Praktische Informatik II SS 2009

Übungsblatt 5

Abgabe: 11.05.2009

Aufgabe 1 Binärbäume für jede Lebenslage

In diesem Aufgabenblatt soll eine objektorientierte Version der kanonischen Operationen auf Binärbäumen in Java entwickelt und getestet werden.

Aufgabe 1.1 Datenstrukturen, Methoden, Operationen (60%)

1. Der Baum soll – im Gegensatz zur Vorlesung – mit Hilfe *einer* Klasse `TreeNode` realisiert werden, ohne dass eine Unterscheidung zwischen Bäumen und Baumknoten erforderlich wird. Jede Instanz hat einen Vorgänger-Knoten `parent`, der für die Wurzel des Baums mit Null belegt ist. Jeder Knoten hat zwei Nachfolger `left, right`, die für Blätter durch Null-Referenzen belegt sind. Jeder Knoten kann mit einer Referenz auf Nutzdaten ausgestattet werden. Nutzdaten sollen dabei eine Ableitung des Interfaces

   ```
   interface TreeData extends Comparable<TreeData>, Cloneable {
       abstract public TreeData clone();
       abstract public Comparable getData();
   }
   ```

   sein. Erläutert im beschreibenden Text zu Eurer Lösung, warum dieses Interface sinnvoll ist. Studiert hierzu die Java-Bibliotheksreferenz in Bezug auf den `clone()`-Aufruf.

   Es dürfen nur Knoten im Baum existieren, deren Nutzdatenreferenz nicht Null ist. **Ausnahme:** der leere Baum wird durch eine Wurzel mit null-Referenz auf die Nutzdaten gekennzeichnet.

   Als Anwendung definieren wir eine Klasse `MyTreeData`, welches das Interface `TreeData` implementiert und Strings als Nutzdaten enthält.

2. Als Konstruktoren benötigen wir

   (a) Einen Konstruktor, der einen leeren Baum erzeugt.

   (b) Einen Konstruktor, der einen nur aus der Wurzel mit zugehörigen Nutzdaten bestehenden Baum erzeugt.

   (c) Einen Copy-Konstruktor, der eine **tiefe** Kopie des Baums, einschliesslich Kopie der Nutzdaten erzeugt. Hierzu ist die clone()-Operation zu verwenden, da die Existenz eines Copy-Konstruktors in Java nicht bei jedem Objekt vorausgesetzt werden kann.

3. Als Methoden benötigen wir (die Signatur und Rückgabewerte sollen von Euch ausgearbeitet werden, wir geben hier nur die Methodennamen an):

(a) `isEmpty()`: Abfrage, ob der Baum leer ist.

(b) `isConsistent()`: Prüft, ob alle Verkettungen zwischen Vorgänger- und Nachfolgerknoten korrekt sind.

(c) `getHeight()`: berechnet die Höhe des Baums.

(d) `getNumNodes()`: zählt die Anzahl der Knoten im Baum.

(e) `insert()`: Fügt einen neuen Knoten mit Nutzdaten sortiert in den binären Baum ein. Zur Sicherstellung der Sortierung verwendet man die Methode `compareTo` des Interfaces `Comparable`, welches durch MyTreeData realisiert ist.

(f) `find()` Bekommt einen Eingangsparameter x vom Typ `TreeData` und sucht einen Knoten, der mit x belegt ist. Die Referenz auf diesen Knoten wird zurück gegeben.

(g) `delete()`: Löscht den Knoten aus dem Baum, auf dem die Operation angewendet wird. Typischerweise findet man den zu löschenden Knoten mittels `find()`.

(h) `toString()`: Gibt eine Textdarstellung des Baumes aus, bei der die Baumstruktur mit Hilfe einer gut gewählten Klammerung eindeutig erkennbar ist.

(i) `createDotFile()` erzeugt zu gegebenem Dateinamen (Extension .dot) eine Textausgabedatei, in welcher der Baum im GraphViz-Format (Programm dot, frei erhältlich) repräsentiert wird. Das Format ist nach folgendem Beispiel aufgebaut:

```
digraph MyTree {
  T [ label ="abc" ]
  T0 [ label ="aaa" ]
  T1 [ label ="def" ]
  T -> T0
  T -> T1
}
```

`label` markiert die Knoten und soll die Nutzdaten des Knotens tragen. `T, T0, ...` sind die internen Knotenbezeichnungen für dot, die dann in den Kantenspezifikationen `T -> T0` etc. wieder referenziert werden.

Aus einer solchen Ausgabedatei – z. B. Tree.dot – kann mittels dot ein formatiertes Bild des Baumes erzeugt werden:

```
# Erzeuge encapsulated Postscript output
dot -Tps -o Tree.eps Tree.dot
# Erzeuge pdf
epstopdf Tree.eps
```

(j) Die üblichen getter- und setter-Funktionen

Aufgabe 1.2 Und jetzt die Tests – einmal ganz anders (40%)

Erstellt eine Klasse mit main-Methode, in welcher ein Baum n0 mittels insert() aus den Labeln der beiliegenden Datei `data.txt` aufgebaut wird.

Anschliessend sucht das Element `‘‘shvckyhj8ssdkshv’’` und löscht es. Aus dem so veränderten n0 erstellt eine Kopie n1.

Die Prüfung der Methoden soll diesmal folgendermaßen erfolgen:

- Macht nach allen Veränderungen des Baumes klug gewählte Assertions, welche die Konsistenz der Veränderung prüfen – nützt dazu beispielsweise `isConsistent()` und `getNumNodes()` und `toString()` (letzteres zum Vergleichen der Baumkopie mit dem Original).
- Gebt drei dot-Visualisierungen aus:
  - Nach dem ersten Aufbauen von n0
  - Nach dem Löschen in n0
  - Nach Erstellung der Kopie n1 für dies Kopie.

  Vergleicht die visuelle Baumdarstellung und prüft, ob die Operationen für das Beispiel korrekt gelaufen sind.

Zur Abgabe gehören...

1. java-Datei mit den entwickelten Klassen
2. dot-Files für die 3 Bäume
3. pdf-Files für die 3 Bäume
4. Eure Beschreibung der Lösung in LaTeX