

4. Baum-Strukturen

4.1. Motivation

Listen, egal ob mit Array oder verlinkten Objekten implementiert, sind linear aufgebaute Strukturen. Operationen darauf, die sich nicht mit konstantem Aufwand erledigen lassen, haben deswegen immer einen Aufwand von $O(n)$. Das entspricht auch der Weglänge zwischen dem ersten und dem letzten Element.

Hat jeder Knoten nicht nur einen Nachfolger, wie bei Listen, sondern mehrere, spricht man von *Bäumen*. In solchen Strukturen kann man die Entfernung zwischen zwei Knoten auf $O(\log n)$ begrenzen. Es besteht daher eine gewisse Hoffnung, die für *Collections* benötigten Operationen auch mit geringerem Aufwand als $O(n)$ umsetzen zu können.

Die wichtigste solche Datenstruktur sind *binäre Suchbäume*. Darin lassen sich tatsächlich Elemente, für die eine Ordnungseigenschaft definiert ist, effizient speichern und wiederfinden. Allerdings können solche Bäume *entarten* und dann zu Listen werden. Man benötigt daher zusätzliche *Ausgleichsmechanismen*, die den Baum wieder *in Form bringen*.

Möchte man sehr grosse Datenmengen speichern, hält man diese nicht im Hauptspeicher, sondern verwendet externe Speichermedien, wie Disks. Dort werden Daten in Blöcken abgelegt. Diese Blöcke werden immer als Ganzes geschrieben und gelesen. Um Zugriffszeiten zu reduzieren, lohnt es sich, die Knoten-Strukturen von Bäumen an diese Blockgrösse anzupassen. Das tun *B-Bäume*. Solche B-Bäume werden verwendet, um Datei-Verzeichnisse oder Indizes in Datenbanken zu speichern.

4.2. Lernziele

- Sie können die Konzepte von Bäumen benennen und erklären.
- Sie können Algorithmen zum Verarbeiten aller in einem Baum gespeicherten Elemente erklären und programmieren.
- Sie können Algorithmen zum *Einfügen*, *Suchen* und *Löschen* in *Binären Suchbäumen* erklären und kennen wichtige Eigenschaften, wie Voraussetzungen und asymptotische Komplexität.
- Sie können diese grundsätzlichen Algorithmen auf binären Suchbäumen in Java implementieren.
- Sie können die Ausgleichsmechanismen von AVL-Bäume erklären und an Beispielen demonstrieren.
- Sie können das *Einfügen* in einen AVL-Baum in Java programmieren.
- Sie können die definierenden Eigenschaften von *B-Bäumen* erklären.
- Sie können Algorithmen zum *Einfügen*, *Suchen* und *Löschen* in *B-Bäumen* erklären und an Beispielen demonstrieren.

4.3. Wegleitung zur Arbeit mit Leitprogramm und TreeEditor

Die Datenstruktur *Binäre Suchbäume* samt einigen Grundlagen zu *Bäumen* allgemein sollen Sie sich im Selbststudium mithilfe eines von der ETH Zürich herausgegebenen Leitprogramms erarbeiten. Zusätzlich haben wir Ihnen einen interaktiven *TreeEditor* als Programmier-Projekt vorbereitet, den Sie als Basis für ein paar Programmieraufgaben benutzen können.

Das Leitprogramm enthält eine Folge von Erklärungen und Aufgaben, die Sie in Ihrem eigenen Tempo bearbeiten können. Pro Kapitel gibt es ausserdem in separaten Dokumenten *Kapiteltests* und *Programmieraufgaben*.

Wenn Sie ein Kapitel des Leitprogramms durchgearbeitet haben, sollten Sie die Kapiteltests zu lösen versuchen. Gelingt Ihnen dies, machen Sie mit dem nächsten Kapitel weiter. Andernfalls repetieren Sie die relevanten Teile des zuletzt bearbeiteten Kapitels und wiederholen den Kapiteltest dann.

Lösungen der Aufgaben finden Sie jeweils am Ende des Leitprogramms und der Kapiteltests.

Sie finden die Unterlagen des Leitprogramms als PDF auf dem Datei-Server. Sollte Sie diese selbst ausdrucken wollen, überlegen Sie, ob Sie die Lösungen auch auf Papier benötigen.

Der *TreeEditor* erlaubt Ihnen etwas ausführlichere Programmieraufgaben zu lösen als die im Leitprogramm enthaltenen. Wir haben Ihnen ein Aufgabenblatt beigelegt. Zum Testen erzeugen Sie selbst interaktiv Bäume und kontrollieren, ob die so aussehen, wie sie sollen.

Empfohlener Zeitplan: s. Rückseite

Semester-Woche 4:

Themen *Bäume*, *Binärbäume* und *Binäre Suchbäume*:

- Kapitel 2 bis 4 im Leitprogramm
- Material für die Programmieraufgaben 3.3 (S. 28), 4.3 (S. 38), 4.5 (S. 38), 4.8 (S. 40) und 4.13 (S. 44) finden Sie als Programmier-Projekt auf dem File Server in *ch.fhnw.algd2.binsearchtrees.eth.zip*. Programmieren Sie in der Klasse *BinaryNode* bzw. *BinarySearchTree*. Die Klasse *AufgabeX_Y* sind als Applikation ausführbar und enthalten einfache Testprogramme. Für die Aufgaben 4.3, 4.8 und 4.13 finden Sie noch einige Unit Tests.
- Eine einfachere Alternative zur Aufgabe 4.13 (S. 44) im Leitprogramm stellt die *delete()* Methode im *TreeEditor* dar (s. separates Aufgabenblatt), da dieser keine Duplikate unterstützt.
- *Aufgaben für Binäre Suchbäume* mit dem *TreeEditor*

Semester-Woche 5:

Zu Beginn des Unterrichts besteht Zeit für Restklärungen zu den Kapiteln 2 bis 4 im Leitprogramm. Anschliessend Fortsetzung des Selbststudiums.

Themen *Bäume*, *Binärbäume* und *Binäre Suchbäume*:

- Kapitel 5 (bis einschl. 5.4) im Leitprogramm
- *Aufgaben für AVL-Bäume* mit dem *TreeEditor*

Teilen Sie sich die Zeit im Unterricht und zuhause selbst ein. Wir Dozierende sind während der Unterrichtszeit da, um Sie zu unterstützen und Ihre Fragen zu beantworten.

Semester-Woche 6:

Zum Beginn des Unterrichts können wir noch ca. 30 Minuten für Restklärungen verwenden. Anschliessend bieten wir Ihnen einen *Kurztest* an, den Sie in 15 Minuten unter Prüfungsbedingungen lösen können. Wir besprechen die Ergebnisse anschliessend. Es gibt keine Note, der Kurztest dient nur als Feedback für Sie selbst.

Danach wenden wir uns den *B-Bäumen* zu.

4.4. B-Bäume

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:

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

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. Wie viele Suchanfragen pro Sekunde lassen sich dann mit einem Baum der eben berechneten Höhe beantworten?

Bei sich (mit 7'200 U/min) drehenden Disks rechnet man damit, bis zu 120 Blöcke pro Sekunde lesen zu können.² Wie oft kann man damit im gerade betrachteten Baum suchen?

Zum Vergleich: Für die Zugriffszeit im Hauptspeicher rechnen wir mit 10ns, also 10^8 Zugriffen pro Sekunde. Wie oft kann dann im Baum gesucht werden?

Ziel

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.

Idee

Ein Baumknoten sollte gerade einem Diskblock entsprechen. Statt einem Binärbaum verwenden wir einen Mehrwegbaum.

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

² siehe beispielsweise http://de.wikipedia.org/wiki/Input/Output_operations_Per_Second

Hat in einem solchen Baum jeder Knoten s Nachfolger, ist die Höhe h bei N gespeicherten Elementen:

$$h = \lceil \log_s(N + 1) \rceil.$$

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];
}
```

Veranschlagen wir jeweils 4 Byte pro Referenz auf einen (Nachfolge-)Knoten, sowie 4 Byte als Referenz auf die zu einem Schlüssel gehörenden Daten. Wie viele 64-Byte Schlüssel passen dann in einen Knoten mit 4096 Bytes (häufige Block-Grösse) und wie hoch wird der Baum dann für $8 \cdot 10^6$ Elemente?

Auch wenn man die Blöcke füllt, kann der Baum zu einer Liste solcher Blöcke entarten. Das führt im schlimmsten Fall für eine Anfrage zu vielen Plattenzugriffen:

Der Mehrwegbaum muss daher auch ausgeglichen sein. Diese Eigenschaften haben *B-Bäume*.

Definition: B-Baum [Bayer / McCreight 1970]³

Achtung: Für B-Bäume wird der Begriff Ordnung anders definiert, als für allgemeine Bäume!⁴

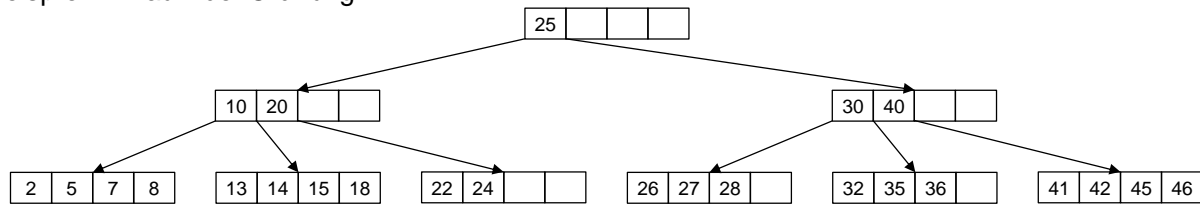
Ein B-Baum der Ordnung n ist ein Mehrwugsuchbaum vom Grad $2n + 1$ mit folgenden Struktureigenschaften:

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.

³ 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.

⁴ 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.

Beispiel: B-Baum der Ordnung 2



Wie viele Elemente können noch eingefügt werden, ohne dass die Höhe wächst?

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

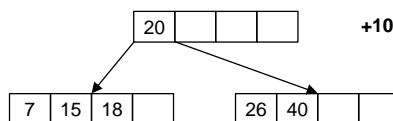
- Man weiss nicht wie gross die Disk-Blöcke sind (Absicht: Java soll plattformunabhängig sein)
- Keine Kontrolle über Caching
- Daten müssen serialisiert werden

Deswegen betrachten wir die weiteren Operationen auf B-Bäumen ausschliesslich auf der Konzept-Ebene.

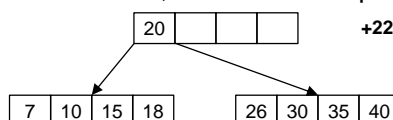
4.4.2. Einfügen im B-Baum

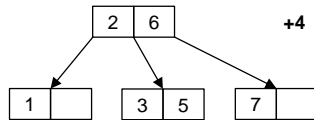
Neue Elemente werden immer in Blätter eingefügt. Dabei können folgende Situationen entstehen:

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



- b) Seite ist voll, man muss sie splitten



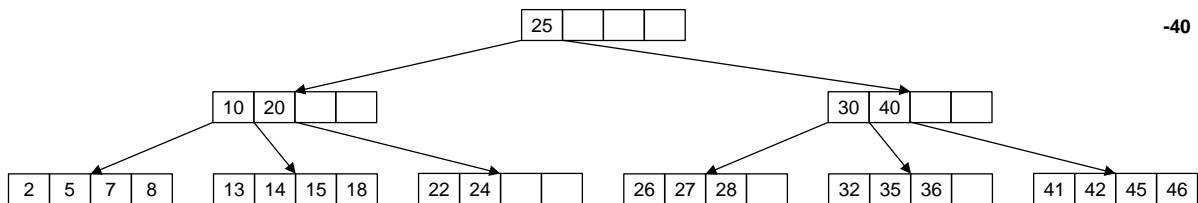


4.4.3. Löschen im B-Baum

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

- Element befindet sich in einem Blatt
- Element befindet sich in einem inneren Knoten

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

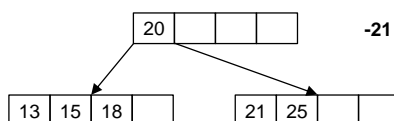


Es müssen folgende Fälle beim Löschen aus einem Blattknoten unterschieden werden:

- Blattseite hat $m > n$ Elemente
- Blattseite hat n Elemente. Das Entfernen eines Elementes würde die Bedingung $m \geq n$ verletzen (ausser beim Wurzelknoten).
⇒ Ausgleichen.

Ausgleichen:

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



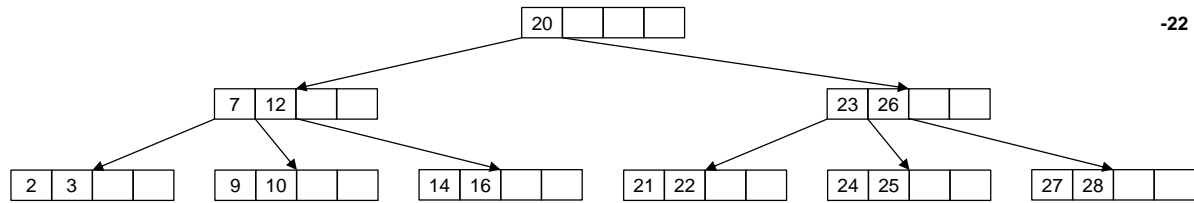
Variante:

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

$$m + 1 + n - 1 = m + n \quad \Rightarrow \quad \begin{array}{ll} \text{neu links:} & (m + n) / 2 \\ \text{neu rechts:} & (m + n - 1) / 2 \end{array}$$

2. Falls Nachbarseite nur n Elemente hat: Beide Seiten zusammenlegen. Mit dem dazwischenliegenden Element im Vaterknoten erhält man $n + 1 + n - 1 = 2n$ Elemente, was einer vollen Seite entspricht.

Falls der Vaterknoten damit weniger als n Elemente enthält, muss rekursiv ausgeglichen werden. Dies kann sich bis zur Wurzel fortsetzen. Hat die Wurzel kein Element mehr, dann wird sie gelöscht. Die Höhe verringert sich dabei.



Variante:

Zusammenlegen von drei benachbarten Seiten und Bildung von zwei neuen Seiten daraus. Die entstehenden Seiten sind dann nicht ganz gefüllt.