

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im WS 2007/08 (WI h103, II h105, MI h353)

Zeit: 150 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.

Viel Erfolg !

Diese Klausur besteht einschließlich dieses Deckblattes aus 20 Seiten

Aufgabe 1:

Rot-Schwarz-Bäume sind binäre Suchbäume, die gewisse Ausgewogenheitskriterien erfüllen müssen. Diese Bedingungen werden mit Hilfe Invarianten formuliert. Eine Bedingung ist die, dass keine roten Knoten einen roten Kindknoten besitzen.

Die Datenstruktur-Definition für einen als Rot-Schwarz-Baum realisierten Mengendatentyp habe folgendes Aussehen:

```
typedef int Element;

typedef struct Node * Set;

struct Node
{
    enum { RED, BLACK } color;
    Element info;
    Set l;
    Set r;
};

static struct Node finalNode = {BLACK, 0, 0, 0};

#define mkEmptySet() (&finalNode)
#define isEmptySet(s) ((s) == &finalNode)

extern unsigned int maxPathLength(Set s);
extern unsigned int minPathLength(Set s);

#define isBlackNode(s) ((s)->color == BLACK)
#define isRedNode(s) (! isBlackNode(s))

extern int hasRedChild(Set s);
extern int invNoRedNodeHasRedChild(Set s);
```

In diesem Codefragment sind einige Makros und einige Funktionen deklariert. Die Bedeutung der Funktionen geht aus dem Namen hervor. Benutzen Sie bitte diese Makros und Funktionen zur Formulierung Ihrer Lösung.

Man erkennt an den Makros *isEmptySet* und *mkEmptySet*, dass in dieser Implementierung der leere Baum durch einen Zeiger auf einen speziellen Knoten repräsentiert wird, nicht durch den 0-Zeiger.

Es soll als erstes die Hilfsfunktion *hasRedChild* entwickelt werden. Dieses Prädikat soll überprüfen, ob für einen beliebigen Baum ein möglicher Wurzelknoten einen roten Kindknoten besitzt.

Die Funktion *hasRedChild*:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Mit dieser Hilfsfunktion kann der Teil der Invariante für Rot-Schwarz-Bäume formuliert werden, der überprüft, dass an keiner Stelle in einem Baum ein roter Knoten einen roten Kindknoten besitzt. Dieses überprüft die Funktion *invNoRedNodeHasRedChild*

Die Funktion *invNoRedNodeHasRedChild*:

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Aufgabe 2:

Gegeben seien die folgenden Typdefinitionen und Variablendeklarationen:

```
#include <stdlib.h>

typedef struct X * Tree;

typedef Tree (*Tf)(Tree);

struct X {
    char * k;
    struct D * a;
    Tree cs[2];
};

struct D {
    Tf f;
    char * n;
    unsigned int i;
};

Tree root;

long int li = 2;
```

Bestimmen Sie für die folgenden Ausdrücke den Typ gemäß ANSI-C. Vorsicht: Sollten Ausdrücke vorkommen, die zur Übersetzungszeit Fehlermeldungen erzeugen, so kennzeichnen Sie diese mit dem Wort **FEHLER**

- root→k
- root→k[1]
- root→cs[li]
- *((*root).k)
- (* (root→cs))→a→f
- *(root→cs + 1)
- root→a→i + li
- (root→a→f)(root)
- root→a→i++
- root ? root→cs : 0
- root ? root→cs[0] : root
- sizeof** *root
- sizeof** (**struct X**)

Des weiteren seien folgende Funktionen definiert:

```
#include <stdlib.h>
#include <assert.h>

#include "types.c"

static unsigned int cnt = 0;

Tree mkTree(char * k,
            struct D * a,
            Tree t1,
            Tree t2) {
    Tree res = malloc(sizeof *res);
    if (! res) exit(1);
    res->k = k;
    res->a = a;
    res->cs[0] = t1;
    res->cs[1] = t2;
    return res;
}

struct D * mkD(Tf f, char * n) {
    struct D * res = malloc(sizeof *res);
    if (! res) exit(1);
    res->f = f;
    res->n = n;
    res->i = cnt++;
    return res;
}

Tree left(Tree t) {
    assert(t != 0);
    return t->cs[0];
}

Tree right(Tree t) {
    assert(t != 0);
    return t->cs[1];
}
```

Welche Ausgaben erzeugt das folgende Programm? Es können in diesem Programm Ausdrücke vorkommen, die nicht definiert sind. Kennzeichnen Sie diese mit UNDEF in der Lösungszeile. Geben Sie alle Ausgaben an, auch wenn vorher undefinierte Auswertungen gemacht wurden.

```
#include <stdio.h>

#include "mk.c"

int main(void) {
    Tree t1, t2;
    struct D *f1, *f2;
    f1 = mkD(left,"links");
    f2 = mkD(right,"rechts");

    t1 = mkTree("zwiebel", f1, 0, 0);
    t2 = mkTree("moere", f2, t1, 0);
    root = mkTree("wurzel", f2, t1, t2);

    printf("1) %s\n", root->k+1);
    printf("2) %d\n", (int)(root->a->i));
    printf("3) %s\n", root->a->n + 1);
    printf("4) %s\n", (root->a->f)(root->k);
    printf("5) %s\n", root->cs[0]->a->n);
    printf("6) %s\n", (*(root->cs[1]->cs))->k);
    printf("7) %d\n", (*(root->cs))->cs[0]->a->i);

    return 0;
}
```

- 1)
- 2)
- 3)
- 4)
- 5)
- 6)
- 7)

Aufgabe 3:

Gegeben sei das folgende C-Programm zur Verarbeitung von Mengen als Bitstrings.

```
#include <stdio.h>

typedef unsigned char Set;
#define SetMax 8

void printSet(Set s) {
    unsigned int i = SetMax;
    while ( i-- != 0 )
        printf("%1u", (unsigned int)((s >> i) & 1));
}

#define PRINT(s) { printSet(s); printf("\n"); }

#define single(i) ( (Set)(1 << (i)) )
#define first(n) (single(n) - 1)
#define interval(n,m) (first(m+1) ^ first(n))

int main(void) {
    Set s1, s2;

    s1 = 0xda; PRINT(s1);
    s2 = -s1; PRINT(s2);
    s2 = s1 & -s1; PRINT(s2);
    s2 = s1 & (~s1 + 1); PRINT(s2);
    s2 = s1 && (~s1 - 2); PRINT(s2);
    s2 = s1 ^ -s1; PRINT(s2);
    s2 = s1 ^ (s1 & (~s1 + 1)); PRINT(s2);
    s2 = (s1 ^ s1) & (~s1 + 1); PRINT(s2);
    s2 = (s1 | first(4)) + 1; PRINT(s2);
    s2 = s1 | single(5); PRINT(s2);

    return 0;
}
```

Die Mengen sind in diesem Beispiel 8 Bits lang, können also die Elemente $0, 1, \dots, 7$ enthalten. *printSet* gibt eine Menge im Binärformat aus. Die Menge, die nur die 1 enthält würde als 00000010 ausgegeben werden. Das *PRINT* Makro gibt jeweils eine Menge pro Zeile aus.

Welche 10 Ausgabezeilen erzeugt dieses Programm unter der Annahme, dass die Maschine mit 2er-Komplement Zahlendarstellung arbeitet?

- 1)
- 2)
- 3)
- 4)
- 5)
- 6)
- 7)
- 8)
- 9)
- 10)

Aufgabe 4:

Tipp: Bitte lesen Sie alle Programmteile einschließlich des Testprogramms und der Fragen sorgfältig durch, bevor Sie mit der Bearbeitung der Aufgabe beginnen.

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 Zugriff und das Einfügen muss man jeweils den richtigen Teilbaum auswählen. Beachten Sie bitte das final Attribute 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 ausgelöst wird.

Die Konkatenation ist etwas trickreich, da es insgesamt 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:

.....