Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

10. August 2005

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2005 (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 19 Seiten

---

Aufgabe 1:

In dieser Aufgabe geht es darum, die Laufzeit von Operationen für verschiedene Datenstrukturen abzuschätzen. Die Laufzeit von Operationen auf Listen und Bäumen hängt üblicherweise von der Anzahl der Elemente in einer Liste oder in einem Baum ab. In dieser Aufgabe sollen $n, n_1, n_2, ...$ die Anzahl der Elemente in den Listen $l, l_1, l_2, ...$ oder Bäumen $b, b_1, b_2, ...$ bezeichnen. Verwenden Sie bitte die *Groß–O*–Notation.

1. Laufzeitkomplexität für das Einfügen eines Elementes e in eine unsortierte, einfach verkettete Liste mit Duplikaten: $insert(e, l)$

   .....

2. Laufzeitkomplexität für das Einfügen eines Elementes e in eine unsortierte, einfach verkettete Liste ohne Duplikate: $insert(e, l)$

   .....

3. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete Liste mit Duplikaten: $insert(e, l)$

   .....

4. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete Liste ohne Duplikate: $insert(e, l)$

   .....

5. Laufzeitkomplexität für das Anhängen eines Elementes e am Ende einer einfach verketteten Liste: $append(l, e)$

   .....

6. Laufzeitkomplexität für das Konkatenieren zweier einfach verketteter Listen: $concat(l_1, l_2)$

   .....

7. Laufzeitkomplexität für das Mischen aller Elemente zweier unsortierter, einfach verketteter Listen ohne Duplikate: $merge(l_1, l_2)$

   .....

8. Laufzeitkomplexität für das Mischen aller Elemente zweier sortierter, einfach verketteter Listen ohne Duplikate: $merge(l_1, l_2)$

   .....

9. Laufzeitkomplexität für das Suchen eines Elementes e in einer sortierten, einfach verketteten Liste mit Duplikaten: $isIn(e, l)$

. . . . . .

10. Laufzeitkomplexität im Mittel für das Suchen eines Elementes e in einem binären Suchbaum: $isIn(e, b)$

. . . . . .

11. Laufzeitkomplexität im schlechtesten Fall für das Suchen eines Elementes e in einem binären Suchbaum: $isIn(e, b)$

. . . . . .

12. Laufzeitkomplexität im Mittel für das Einfügen eines Elementes e in einen binären Suchbaum: $insert(e, b)$

. . . . . .

13. Laufzeitkomplexität im schlechtesten Fall für das Einfügen eines Elementes e in einen binären Suchbaum: $insert(e, b)$

. . . . . .

14. Laufzeitkomplexität für das Anhängen eines Elementes e am Ende einer einfach verketteten, als Ring implementierten Liste: $append(l, e)$

. . . . . .

15. Laufzeitkomplexität für das Konkatenieren zweier einfach verketteter, als Ring implementierten Listen: $concat(l_1, l_2)$

. . . . . .

16. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete, als Ring implementierte Liste: $insert(e, l)$

. . . . . .

Aufgabe 2:

Gegeben seien die folgenden C-Programmteile für die Implementierung von 2–stelligen Bäumen zur Speicherung von beliebig vielen Werten eines *Element*–Typs. Als Elemente werden in dieser Aufgabe beispielhaft Zeichenketten verwendet. Auf dem Elementbereich sind Vergleichsfunktionen definiert. Diese werden für die Baumalgorithmen benötigt.

Die Definitionen für den *Element*–Datentyp:

```
#include <string.h>

typedef char *Element;

int ge(Element e1, Element e2)
{
  return strcmp(e1, e2) >= 0;
}

int gr(Element e1, Element e2)
{
  return strcmp(e1, e2) > 0;
}

int ls(Element e1, Element e2)
{
  return gr(e2, e1);
}
```

Die Definitionen für die Baum–Datenstruktur:

Für die zu entwickelnden Algorithmenteile sind für den Baum–Datentyp einige nützliche Vergleichsfunktionen vorgegeben, die in den Programmteilen auch zu verwenden sind.

```
typedef struct node *Tree;

struct node
{
  Element info;
  Tree l;
  Tree r;
};

Tree mkEmpty(void)
{
  return (Tree) 0;
}

int isEmpty(Tree t)
{
  return t == (Tree) 0;
}

int isGE(Tree t, Element e)
{
  return isEmpty(t) || ge(t->info, e);
}

int isGR(Tree t, Element e)
{
  return isEmpty(t) || gr(t->info, e);
}

int isLS(Tree t, Element e)
{
  return isEmpty(t) || ls(t->info, e);
}
```

Die oben definierten Datentypen können zur Implementierung einer binären Halde (oder Vorrangwarteschlange) verwendet werden. Für Vorrangwarteschlangen gilt die Datenstruktur–Invariante, dass für alle Knoten die an den Kindknoten gespeicherte Information nicht kleiner sein darf als die Information an dem Knoten selbst. Dieses kann mit einer Funktion überprüft werden.

Entwickeln Sie die fehlenden Teile dieser Funktion:

```
int invBinHeap(Tree t)
{
  return

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

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

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

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

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

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

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

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

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

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

Die gleiche Datenstruktur kann für die Implementierung binärer Suchbäume verwendet werden. Nur muss dann eine andere Invariante für die Datenstruktur gesichert sein. Die folgende Funktion ist hierzu ein Versuch.

```
int invBinSearchTree(Tree t)
{
  return
    isEmpty(t) ||
    (isLS(t->l, t->info)
     && isGR(t->r, t->info)
     && invBinSearchTree(t->l)
     && invBinSearchTree(t->r));
}
```

Diese Funktion enthält einen groben Denkfehler. Schreiben Sie neue Vergleichsfunktionen *isLS1* und *isGR1*, die diesen Fehler beheben.

```
int isLS1(Tree t, Element e)
{
  return

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

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

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

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

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

int isGR1(Tree t, Element e)
{
  return

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

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

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

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

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

Bei der Entwicklung der Vergleichsfunktionen *isGE, isGR* und *isLS* ist mit Kopieren und Einfügen gearbeitet worden. Diese drei Funktionen können durch Abstraktion zusammengefasst werden zu einer allgemeinen Vergleichsfunktion mit einem zusätzlichen Parameter. Entwickeln Sie diese verallgemeinerte Funktion und nutzen Sie diese zur Reimplementierung der drei Funktionen.

```
/* die Typdefinition für den zusätzlichen Parameter */

typedef ..........................................

int isInRel(Tree t, Element e, Rel r)
{
  return

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

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


int isGE(Tree t, Element e)
{
  return

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

int isGR(Tree t, Element e)
{
  return

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

int isLS(Tree t, Element e)
{
  return

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

Aufgabe 3:

Gegeben sei die folgende Java-Klasse.

```
public class Buffer {
    private boolean empty = true;
    private Data value = null;

    public void put(Data d) {
        value = d;
        empty = false;
    }

    public Data get() {
        Data d = value;

        value = null;
        empty = true;

        return d;
    }
}
```

Diese Klasse implementiert einen Puffer für ein Exemplar aus der Klasse *Data*. Es soll dabei sicher gestellt sein, dass der Puffer entweder leer ist, angezeigt durch die Variable *empty*, oder voll, also eine Referenz auf ein *Data*-Objekt enthält. Diese Eigenschaft wird in der Variablen *empty* gespeichert.

Diese Klasse ist nicht *Thread*-sicher. Außerdem wird nicht sichergestellt, dass die *put*- und *get*-Operationen immer genau wechselseitig aufgerufen werden, so dass alle mit *put* geschriebenen Daten auch genau einmal mit *get* gelesen werden.

Erweitern Sie die *get*- und *put*-Methoden so, dass diese *Thread*-sicher sind und dass die zusätzlichen Bedinungen für den Einsatz in einem Erzeuger-Verbraucher-Muster für die *Buffer*-Klasse erfüllt sind.

Hinweis: in Java gibt es die Methoden *wait()* und *notify()*. *wait()* kann eine überprüfte Ausnahme *InterruptedException* auslösen.

Die modifizierte *put()*-Methode:

Die modifizierte *get()*-Methode:

Aufgabe 4:

Gegeben sei das folgende Testprogramm:

```
public class ExcTest {

    public static void main(String[] argv) {
        int i = 0;
        try {
            i = Integer.parseInt(argv[0]);
        }
        catch ( Exception e ) { }
        test(i);
    }

    public static void test(int i) {

    tryblock:
        try {
            switch ( i ) {
            case 1:
                System.out.println("test:  call bar");
                bar();
                break;
            case 2:
                System.out.println("test:  call foo");
                foo();
                break;
            default:
                System.out.println("test:  normal exit");
            }
        }
        catch ( MyException e ) {
            System.out.println("test:  caught MyException");
        }
        catch ( Exception e ) {
            System.out.println("test:  caught Exception");
        }
        finally {
            System.out.println("test:  exec finally");
        }
        System.out.println("test:  normal exit");
    }
```

```
    public static void foo()
        throws MyException {
        int [] a = {0};
        int res;
        try {
            System.out.println("foo :  array access");
            res = a[1];
        }
        catch (Exception e) {
            System.out.println("foo :  caught Exception");
            System.out.println("foo :  throw MyException");

            throw
                new MyException("foo test");
        }
    }

    public static void bar()
        throws MyException {
        try {
            System.out.println("bar :  throw MyException");
            throw
                new MyException("bar test");
        }
        finally {
            System.out.println("bar :  exec finally");
        }
    }
}

class MyException extends Exception {
    public
        MyException(String s) {
        super(s);
    }
}
```

Welche Ausgabe wird bei einem Aufruf von `java ExcTest 1` erzeugt:

.....

.....

.....

.....

.....

.....

.....

Welche Ausgabe wird bei einem Aufruf von `java ExcTest 2` erzeugt:

.....

.....

.....

.....

.....

.....

.....

Aufgabe 5:

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

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, drei konstante Funktionen (*zero*, *one*, *minus*1) und die Sinus–, Cosinus– und Exponetialfunktion, werden beim Laden der Schnittstelle erzeugt. Diese sind global zugreifbar.

Die Schnittstelle:

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

    static final Function
        zero = new ConstFunction(0.0),
        one = new ConstFunction(1.0),
        minus1 = new ConstFunction(−1.0),
        sin = new Function() {
                public double at(double x) {
                    return Math.sin(x);
                }
                public Function derive() {
                    return cos;
                }
            },
        cos = new Function() {
                public double at(double x) {
                    return Math.cos(x);
                }
                public Function derive() {
                    return
                        new MultFunction(minus1, sin);
                }
            },
        exp = new Function() {
                public double at(double x) {
                    return Math.exp(x);
                }
                public Function derive() {
                    return exp;
                }
            };
}
```

Die Klasse für konstante Funktionen (*ConstFunction*):

```
public final
class ConstFunction implements Function {
  private
  double c;

  public ConstFunction(double c) {
    this.c = c;
  }

  public double at(double x) {
    return
      c;
  }

  public Function derive() {
    return
      zero;
  }
}
```

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

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: $f'(x) = f_1'(x) * f_2(x) + f_1(x) * f_2'(x)$

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 abstrakte Klasse?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

. . . . . .
