FAU, Informatik 2, AUD-Team
aud@i2.cs.fau.de

11. Übung

Abgabe bis 30.01.2017, 10:00 Uhr

Einzelaufgabe 11.1: TreeDetector — 9 EP

In der Vorlesung haben Sie verschiedene Arten von Binärbäumen kennengelernt. Unter anderem Halden und binäre Suchbäume. Ziel dieser Aufgabe ist es, für einen gegebenen Binärbaum festzustellen, ob dieser ein binärer Suchbaum oder eine Halde (oder nichts von beidem) ist.
Ein Binärbaum setzt sich aus verschiedenen Objekten der Klasse `Node` zusammen. Ein so gebildeter Baum beinhaltet die allgemeinen Eigenschaften eines Binärbaums. Die Klasse `TreeGenerator` stellt ihnen eine beispielhafte Halde oder binären Suchbaum zur Verfügung.
Die Erkennung von binären Suchbäumen und Halden wird von der Klasse `TreeDetector` durchgeführt, die im Verlauf dieser Aufgabe Schritt für Schritt erweitert wird. Laden Sie sich die Dateien `TreeDetector.java`, `Node.java` und `TreeGenerator.java` herunter und ergänzen Sie sämtliche Methoden der Klasse `TreeDetector`.

a) Implementieren Sie die **rekursive** Methode `isHeap()`, die `true` zurückgibt, falls der als Parameter übergebene Binärbaum eine Max-Halde ist. Beachten Sie, dass ein Heap in dieser Aufgabe nicht zwangsläufig links-vollständig sein muss.

b) Ergänzen Sie die nun die **rekursive** Methode `isBinarySearchTree()`, die feststellt, ob der Baum ein binärer Suchbaum ist. Führen Sie hierzu eine neue Methode ein, die als Parameter den maximalen und minimalen Schlüsselwert, den ein Teilbaum enthalten darf, empfängt.

c) Passen Sie ihre Implementierungen so an, dass auch ein leerer Baum (`null`), sowie ein Baum mit nur einem Knoten als Heap oder binärer Suchbaum erkannt wird.

Bitte geben Sie Ihre Lösung als `TreeDetector.java` ab.

Einzelaufgabe 11.2: MergeSort — 15 EP

In dieser Aufgabe sollen Sie den Mergesort-Algorithmus implementieren, der eine einfach verkettete Liste von Objekten aufsteigend sortiert. Laden Sie sich hierfür die Dateien `List.java`, `Element.java` und `MergeSort.java` herunter. Die Klasse `List` repräsentiert die zu sortierende Liste und beinhaltet ein Objekt vom Typ `Element`, welches den Start der verketteten Liste darstellt. Über das Attribute `next` der Klasse `Element` werden nachfolgende Elemente der Liste referenziert. Die Methode `size()` der Klasse `List` gibt die Größe der Liste zurück.

a) Implementieren Sie die Methode `merge()`. Dabei sollen zwei Listen miteinander verschmolzen werden. Sie können davon ausgehen, dass beide Listen bereits sortiert sind. Das Ergebnis soll wiederum eine Liste sein, die alle Elemente beider Listen enthält und ebenfalls sortiert ist. Sie sollen kein neues Listen Objekt erzeugen. Verschmelzen Sie stattdessen das rechte `List` Objekt in das linke `List` Objekt. Gleich große Elemente die vorher in einer Liste waren, sollen nach dem Verschmelzen in der selben Reihenfolge auftauchen. Gleich grosse Elemente der linken Liste sollen vor denen in der Rechten Liste einsortiert werden.

b) Implementieren Sie nun die **rekursive** Methode `doSort()`. Die Methode soll einen Teil der Liste sortieren. Diese Teilliste beginnt beim Startelement der Liste und ist `n` Elemente lang. Beim Sortieren dürfen Sie keine neuen `Element` Objekte erzeugen

c) Implementieren Sie als letztes die Methode `sort()`. Diese Methode wird aufgerufen um ein `List` Objekt zu sortieren. Die Methode muss die passenden Parameter für die in der vorherigen Teilaufgabe implementierten Methode erstellen und die übergebene Liste so verändern, dass sie das Ergebnis repräsentiert.

Bitte geben Sie Ihre Lösung als `MergeSort.java` im EST ab.

Einzelaufgabe 11.3: QuickSort **15 EP**

In der Vorlesung haben Sie den Quicksort-Algorithmus kennengelernt. Laden Sie sich die Datei `QuickSort.java` herunter. Die Implementierung erfolgt schrittweise in den zu vervollständigenden Methoden:

a) Implementieren Sie in der Methode `swap()` das Vertauschen zweier Elemente eines Arrays

b) Implementieren Sie nun die Methode `choosePivot()`. Diese Methode dient dazu, die Wahl des Pivot-Elements zu optimieren. Die Methode betrachtet dazu die Elemente am Anfang, am Ende und in der Mitte des gegebenen Bereiches und bestimmt das **mittlere** Element aus diesen. Der **Index** dieses Elements wird zurückgegeben. Beispielsweise würde für das Array `{2, 5, 1, -5, 3}` der Index 0 zurückgegeben, da 2 das mittlere Element aus 2, 1 und 3 ist. Bei gerader Anzahl soll das Element mit den kleineren Index als in der Mitte betrachtet werden. Wenn von den drei Elementen zwei oder drei gleich gross sind, soll das mit dem kleineren Index zurückgegeben werden.

c) Implementieren Sie jetzt die Methode `partition()`. Diese Funktion gruppiert den angegebenen Bereich des Arrays so, dass zunächst alle Elemente vorkommen, die kleiner oder gleich dem Pivot-Element sind. Dann folgt das Pivot-Element selbst sowie alle Elemente, die größer sind. Beachten Sie, dass der **Index** des zu verwendenden Pivot-Elements als Parameter übergeben wird. Die Methode gibt den neuen Index des Pivot-Elements zurück.

d) Implementieren Sie alls letztes unter Verwendung der schon implementierten Methoden die Methode `sort`.
Hinweis: Vermutlich wird es nötig sein, eine rekursive Hilfsmethode zu verwenden, um weitere Aufrufparameter für die Rekursion verwenden zu können.

Bitte geben Sie Ihre Lösung als `QuickSort.java` im EST ab.

Gruppenaufgabe 11.4: BTree **21 GP**

In dieser Aufgabe sollen Sie eine weitere besondere Art von Bäumen, nämlich `B-Baum`, implementieren. Ein `B-Baum` mit Grad g hat folgende Eigenschaften:

a) Jeder Knoten außer der Wurzel enthält mindestens g und maximal $2g$ Schlüsselwerte.

b) Der Wurzelknoten enthält mindestens 1 und maximal $2g$ Schlüsselwerte.

c) Die Schlüsselwerte innerhalb eines Knotens sind aufsteigend sortiert.

d) Jeder innere Knoten mit k Schlüsseleinträgen hat genau $k + 1$ Kinder.

e) Alle Blätter des Baumes liegen in der gleichen Höhe.

f) `BTree`-Sucheigenschaft (ähnlich zu binären Suchbäumen):
Seien $s_1, s_2, ..., s_k$ mit $g \leq k \leq 2g$ die Schlüssel eines Knotens x, dann gilt:

- alle Schlüssel des ersten Sohnes von x sind kleiner als s_1,
- alle Schlüssel des i-ten Sohnes von x $(1 < i < k + 1)$ sind größer als s_{i-1} und kleiner als s_i, und
- alle Schlüssel des $(k + 1)$-ten Sohnes sind größer als s_k.

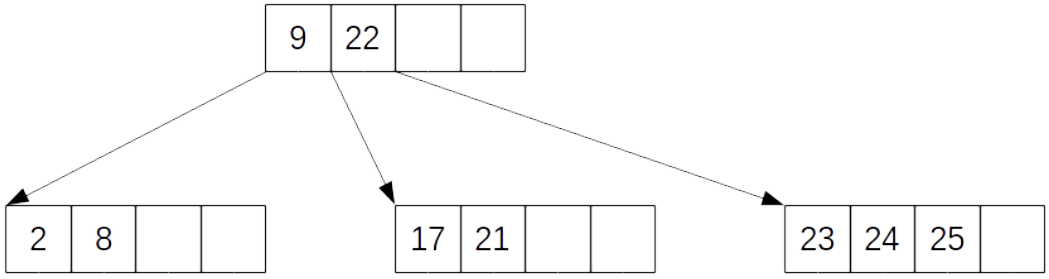

Abbildung 1: Beispiel eines $g = 2$ B-Baums (Einfügereihenfolge: 2,8,9,17,23,21,22,24,25)

Bei der Implementierung von Datenbanken oder Dateisystemen werden B-Bäume eingesetzt. Gerne können Sie im Internet oder in der Literatur weitere Informationen zu B-Bäumen einholen. Bedenken Sie jedoch: Eigenschaften von B-Bäumen in verschiedenen Publikationen weichen zum Teil von oben genannten Eigenschaften ab. Es gelten die oben genannten Eigenschaften.
Laden Sie sich die Dateien `BTree.java`, `BTreeNode.java` und `OverflowNode.java` herunter und vervollständigen Sie die Klassen `BTree` und `BTreeNode`, die den gesamten Baum bzw. seinen Knoten beschreiben. Gehen Sie dazu schrittweise die folgenden Aufgaben durch. Nutzen Sie für ihre Lösungen auch die Klasse `OverflowNode`.

a) Implementieren Sie die Suche nach vorhandenen Schlüsseln im Baum. Dazu müssen die Methoden `hasKey()` der Klasse `BTree` und `hasKey()` der Klasse `BTreeNode` erweitert werden. Die Methode `hasKey()` der Klasse `BTree` stößt die Suche im Wurzelknoten an. Die Methode `hasKey()` der Klasse `BTreeNode` geht die Baumstruktur rekursiv gemäß der oben definierten Sucheigenschaft durch.

b) Zum Einfügen von neuen Schlüsselwerten in den Baum dienen die beiden Methoden `insert()` der Klassen `BTree` und `BTreeNode`. Die Methode des Baums übergibt erneut den Wert an den Wurzelknoten, der das korrekte Einfügen **rekursiv** vornimmt. Dabei wird das passende Blatt bestimmt, an dem der neue Schlüssel eingefügt werden muss. Falls das Einfügen zu einem Überlauf des Knotens führt, muss mithilfe der Methode `split()`, die in der nächsten Teilaufgabe beschrieben ist, der Knoten aufgespaltet werden. Der Überlauf (`OverflowNode`) propagiert sich anschließend auf die überliegende Ebene. Beachten Sie, dass der Überlauf sich bis zur Wurzel propagieren kann und dabei eine neue Wurzel entstehen kann.

c) Falls das Einfügen eines neuen Werts einen Knoten zum Überlauf bringt, muss der Knoten aufgespaltet werden. Dafür ist die Methode `split()` der Klasse `BTreeNode` verantwortlich. Ein solcher Überlauf tritt ein, wenn der Knoten nach dem Einfügen exakt $2g + 1$ Werte enthält. Die rechten g Werte (Schlüsselwerte und ggf. eine gleiche Anzahl zugehöriger Kindknoten) sowie das mittlere Element (Schlüsselwert) werden in einen vorläufigen Überlaufknoten (Klasse `OverflowNode` nutzen!) verpackt, der von der Methode zurückgegeben werden soll. Im Vaterknoten wird der Überlaufknoten dann wieder ausgepackt: Das mittlere Element wird eingefügt und die rechten g Werte werden als neuer Kindknoten eingehängt.

d) Oft ist es notwendig Datenstrukturen zu serialisieren bspw. um eine Speicherung in einer Textdatei zu ermöglichen. `BTree` und `BTreeNode` sollen jeweils um eine `toJson()` Methode erweitert werden, die den Baum in einen JSON String serialisiert. Der B-Baum in der oben stehenden Grafik soll bspw. in folgendem JSON String resultieren:

```
{
    node: [17, 23],
    children:
    [
        {
            node: [2, 8, 9]
        },
        {
            node: [21, 22]
        },
        {
            node: [24, 25]
        }
    ]
}
```

Hinweis: Sie müssen keine Zeilenümbrüche, Tabulatoren oder Leerzeichen beachten.

Bitte geben Sie Ihre Lösung als `BTree.java` und `BTreeNode.java` im EST ab.

39 EP + 21 GP = 60 Punkte