Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

19. August 2008

---

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2008 (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 18 Seiten.

**Vorsicht: Lesen gefährdet die Dummheit!**

---

**Aufgabe 1:**

In dem folgenden Programmstück wird mit variabel langen Mengen für Zahlen gearbeitet. Die Mengen werden dabei durch Bitfolgen repräsentiert. Diese Bitfolgen werden in Feldern von Wörtern gespeichert. Die in den Mengen speicherbaren Elemente bilden immer ein Intervall der natürlichen Zahlen mit der unteren Grenze 0.

Vervollständigen Sie die Funktion *mkEmptySet*. Diese soll dynamisch Speicher für eine neue Menge besorgen und diese Menge so initialisieren, das sie leer ist. Der Parameter dieser Funktion gibt an, wieviele Elemente die Menge maximal aufnehmen kann.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <limits.h>

typedef unsigned int Word;
typedef Word * Set;

#define SLEN (CHAR_BIT * sizeof (Word))

#define empty ((Word)0)
#define full (~empty)
#define single(i) ((Word)1 << (i))
```

```
Set
mkEmptySet (unsigned int len)
{
  unsigned int setWords =

    ..................................................;
  Set res =

    ..................................................;

  if ( ..................................................)
    {

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

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

  {

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

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

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

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

  return res;
}
```

Die Operationen Test auf Enthaltensein (*elem*), ein Element in eine Menge einfügen (*add*) und ein Element löschen (*remove*) sind sehr häufig vorkommende Mengenoperationen. Diese sollen hier als Makros implementiert werden. Es soll dabei auf einen Bereichstest verzichtet werden. Verwenden Sie in den Definitionen dieser Makros die im Programm schon eingeführten Größen. Der Parameter *s* bezeichnet dabei immer die Menge, *e* das Element.

`#define` elem(e, s) .....

.....

.....

`#define` add(s, e) .....

.....

.....

`#define` remove(s, e) .....

.....

.....

Was muss bei der Verwendung dieser Makros beachtet werden?

.....

.....

---

**Aufgabe 2:**

Gegeben seien die folgenden Typdefinitionen und Variablendeklarationen:

```
typedef char *Key;
typedef struct ANode * Attr;
typedef struct Node *List;

struct Node
{
  Key k;
  Attr v;
  List next;
};

struct ANode
{
  char * name;
  unsigned long age[3];
};

typedef struct hashtable *Map;

struct hashtable
{
  unsigned int size;
  unsigned int card;
  List *table;
};

typedef int (* Cmp)(Key k1, Key k2);

extern int compare(Key k1, Key k2);

extern unsigned int hash(Key e);

extern Attr search(Key k, Map m, Cmp c);

extern Map mkEmpty(void);

extern Map m;

extern Key k1,k2;
```

Bestimmen Sie für die folgenden Ausdrücke den Typ gemäß ANSI-C. Nutzen Sie hiefür, wenn möglich, die deklarierten Typnamen. Sollten Ausdrücke vorkommen, die zur Übersetzungszeit Fehlermeldungen erzeugen, so kennzeichnen Sie diese mit dem Wort `FEHLER`

$*$m .....

mkEmpty()$->$table .....

($*$m).card == 0 .....

m$->$table[3] .....

$*$(m$->$table[0]$->$v) .....

$*$(($*$(m$->$table))$->$next) .....

search(k1,m,compare) .....

search(k1,m,compare(k1,k2)) .....

compare .....

m$->$table[compare(k1,k2)] .....

m$->$table[0]$->$next$->$next$->$v$->$name[2] .....

$*$(m$->$table[1]$->$next$->$next$->$v$->$age + 2) .....

---

**Aufgabe 3:**

Das folgende Programmstück definiert die Datentypen für die Implementierung einer Liste als einfach verkettetem Ring.

```
typedef char * Element;

typedef struct Node * Ring;
struct Node {
   Ring next;
   Element e;
};

extern Ring concat(Ring r1, Ring r2);
```

Entwickeln Sie die fehlende *concat*–Routine zur Konkatenation zweier Listen.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Mit welcher Zeitkomplexität arbeitet diese Routine, wenn $n_1$ und $n_2$ die Längen der beiden Listen bezeichnen.

. . . . . .

**Aufgabe 4:**

Vorrang–Warteschlangen werden zur Speicherung beliebig vieler Elemente oder Schlüssel–Wert–Paare verwendet, wobei nur der schnelle Zugriff auf das kleinste Element aus einer Menge gefordert wird. Das effiziente Suchen eines beliebigen Elements ist nicht gefordert. Elemente dürfen hierbei durchaus doppelt vorkommen.

Dieses kann mit einem binären Baum effizient implementiert werden, wenn der Baum so aufgebaut wird, dass die Wurzel eines jeden Teibaums immer einen Wert enthält, der nicht größer ist als die Werte die in den Teilbäumen gespeichert werden.

Das folgende Programmstück definiert die Datentypen und ein Prädikat für die Invariante.

```
typedef double Element;

typedef struct Node * BinaryHeap;

struct Node
{
  Element info;
  BinaryHeap l;
  BinaryHeap r;
};

int isEmptyBinaryHeap(BinaryHeap h) {
  return h == (BinaryHeap)0;
}

static
int
greaterOrEqual (BinaryHeap h, Element e);

extern
int invBinaryHeap(BinaryHeap h);
```

Entwickeln Sie die Invariante einschließlich der Hilfsfunktion:

**greaterOrEqual**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

**invBinaryHeap**

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Mit welcher Zeitkomplexität arbeitet die Invariante?

. . . . . .

**Aufgabe 5:**

Bitte lesen Sie diese Aufgabe vollständig durch, bevor Sie beginnen, die einzelnen Lösungsteile zu entwickeln.

In dieser Aufgabe soll ein einfacher Baukasten für die Verarbeitung von Folgen über den reellen Zahlen entwickelt werden. Folgen besitzen folgende Schnittstelle:

```
interface Sequence {
    double at(int i);
}
```

Die Funtion *at* berechnet den Wert der Zahlenfolge an der Stelle *i*. Legale Indizes sind alle natürlichen Zahlen ab 0. Für Aufrufe mit negativen Indizes soll in dieser Aufgabe keine Fehlererkennung gemacht werden.

Jede Folge kann durch ein Objekt einer Klasse repräsentiert werden, die dieses Interface *Sequence* implementiert. Dieses ist unflexibel, insbesondere können so zur Laufzeit nicht dynamisch neue Folgen erzeugt werden. Geschickter ist es, Klassen für einfache Folgen zu entwickeln und Kombinator–Klassen, dass heisst Klassen mit denen man aus bestehenden Folgen neue zusammensetzen kann.

Für einfache Folgen, gibt es einige einfache Klassen. Mit der Klasse *Const* können konstante Zahlenfolgen generiert werden.

```
public
class Const implements Sequence {
  private
  double value;

  public Const() { this(0.0); }

  public Const(double value) { this.value = value; }

  public double at(int i) { return value; }
}
```

Mit *Ident* erhält man die Zahlenfolge $0.0, 1.0, 2.0, ....$

```
public
class Ident implements Sequence {
  public double at(int i) { return (double)i; }
}
```

Entwickeln Sie eine Klasse *Square* für die Folge der Quadratzahlen: $0.0, 1.0, 4.0, 9.0, 16.0$....

```
public
class Square implements Sequence {

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

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

   ..........................................
   public double at(int i) {
      return

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

Aus einer Folge können neue Folgen erzeugt werden. Eine einfache Art ist die, jeden Folgenwert mit einem Faktor zu multiplizieren. Entwickeln Sie hierfür eine Klasse *Scale*:

```
public
class Scale implements Sequence {

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

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

   ..........................................
   public Scale( .......................................... ) {

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

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

   public double at(int i) {

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

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

Folgen können durch Kombination zweier anderer Folgen konstruiert werden, zum Beispiel durch Addition oder Multiplikation der Folgeglieder. Die Addition der Folgen *new Ident()* und *new Const(1.0)* ergibt dabei die Folge $1.0, 2.0, 3.0, ....$ Entwickeln Sie eine Klasse *Mult* zur Multiplikation von Folgen.

```
public class Mult implements Sequence {

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

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

   public Mult( ..........................................  ) {

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

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

   public double at(int i) {
      return

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

Geben Sie einen Ausdruck an, der die Folge für die Quadratzahlen erzeugt, ohne dass die Klasse *Square* genutzt wird. Bitte verwenden Sie auch keine anonymen Klassen.

..........

..........

Aus Folgen können Reihen gebildet werden. Eine Reihe ist eine Folge, die durch das Aufsummieren der Glieder einer gegebenen Folge entstehen. Dieses soll hier mit Hilfe der Klasse *Sum* realisiert werden. Der Algorithmus soll so arbeiten, dass für die Zahlenfolge $1.0, 1.0, 1.0, ...$ (*new Const(1.0)*) die Zahlenfolge $0.0, 1.0, 2.0, 3.0, ...$ entsteht. Versuchen Sie, eine nicht iterative Lösung für den Algorithmus zu entwickeln.

```
public class Sum implements Sequence {

   .....

   .....

   public Sum( ..... ) {

   .....
   }

   public double at(int i) {

      .....

      .....

      .....

      .....

      .....

      .....
   }
}
```

Aus einer Folge kann eine Differenzenfolge erzeugt werden, indem der $i$-te Wert der Ausgangsfolge vom $i + 1$-ten subtrahiert wird. Entwickeln Sie hierfür eine Klasse *Diff*.

```
public class Diff implements Sequence {

   . . . . . .

   public Diff( . . . . . ) {

      . . . . . .
   }

   public double at(int i) {

      . . . . . .

      . . . . . .
   }
}
```

In einer Anwendung kann es vorkommen, dass für einen Index der Folgewert mehrfach berechnet wird. Dieses kann zu Laufzeitineffizenzen führen. Solche Ineffizienzen kann man beheben, indem man die Werte einer Folge spechert und diesen Cache mit der gleichen Schnittstelle *Sequence* versieht. Diese Ineffizenzen können in dieser Aufgabe insbesondere in der *Sum*-Klasse entstehen.

Nutzen Sie für die Implementierung einer solchen Cache-Klasse als Cachespeicher die Klasse *IntMap*. Der Konstruktor dieser Klasse soll eine neue leere Tabelle (*Map*) erzeugen. Mit *insert* kann man ein beliebiges Objekt unter dem Schlüssel *i* speicher. Mit *lookup* kann man testen, ob für einen Schlüssel *i* ein Wert gespeichert ist, und wenn ja welcher. *lookup* gibt *null* im Fall nicht gefunden und sonst die Referenz auf den gespeicherten Wert als Resultat.

```
public class IntMap {
   public IntMap() {
      // ...
   }
   public void insert(int k, Object a) {
      // ...
   }
   public Object lookup(int k) {
      // ...
   }
}
```

Entwickeln Sie eine Klasse *SequenceCache* für die Vermeidung der wiederholten Berechnung der Glieder einer Zahlenfolge.

```
public class SequenceCache implements Sequence {

   . . . . . .

   . . . . . .
   private

   . . . . . .
   public SequenceCache( . . . . . ) {

      . . . . . .

      . . . . . .
   }
   public double at(int i) {
      double res;

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .

      . . . . . .
      return res;
   }
}
```

Würde eine Folge *new SequenceCache(new Sum(f))* den Cache in einer effizienten Art nutzen?

ja ☐ nein ☐

Begründung:

. . . . . .

. . . . . .

**Aufgabe 6:**

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:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .