Prof. Dr. Uwe Schmidt

ptlwedel
PRIVATE BERUFSFACHSCHULE

16. August 2011

---

Aufgaben zur Klausur **Objektorientierte Programmierung** im SS 2011 (IA 252)

Zeit: 90 Minuten erlaubte Hilfsmittel: keine

Bitte tragen Sie Ihre Antworten und fertigen Lösungen ausschließlich an den gekennzeichneten Stellen in das Aufgabenblatt ein. Ist ihre Lösung wesentlich umfangreicher, so überprüfen Sie bitte nochmals Ihren Lösungsweg.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Verwenden Sie in den zu entwickelnden Java Programmteilen keine impliziten Konversionen zwischen einfachen Datentypen und kein Autoboxing und Autounboxing.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 10 Seiten.

---

**Aufgabe 1:**

Tipp: Bitte lesen Sie alle Programmteile einschließlich des Testprogramms und der Fragen sorgfältig durch, bevor Sie mit der Bearbeitung der Aufgabe beginnen. Kleine Skizzen auf den Rückseiten können ebenfalls hilfreich sein.

Für die Implementierung von Listen gibt es unterschiedliche Anforderungen und daher auch unterschiedliche Implementierungen.

In dieser Aufgabe geht es darum, eine Listenimplementierung zu entwickeln, die einen effizienten indizierten Zugriff aber auch das effiziente Einfügen von Elementen an beliebiger Stelle, auch am Anfang und am Ende, ermöglicht, und bei der eine Konkatenation von Listen in konstanter Zeit machbar ist.

Die Implementierung wird mit einem binären Baum arbeiten. Dieser Baum hat folgende Struktur: An den Blättern (und nur dort) sind die Elemente gespeichert, die inneren Knoten enthalten zwei Kindbäume und zusätzlich eine ganzzahlige Variable, die die Anzahl im Baum gespeicherter Elemente aufnimmt.

Die vollständige Klasse für den Baum:

```
abstract public class Tree {
  public static Tree mkLeaf(Object x) {
    return
      new Leaf(x);
  }
  public static Tree mkFork(Tree l, Tree r) {
    return
      new Fork(l,r);
  }
  abstract public int size();

  public Object at(int i) {
    if (i < 0 || i >= size())
      throw
        new IndexOutOfBoundsException();
    return
      at1(i);
  }
  public Tree insertInFrontOf(Object x, int i) {
    if (i < 0 || i > size())
      throw
        new IndexOutOfBoundsException();
    return
      insertInFrontOf1(x,i);

  }
  abstract Object at1(int i);
  abstract Tree insertInFrontOf1(Object x, int i);
}
```

Die Unterklassen von Tree für die Blätter und die inneren Knoten müssen also entwickelt werden, hierzu sind die Methoden size, at1 und insertInFrontOf zu implementieren.

Objekte der Klasse Leaf repräsentieren einelementige Listen. Die Länge und der indizierte Zugriff für einelementige Listen ist sehr einfach zu realisieren, für das Einfügen gibt es (Achtung) zwei Möglichkeiten, vor dem Element und hinter dem Element.

Vervollständigen Sie die Klasse Leaf:

```
class Leaf extends Tree {
   private final Object v;
   Leaf(Object v) {
      this.v = v;
   }
   public int size() {

      ..............................
   }
   public Object at1(int i) {
      assert (i == 0);

      ..............................

      ..............................
   }
   public Tree insertInFrontOf1(Object x, int i) {
      assert (i == 0 || i == 1);
      if (i == 0) {

         ..............................

         ..............................
      } else {

         ..............................

         ..............................
      }
   }
   public String toString() {
      return v.toString();
   }
}
```

Für mehrelementige Listen wird die Fork-Klasse verwendet. Die Längenmethode ist wieder sehr einfach zu realisieren, für den Zufriff und das Einfügen muss man jeweils den richtigen Teilbaum auswählen. Beachten Sie bitte das final Attribut für die Datenfelder dieser Klasse.

Vervollständigen Sie die Klasse Fork:

```
class Fork extends Tree {
   private final Tree l;
   private final Tree r;
   private final int size;

   Fork(Tree l, Tree r) {

      ..............................

      ..............................

      ..............................
   }

   public int size() {

      ..............................

      ..............................
   }
   public Object at1(int i) {
      assert (0 <= i && i < size());

      ..............................

      ..............................

      ..............................

      ..............................

      ..............................

      ..............................

      ..............................
   }
```

```
  public Tree insertInFrontOf1(Object x, int i) {
    assert (0 <= i && i <= size());

    ..............................

    ..............................

    ..............................

    ..............................

    ..............................

    ..............................

    ..............................
  }
  public String toString() {
    return
      " (" + l.toString() + " . " + r.toString() + ") ";
  }
}
```

Die Tree-Klasse und deren Unterklassen bilden den Kern der Implementierung, besitzen aber noch keine für Listen geeignete Schnittstelle. Außerdem kann die leere Liste nicht durch einen Baum repräsentiert werden.

Eine geeignete abstrakte Klasse für Listen mit den wesentlichen Methoden wird in der Klasse List festgelegt. Führen Sie die Operationen append und prepend auf das Einfügen eines Elements in eine Liste zurück.

Vervollständigen Sie die Klasse List:

```
abstract public class List {
  static final List empty = new EmptyList();

  public static List mkEmpty() {
    return
      empty;
  }

  public static List mkOne(Object x) {
    return
      new NonEmptyList(x);
  }

  public boolean isEmpty() {

    ..............................

    ..............................
  }

  public List append(Object x) {

    ..............................

    ..............................
  }

  public List prepend(Object x) {

    ..............................

      ..............................
  }

  abstract public int length();
  abstract public Object at(int i);
  abstract public List insert(Object x, int i);
  abstract public List concat(List l2);

  abstract protected List flipConcat(NonEmptyList l1);
}
```

Wie aus der Definition von List hervorgeht, werden zwei Unterklassen EmptyList und NonEmptyList benötigt, in denen die fehlenden Methoden zu implementieren sind. Vervollständigen Sie beide Unterklassen so, dass bei illegalen indizierten Zugriffen eine IndexOutOfBoundsException ausgelößt wird.

Die Konkatenation ist etwas trickreich, da es insgesammt vier Fälle zu unterscheiden gilt, die hier mit zweimaligem dynamischen Binden und einer Hilfsmethode realisiert werden.

Vervollständigen Sie die Klasse EmptyList:

```
class EmptyList extends List {
   public int length() {

      ..............................
   }
   public Object at(int i) {

      ..............................

      ..............................
   }
   public List insert(Object x, int i) {

      ..............................

      ..............................

      ..............................

      ..............................

      ..............................
   }
   public List concat(List l2) {

      ..............................
   }
   protected List flipConcat(NonEmptyList l1) {
      return l1;
   }
   public String toString() {
      return "[]";
   }
}
```

Vervollständigen Sie die Klasse NonEmptyList:

```
class NonEmptyList extends List {
   final private Tree t;
   NonEmptyList(Object x) {
      t = Tree.mkLeaf(x);
   }
   NonEmptyList(Tree t) {
      this.t = t;
   }
   public int length() {

      ..............................

      ..............................
   }
   public Object at(int i) {

      ..............................

      ..............................
   }
   public List insert(Object x, int i) {

      ..............................

      ..............................
   }
   public List concat(List l2) {
      if ( l2.isEmpty() ) {

         ..............................
      } else {

         ..............................

         ..............................
      }
   }
   protected List flipConcat(NonEmptyList l1) {

      ..............................

      ..............................
   }
```

```
  public String toString() {
    return
      "[" + t.toString() + "]";
  }
}
```

Ein kleines Testprogramm liefert die Klasse Test:

```
public class Test {
  public static void main(String [] args) {
    List l1 = List.mkEmpty().append("123").prepend("987");
    List l2 = List.mkOne("def").prepend("xyz").append("abc");
    List l3 = l2.concat(l1);
    List l4 = l2.insert("+++",2);
    List l5 = l2.insert("---",3);

    System.out.println("l1 = " + l1);
    System.out.println("l2 = " + l2);
    System.out.println("l3 = " + l3);
    System.out.println("l4 = " + l4);
    System.out.println("l5 = " + l5);
  }
}
```

Welche 5 Zeilen gibt dieses Programm aus:

1) .....

2) .....

3) .....

4) .....

5) .....

Software-technische Fragen zu den Programmteilen:

1. Gibt es einen Software-technischen Nutzen für die Einführung der beiden statischen Methoden in der Klasse *Tree*?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

2. Würde eine generische Variante für die Klasse *Tree* die Flexibilität dieser Klasse einschränken?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

3. Gibt es einen Software-technischen Nutzen dafür, dass in der Klasse *Tree* sowohl der Algorithmus für das Indizieren als auch der für das Einfügen auf die Methoden *at* und *at1* und *insertInFrontOf* und *insertInFrontOf1* aufgeteilt wird?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

4. Können in dieser Implementierung Teillisten in verschiedenen Listen gemeinsam genutzt werden, ohne dass Seiteneffekte die Semantik verändern?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

5. Ist es ein Fehler, dass sich in der Klasse *Fork* die Zusicherungen in *at1* und *insertInFrontOf1* unterscheiden?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

6. Ist die Implementierung von *isEmpty* in der Klasse List effizient?

   ja ☐ nein ☐

   Begründung:

   . . . . . .