Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

17. August 2010

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2010 (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.

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 19 Seiten.

Aufgabe 1:

Die folgende C Header Datei enthält Deklarationen für eine Listenimplementierung mit verketteten Listen.

```
#include <string.h>

typedef char * Element;
typedef struct node * List;
struct node {
   Element info;
   List next;
};

#define isEmpty(l) ((l) == (List)0)

extern int compare(Element e1, Element e2);
extern int invList(List l);
extern List merge(List l1, List l2);
```

Die hier eingeführten Größen sind bei der Lösung der folgenden Aufgaben zu verwenden.

Implementieren Sie als erstes die *compare* Funktion. Diese soll zwei Elemente vergleichen und als Resultat die Werte -1, 0 und $+1$ liefern, -1 wenn $e1 < e2$ gilt, $+1$ wenn $e1 > e2$ gilt, 0 sonst.

Die *compare* Funktion:

```
#include "List.h"

int compare(Element e1, Element e2) {

   ..........

   ..........

   ..........

   ..........

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

In dieser Aufgabe soll mit sortierten verketteten Listen gearbeitet werden. Die Listen sollen als absteigend sortierte Listen organisiert sein, doppelte Elemente sind nicht erlaubt.

Entwickeln Sie die entsprechende Invariante für diese Bedingungen.

```
#include "List.h"

int invList(List l) {
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
}
```

Es fehlt aus der Header Datei noch die *merge* Funktion. Entwickeln Sie diese Funktion so, dass aus den Knoten der Listen $l1$ und $l2$ eine neue Liste aufgebaut wird, die alle Elemente aus $l1$ und $l2$ enthält und die Invariante wieder gilt. Diese Funktion soll rekursiv arbeiten. Die Funktion soll keine neuen Knoten erzeugen, sondern nur die Knoten aus $l1$ und $l2$ neu verketten. Nicht weiter genutzter Speicher soll freigegeben werden.

```
#include "List.h"

List merge(List l1, List l2) {
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
   .....
}
```

Aufgabe 2:

Gegeben sei das folgende (unvollständige) C–Programmstück für die Implementierung von binären Suchbäumen. Alle Programmteile, die zur Lösung der Aufgabe nicht notwendig sind, sind hier weggelassen.

```
typedef int Element;

int compare(Element e1, Element e2) {
   return (e1 >= e2) − (e1 <= e2);
}

typedef struct node ∗ BinTree;
struct node {
      Element info;
      BinTree l;
      BinTree r;
};

#define isEmpty(b) (! (b))

int searchMax(Element e, BinTree t, Element ∗ max);
```

Entwickeln Sie die Routine **searchMax**. Diese Funktion soll das größte Element in dem Baum suchen, das kleiner als der Parameter e ist. Sie soll als Funktionsresultat berechnen, ob ein solches Element existiert. Im Parameter max soll im Fall der Existenz der gesuchte Wert zurückgegeben werden.

```
int searchMax (Element e, BinTree t, Element * max)
{
  if (isEmpty (t))

    ..........

    ..........

  switch (compare (e, t->info))
    {
    case -1:

    ..........

    ..........

    ..........

    ..........
    case 0:

    ..........

    ..........

    ..........

    ..........
    case +1:

    ..........

    ..........

    ..........

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

Sei n die Anzahl der Elemente, die in einem Baum gespeichert sind.

1. Mit welcher Zeitkomplexität arbeitet diese Funktion im Mittel?

   .....

2. Mit welcher Zeitkomplexität arbeitet diese Funktion im schlechtesten Fall?

   .....

3. Mit welcher Zeitkomplexität würde diese Funktion arbeiten, wenn eine unsortierte verkettete Liste zur Speicherung der Elemente verwendet werden würde?

   .....

4. Mit welcher Zeitkomplexität würde diese Funktion arbeiten, wenn eine sortierte verkettete Liste zur Speicherung der Elemente verwendet werden würde?

   .....

5. Mit welcher Zeitkomplexität würde diese Funktion arbeiten, wenn eine binäre Halde zur Speicherung der Elemente verwendet werden würde?

   .....

Aufgabe 3:

Gegeben seien zwei Funktionen f und g von der Art $\mathbb{N} \to \mathbb{R}$. Definieren Sie, was es heißt, dass $f \in O(g)$ ist. Geben sie dafür sowohl die mathematische Notation als auch eine veranschaulichende Grafik an.

Mathematische Definition:

Grafik:

Aufgabe 4:

Rot–Schwarz–Bäume sind binäre Suchbäume, die gewisse Ausgewogenheitskriterien erfüllen müssen. Diese Bedingungen werden mit Hilfe einer 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 5:

Bitte lesen Sie diese Aufgabe vollständig durch, bevor Sie die Lösungen für die Teilaufgaben entwickeln. Es gibt Vorwärtsreferenzen.

Gegeben sei eine Klassenhierarchie zur Verarbeitung von einstelligen reellwertigen Funktionen, reelwertige Funktionen werden also durch Java–Objekte dargestellt.

Die Schnittstelle für die Funktionen wird als Java–Interface realisiert. Sie enthält eine Methode at zur Berechnung der Funktion an einer Stelle x. Außerdem ist eine Methode $derive$ (ableiten) zur Berechnug der 1. Ableitung zu implementieren. Einige häufig verwendete Funktionen, die Identität, drei konstante Funktionen ($zero$, one, $minus1$) und die Sinus–, Cosinus– und Exponentialfunktion, werden beim Laden der Schnittstelle erzeugt. Sie sind mit Hilfe anonymer Klassen realisiert und sind global zugreifbar.

Die Schnittstelle:

```
interface Function {
    double at(double x);
    Function derive();

    static final Function
        ident = new Function() {
                public double at(double x) {
                    return x;
                }
                public Function derive() {
                    return one;
                }
            };
    static final Function
        zero = new ConstFunction(0.0),
        one = new ConstFunction(1.0),
        minus1 = new ConstFunction(−1.0);

    static final Function
        sin = new Function() {
                public double at(double x) {
                    return Math.sin(x);
                }
                public Function derive() {
                    return cos;
                }
            };

    static final Function
        cos = new Function() {
                public double at(double x) {
                    return Math.cos(x);
```

```
                }
                public Function derive() {
                    return
                        new MultFunction(minus1, sin);
                }
            };

    static final Function
        exp = new Function() {
                public double at(double x) {
                    return Math.exp(x);
                }
                public Function derive() {
                    return exp;
                }
            };
}
```

Die Schnittstelle wird von der folgenden abstrakten Klasse (nächste Seite) beerbt:

```
abstract
public
class AbstractFunction implements Function {
    static public Function constFunction(double c) {
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
    }
    static public Function multFunction(Function f1, Function f2) {
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
    }
    static public Function addFunction(Function f1, Function f2) {
        if (f1 == zero) return f2;
        if (f2 == zero) return f1;
        return new AddFunction(f1, f2);
    }
}
```

Entwickeln Sie analog zur Funktion *addFunction* den Funktionsrumpf für *constFunction*, und zwar so, das die Funktionsobjekte für die konstanten Funktionen $f(x) = 0$, $f(x) = 1$ und $f(x) = -1$ wiederverwendet werden.

Entwickeln Sie in gleicher Weise den Funktionsrumpf für *multFunction*, und zwar so das für Funktionen der Form $f(x) = 0 * f_2(x)$, $f(x) = f_1(x) * 0$, $f(x) = 1 * f_2(x)$ und $f(x) = f_1(x) * 1$ schon vorhandene Funktionsobjekte genutzt werden.

Die Klasse für konstante Funktionen ($ConstFunction$):

final
class ConstFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

Erweitern Sie diese Klassenhierarchie um eine Klasse $AddFunction$ für Funktionen der Form $f(x) = f_1(x) + f_2(x)$. Hinweis: $f'(x) = f_1'(x) + f_2'(x)$

final
class AddFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

Erweitern Sie diese Klassenhierarchie um eine Klasse $MultFunction$ für Funktionen der Form $f(x) = f_1(x) * f_2(x)$. Diese wird von dem Interface $Function$ schon verwendet. Hinweis: $f'(x) = f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$

final
class MultFunction

.....
{

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....
}

Ist es sinnvoll, die Klasse *AbstractFunction* in diesem Beispiel als **public** zu deklarieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Kann man die Funktionen aus *AbstractFunction* in den hier zu entwickelnden Funktionen nutzen?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Ist es sinnvoll, die konkreten Klassen in diesem Beispiel nicht als **public** zu deklarieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .

Ist es sinnvoll, die konkreten Klassen in diesem Beispiel als **final** zu deklarieren?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Kann das **final** Attribut vom Compiler im Allgemeinen zur Verbesserung des JVM–Codes genutzt werden?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Die Schnittstelle $Function$ ist als Java–Interface deklariert. Ist diese Deklaration von Vorteil gegenüber einer Deklaration als abstrakter Klasse?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

Veranschaulichen Sie durch ein Objektdiagramm, was für eine Objektstruktur in der Variablen f aufgebaut wird, wenn der folgende Ausdruck, der die Funktion $(x^2)'$ repräsentiert, ausgewertet wird:

Function f = multFunction(ident,ident).derive();