt-Seite, in der der Schlüsselwert gespeichert ist. (nach Saake
303       * Seite 395)
304       *
305       * @param schlüssel Der Schlüsselwert, nach dem gesucht werden soll.
306       *
307       * @return Die Blatt, in der der Schlüsselwert gespeichert ist.
308       */
309      private BBaumSeite findeBlattSeite(Comparable schlüssel) {
310        BBaumSeite seite = wurzel;
311        Comparable[] key = { null }; // wird eigentlich nicht benötigt
312        // den Baum von der Wurzel aus nach unten durchlaufen,
313        // bis ein Blattknoten gefunden wurde
314        while (seite.seitenTyp != BBaumSeite.BLATT_SEITE) {
315          // Verweis verfolgen
316          seite = seite.gibKindDurchIndex(seite.findeSchlüsselInSeite(schlüssel,
             ↪  key));
317        }
318        return seite;
319      }
320    }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/baum/BBaum.java`

Literatur

[1] Alfons Kemper und André Eickler. *Datenbanksysteme. eine Einführung*. 2013.

[2] *Qualifizierungsmaßnahme Informatik: Algorithmen und Datenstrukturen 5. Bäume, Hashing*. `https://www.studon.fau.de/file2619756_download.html`.

[3] Gunter Saake und Kai-Uwe Sattler. *Algorithmen und Datenstrukturen. Eine Einführung in Java*. 2014.

[4] Uwe Schneider. *Taschenbuch der Informatik*. 7. Aufl. Hanser, 2012. ISBN: 9783446426382.

[5] *Wikipedia-Artikel „B-Baum“*. `https://de.wikipedia.org/wiki/B-Baum`.