4.4. B-Bäume

Motivation

Die bisher betrachteten Baumstrukturen sind gut geeignet für *Collections* im Hauptspeicher, aber schlecht, wenn Baumstrukturen auf externen Speichern (d.h. *Platten*) abgelegt werden müssen. Das ist bei grossen Datenmengen der Fall, wenn von mehreren Programmen gleichzeitig darauf zugegriffen wird oder wenn man mit persistenten Daten arbeitet, die nicht alle in den Hauptspeicher kopiert werden sollen.

Betrachten wir als Beispiel eine eher kleine Datenbank über die Schweizer Bevölkerung:

- Daten über ca. $8 \cdot 10^6$ Personen
- Schlüssel (Annahme): 64 Byte
- Daten pro Person (Annahme): 2048 Byte

Insgesamt zu speichernde Daten:

*(2048 + 64) * $8 \cdot 10^6$ Byte = 16896000000 Byte = 15.7 GB. (Nur Daten, ohne allfällige Zeiger)*

Speichert man die Schlüssel in einem AVL-Baum, ergibt sich eine durchschnittliche Höhe von:[1]

*1.1 * log($8 \cdot 10^6$) ≈ 26*

Ein im Hauptspeicher abgelegter AVL-Baum besteht pro Datensatz aus einem Knoten (Objekt). Plattenspeicher (*Disks*) sind hingegen in Blöcken fixer Grösse organisiert, die grundsätzlich nur als Einheit gelesen und geschrieben werden können.

Moderne SSD haben sehr hohe Lesegeschwindigkeiten, wie 45‘000 Blöcke pro Sekunde.

1 Suchanfrage benötigt ca. 26 Zugriffe auf Blöcke. Das gibt etwa 1730 Suchanfragen pro Sekunde.

Bei sich (mit 7‘200 U/min) drehenden Disks lassen sich bis zu 120 Blöcke pro Sekunde lesen.

4 bis 5 Suchanfragen pro Sekunde.

Für die Zugriffszeit im Hauptspeicher rechnen wir mit 10ns, also 10^8 Zugriffen pro Sekunde.

Ca. 3846154 Suchanfragen pro Sekunde.

Wir benötigen eine Baumstruktur, auf der die üblichen Operationen (*Suchen*, *Einfügen*, *Löschen*) mit einer minimalen Anzahl Plattenzugriffe ausgeführt werden können. Wegen der langen Zugriffszeiten auf die Disk darf das ruhig ein paar CPU-Instruktionen brauchen.

Mehrwegbaum

Ein Baumknoten sollte gerade einem Diskblock entsprechen. Statt einem Binärbaum verwenden wir einen Mehrwegbaum: Jeder Knoten hat Kapazität für s Elemente und zeigt auf s+1 Nachfolger-Knoten. Ein Datentyp für einen Knoten in einem B-Baum könnte so programmiert werden:

```
class Node<K> {
          int m;
          K[] keys = (K[])new Object[CAPACITY];
          int[] data = new int[CAPACITY];
          int[] successor = new int[CAPACITY + 1];
}
```

Die minimale Höhe *h* bei *N* gespeicherten Elementen lässt sich dann wie folgt berechnen:

$$h = \lceil log_{s+1}(N+1) \rceil.$$

Im schlimmsten Fall kann dieser Baum aber auch zu einer Liste solcher Blöcke entarten. Das führt für eine Anfrage wiederum zu vielen Plattenzugriffen. Der Mehrwegbaum muss daher auch ausgeglichen sein. Diese Eigenschaften haben *B-Bäume*.

[1] Die Höhe von AVL-Bäumen mit *N* Elementen ist $1.1 \cdot log_2(N)$. Die Höhe von unausgeglichenen binären Such-Bäumen ist bei gleichverteilt eingefügten Schlüsseln im Schnitt $2.3 \cdot log_2(N)$.

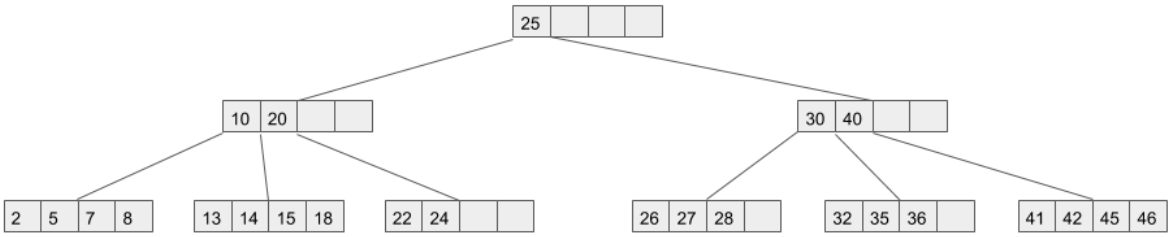
Definition: B-Baum [Bayer / McCreight 1970][2]

Achtung: Für B-Bäume wird der Begriff Ordnung *anders definiert, als für allgemeine Bäume![3]*

Ein B-Baum der Ordnung *n* ist ein Mehrwegsuchbaum vom Grad *2n + 1* mit folgenden Struktur-Eigenschaften:

1. Jeder Knoten enthält höchstens *2n* Elemente.
2. Jeder Knoten ausser der Wurzel enthält mindestens *n* Elemente (ist zur Hälfte gefüllt).
3. Jeder Knoten, der nicht Blattknoten ist, hat *m+1* Nachfolger, wobei *m* die Anzahl der Elemente ist.
4. Alle Blattknoten liegen auf derselben Stufe.
5. Ein Knoten enthält
   m Schlüssel + Daten (bzw. Verweise auf einen Datensatz)
   m+1 Referenzen auf Nachfolger.

Beispiel: B-Baum der Ordnung 2



4.4.1. Suchen im B-Baum

```
E find(Node<K> node, K key) {
   // search inside the node p, either sequentially or binary
   int i = 0;
   while (i < node.m && key.compareTo(node.keys[i]) > 0) {
       i++;
   }
   if (i < node.m && key.equals(node.keys[i])) {
       return dataBlock(node.data[i]);
   }
   if (node.isLeaf()) {
       return null;
   }
   Node<K> child = diskRead(node.successor[i]);
   return find(child, key);
}
```

Es ist allgemein schwierig, B-Bäume in Java für praktische Anwendungen zu implementieren:

- Man weiss nicht wie gross die Disk-Blöcke sind (Absicht: Java soll plattformunabhängig sein)
- Keine Kontrolle über Caching
- Daten müssen serialisiert werden

Deshalb betrachten wir die weiteren Operationen auf B-Bäumen ausschliesslich auf der Konzeptebene.

---

[2] B-Bäume wurden ursprünglich für relationale Datenbanken entwickelt, um die dort benötigten Indizes effizient speichern zu können. Dafür werden sie bis heute benutzt. Darüber hinaus sind in vielen Datei-Systemen die Verzeichnisse mit B-Bäumen aufgebaut.

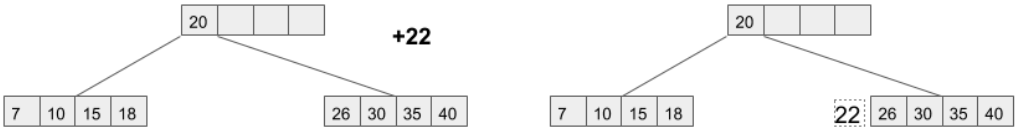
[3] Die allgemein übliche Definition von *Ordnung* in allgemeinen Bäumen entstand gleichzeitig wie die B-Bäume und die Begriffe wurden unabhängig voneinander definiert. Da in jeder der beiden Welten die jeweils verwendete Definition sehr sinnvoll ist, hat sich das bis heute so erhalten. Ein *B-Baum der Ordnung n* hat also gemäss des allgemeinen Ordnungsbegriffs für Bäume eine Ordnung von 2n+1. Das ist auch der Grad aller Knoten im B-Baum, wobei jedoch `null` als „Nachfolger“ zulässig ist.

4.4.2. Einfügen im B-Baum

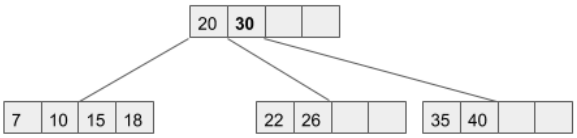
Neue Elemente werden immer in Blätter eingefügt. Dabei können folgende Situationen entstehen:

a) Neues Element wird in Seite mit $m < 2n$ Elementen eingefügt

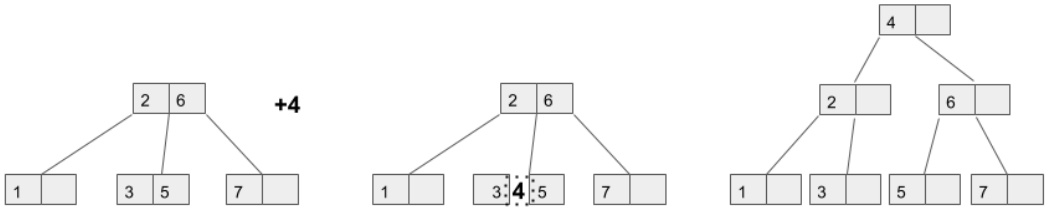
b) Seite ist voll, man muss sie splitten



- Schlüssel 22 hat keinen Platz im Blatt / das Blatt hat nun zu viele Schlüssel
- Das Blatt wird deshalb “gesplittet” und der mittlere Schlüssel dieser fünf (also die 30) wird in den Vater geschoben.
- Falls es zwei mittlere Schlüssel gibt, wird der kleinere der beiden in den Vater geschoben.



c) Seite ist voll, man muss sie und Vorgänger splitten



- Schlüssel 4 hat keinen Platz im Blatt / das Blatt hat nun zu viele Schlüssel
- Das Blatt wird deshalb “gesplittet” und der mittlere Schlüssel dieser drei (also die 4) wird in den Vater geschoben.
- Dort hat es auch keinen Platz, deshalb wird auch der Vater “gesplittet”. In unserem Beispiel wird dabei eine neue Wurzel erstellt.
- Das «Splitten» kann also auf dem Pfad vom Blatt zur Wurzel propagieren.

4.4.3. Löschen im B-Baum

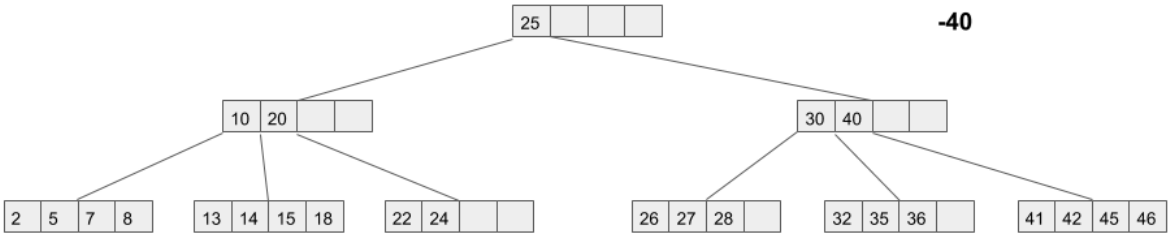
Beim Löschen eines Elementes aus einem B-Baum können folgende Situationen entstehen:

a) Element befindet sich in einem Blatt.

b) Element befindet sich in einem inneren Knoten

   In diesem Fall wird es durch das *nächstgrössere* Element ersetzt. Dieses befindet sich *immer* auf einem Blatt.

Beispiel:



In diesem Beispiel würde die 40 durch die 41 ersetzt und anschliessend die 41 im Blatt gelöscht (Fall a)).

Es müssen folgende Fälle beim Löschen aus einem Blattknoten unterschieden werden:

i. Blattseite hat $m > n$ Elemente

   In diesem Fall kann das Element einfach gelöscht werden. Beispiel:

ii. Blattseite hat n Elemente.

   Das Entfernen eines Elementes würde die Bedingung $m \geq n$ verletzen (ausser beim Wurzelknoten).

   In diesem Fall müssen wir die Bedingungen wieder herstellen. Dies erreichen wir dur das sogenannte «*Ausgleichen*» der Elemente in den inneren Knoten und Blättern. Beim Ausgleichen unterscheiden wir wiederum zwei Strategien, das «*Ausleihen*» und das «*Zusammenlegen*».

Ausgleichen:

1. *Ausleihen* von Elementen aus einem benachbarten Knoten (links oder rechts), falls dieser mehr als *n* Elemente enthält.

   Beispiel:

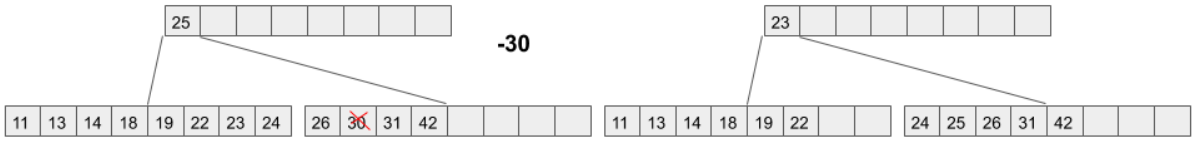
   - Nach dem Löschen hat das rechte Blatt zu wenig Elemente. Das linke Blatt ist allerdings mehr als zur Hälfte voll.
   - Deshalb wandert das Element mit dem *grössten* Schlüssel vom linken Blatt in den Vaterknoten und ein Element des Vaterknotens in das rechte Blatt. Der rechte Knoten «leiht» sich also ein Element von weiter links.
   - Würden wir von weiter rechts leihen, dann wandert das Element mit dem kleinsten Schlüssel aus dem Nachbarblatt.

   Variante:

   Da die Nachbarseite sowieso in den Hauptspeicher geladen werden muss, kann man die Situation ausnützen und die Elemente gleichmässig aufteilen:

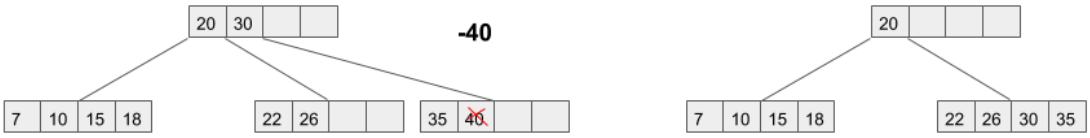
   $m + 1 + n - 1 = m + n$ neu links: $(m + n) / 2$
   neu rechts: $(m + n - 1) / 2$

   Beispiel:



2. Zwei Seiten *zusammenlegen*, falls die Nachbarseite nur *n* Elemente hat. Mit dem dazwischenliegenden Element im Vaterknoten erhält man exakt $n + 1 + n - 1 = 2n$ Elemente, was einer vollen Seite entspricht.

   Beispiel:

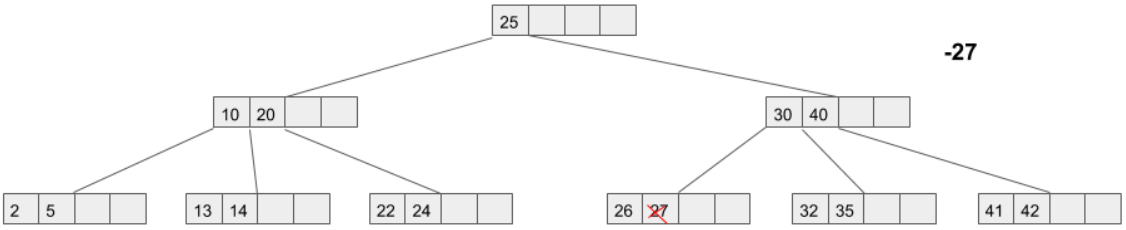


   - Das Blatt hat nun zu wenige Elemente und verletzt somit die B-Baum Definition.
   - Kein Nachbarblatt ist mehr als zur Hälfte voll, d.h. “Ausleihen” ist nicht möglich.
   - Das Blatt wird deshalb mit einem Nachbarblatt verschmolzen.
   - Dabei wandert auch ein Element des Vaterknotens (hier die 30) ins neue Blatt.

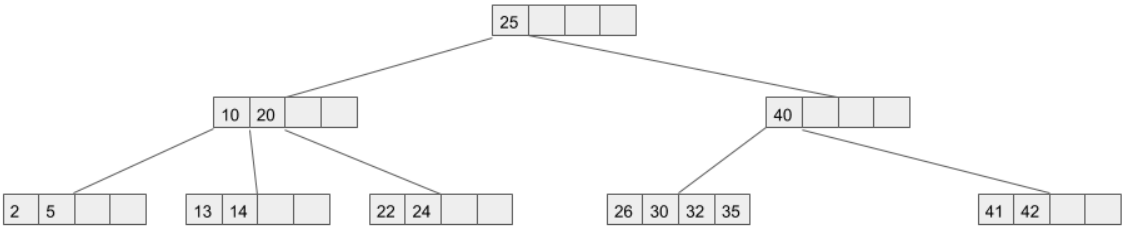
   Falls der Vaterknoten damit weniger als *n* Elemente enthält (und nicht die Wurzel ist), muss rekursiv ausgeglichen werden. Dabei ist sowohl Ausleihen als auch Zusammenlegen möglich. Dies kann sich bis zur Wurzel fortsetzen. Hat die Wurzel kein Element mehr, dann wird sie gelöscht. Die Höhe verringert sich dabei.

Beispiel für Ausgleichen bis zur Wurzel:

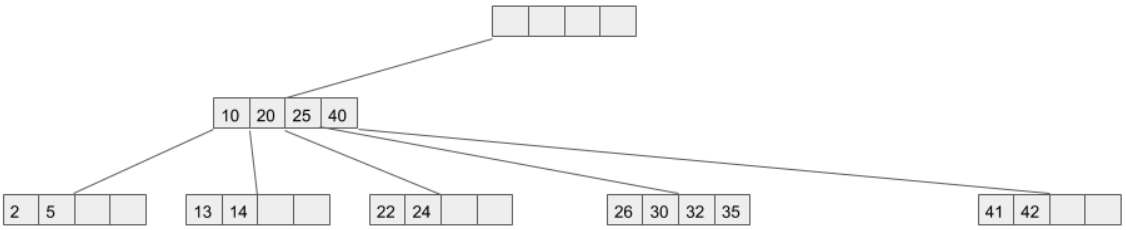
1. Löschen von 27 im Blattknoten



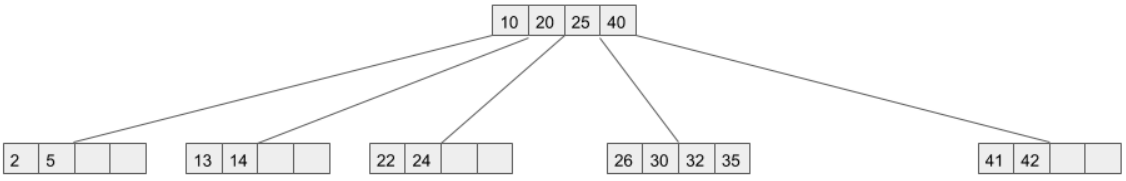
2. Verschmelzen von zwei Nachbarblättern



3. Verschmelzen von zwei inneren Knoten. Dabei werden auch Teilbäume / Blätter umgehängt.



4. Entfernen der leeren Wurzel. Die Höhe des Baumes nimmt dabei um 1 ab.



Variante:

Zusammenlegen von drei benachbarten Seiten und Bildung von zwei neuen Seiten daraus. Die entstehenden Seiten sind dann nicht ganz gefüllt. Dies hat den Vorteil, dass beim nächsten Löschen aus einem der drei Blätter kein Ausgleichen erforderlich ist.

Beispiel: