Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

6. Februar 2014

Aufgaben zur Klausur **Objektorientierte Programmierung** im WS 2013/14 (B_ECom 18b,B_Inf v211 od. 18b, B_TInf v211 od. 18b, B_MInf v211 od. 18b, B_WInf v211 od. 18b)

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.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

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

Tipp: Bei der Entwicklung der Lösung können kleine Skizzen manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 10 Seiten.

Aufgabe 1:

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

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

In dieser Aufgabe geht es darum, eine Listenimplementierung zu entwickeln, die einen effizienten indizierten Zugriff aber auch das effiziente Einfügen und Löschen 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. Für den leeren Baum gibt es ein Spezialobjekt.

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);
    }
    public static Tree mkEmpty() {
        return
            Empty.emptyTree;
    }
    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);

    }
    public Tree removeAt(int i) {
        if (i < 0 || i >= size())
            throw
                new IndexOutOfBoundsException();
        return
            removeAt1(i);
    }

    abstract public boolean isEmpty();
    abstract public int size();
    abstract Object at1(int i);
    abstract Tree insertInFrontOf1(Object x, int i);
    abstract Tree removeAt1(int i);
    abstract public Tree concat(Tree t2);
}
```

Die Unterklassen von Tree für die Blätter und die inneren Knoten und den leeren Baum müssen noch entwickelt werden, hierzu sind die Methoden *isEmpty, size, at1, insertInFrontOf, removeAt* und *concat* zu implementieren.

Für die Repräsentation des leeren Baums wird ein Object aus der Klasse *Empty* benötigt. Für *Empty* gibt es eine Konsistenzbedingung: Dieses Objekt darf nur für den Wurzelknoten eines leeren Baums genutzt werden, er darf nie als Teilbaum eines anderen Baums genutzt werden.

Vervollständigen Sie die Klasse Empty:

```
class Empty extends Tree {
    static final Tree emptyTree = new Empty();

    private Empty() {}

    public boolean isEmpty() {

        ..............................
    }
    public int size() {

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

        ..............................
    }
    public Tree insertInFrontOf1(Object x, int i) {
        assert( ..............................);

        ..............................
    }
    public Tree removeAt1(int i) {
        assert( ..............................);

        ..............................
    }
    public Tree concat(Tree t2) {

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

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 boolean isEmpty() {

        . . . . . .
    }
    public int size() {

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

        . . . . . .

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

        . . . . . .

        . . . . . .

        . . . . . .

        . . . . . .

        . . . . . .

        . . . . . .
    }
```

```
    public Tree removeAt1(int i) {
        assert( . . . . . );
        . . . . . .
        . . . . . .
    }
    public Tree concat(Tree t2) {
        assert( . . . . . );
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
        . . . . . .
    }
    public String toString() {
        return v.toString();
    }
}
```

Für mehrelementige Listen wird die Fork-Klasse verwendet. Die Längenmethode ist wieder sehr einfach zu realisieren, für den Zugriff und das Einfügen und Löschen 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 boolean isEmpty() {

        . . . . . .
    }
    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 Tree removeAt1(int i) {

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .
}
public Tree concat(Tree t2) {

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .

    . . . . . .
}
```

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

Software-technische Fragen zu den Programmteilen:

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

   ja ☐ nein ☐

   Begründung:

   .....

2. Gibt es einen Software-technischen Nutzen für die gewählten Sichtbarkeitsattribute von *at1, insertInFrontOf1* und *removeAt1*

   ja ☐ nein ☐

   Begründung:

   .....

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

   ja ☐ nein ☐

   Begründung:

   .....

4. 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:

   .....

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

   ja ☐ nein ☐

   Begründung:

   .....

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

   ja ☐ nein ☐

   Begründung:

   . . . . . .

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

   ja ☐ nein ☐

   Begründung:

   . . . . . .

8. Kann die Methode *concat* geschickter implementiert werden?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

   . . . . . .

9. Sind die Sichtbarkeitsattribute für den Konstruktor *Empty* sinnvoll gewählt?

   ja ☐ nein ☐

   Begründung:

   . . . . . .

   . . . . . .

10. Kann die Methode *isEmpty* geschickter implementiert werden?

    ja ☐ nein ☐

    Begründung:

    . . . . . .

    . . . . . .