Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

21. August 2007

---

Aufgaben zur Klausur **C** und **Objektorientierte Programmierung** im SS 2007 (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

---

**Aufgabe 1:**

Gegeben sei das folgende (unvollständige) C–Programmstück für die Implementierung von binären Suchbäumen. Alle Programteile, 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 oder gleich dem 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 2:**

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 ersetes die *compare* Funktion. Diese soll zwei Elememte 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 aufsteigend sortierte Listen organisiert sein, doppelte Elemente sollen erlaubt sein.

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 Einträge aus $l1$ und $l2$ enthält und die Invariante wieder gilt. Diese Funktion soll rekursiv arbeiten.

```
#include "List.h"

List merge(List l1, List l2) {

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

**Aufgabe 3:**

Gegeben seien die folgenden Variablen und Funktionen:

```
typedef int (*Fct1)(void);
typedef void (*Fct2)(double);
typedef int (*Fct3)(int, int);
unsigned long int x,y;
long int s;
float f;
unsigned char *p1;
int *p2;
void *p3;
int (*pf)(void);
void (*pf2)(double);
int f1(void);
int f2(int x1,int x2);
void f3(double x1);
```

Bestimmen Sie für die folgenden Ausdrücke den Typ gemäß ANSI-C. Vorsicht: Es kommen fehlerhafte und logisch nicht sinnvolle Ausdrücke von. Kennzeichnen Sie diese mit dem Wort ERROR.

*p1 & p1 .....

*p1 && p1 .....

*(p2+x+y) .....

*p3 .....

(*pf)(f1()) .....

*(x+p3) .....

f1=pf .....

f3==pf2 .....

p1 ? pf : f1 .....

! p2 .....

p3[0] .....

p3+x .....

pf=f1 .....

~p2 .....

s || x .....

s | x .....

**Aufgabe 4:**

Generische ADTs werden in C++ durch sogenannte *templates* unterstützt, in Java ab 1.5 durch *generics*. Überprüfen Sie die folgenden Aussagen über generische ADTs.

1. Typparameter für Klasen werden zur Laufzeit des Programms in zusätzlichen Feldern in den Objekten gespeichert und führen so zu größerem Speicherbedarf zur Laufzeit.

   ja ☐ nein ☐

   Begründung:

   .....

2. Typparameter für Klassen sind ein Konzept, das die Flexibilität von Java– und C++–Programmen einschränkt.

   ja ☐ nein ☐

   Begründung:

   .....

3. In Java Programmen mit Typparametern lassen sich up– und down–casts zur Laufzeit vermeiden.

   ja ☐ nein ☐

   Begründung:

   .....

4. In Programmen mit Typparametern können Fehler in der Verwendung von Methoden und Variablen teilweise schon zur Übersetzungszeit entdeckt werden.

   ja ☐ nein ☐

   Begründung:

   .....

5. In Java Programmen mit Typparametern können zur Laufzeit noch mehr Überprüfungen gemacht werden als ohne.

   ja ☐ nein ☐

   Begründung:

   .....

6. In Programmen mit Typparametern können Laufzeit–Überprüfungen auf legale Feldzugriffe oder das Dereferenzieren von null-Referenzen teilweise eliminiert werden.

   ja ☐ nein ☐

   Begründung:

   .....

**Aufgabe 5:**

Die folgenden Fragen beziehen sich alle auf Java Programme, die mit Threads arbeiten. Es wird dabei angenommen, dass kein Thread sich unbeschränkt lange in einem Monitor aufhält, zum Beispiel indem er in eine Endlosschleife läuft.

1. Threads brauchen nicht synchronisiert werden, wenn alle Threads gemeinsame Variablen nur lesen.

   ja ☐ nein ☐

   Begründung:

   .....

2. Nur die Threads, die gemeinsame Variablen auch beschreiben, müssen beim Zugriff auf die Variablen synchronisiert werden. Die ausschließlich lesenden brauchen nicht synchronisiert werden.

   ja ☐ nein ☐

   Begründung:

   .....

3. Threads brauchen nicht synchronisiert werden, wenn sie gemeinsame Variablen nur schreiben.

   ja ☐ nein ☐

   Begründung:

   .....

4. Wenn es zur Berechnung einer Aufgabe ein deterministisches Programm und ein nichtdeterministisches Programm gibt, so ist das deterministische das effizientere.

   ja ☐ nein ☐

   Begründung:

   .....

5. Thread–Programme, die nur mit einer einzigen gemeinsamen synchronisierten Variablen arbeiten, sind immer Deadlock–frei.

   ja ☐ nein ☐

   Begründung:

   .....

**Aufgabe 6:**

Die folgenden Klasse **NTree** und einige Hilfsklassen und Schnittstellen dienen zur Implementierung von beliebigstelligen Bäumen.

In diesen Klassen sind einige Methodenrümpfe zu entwickeln, und zwar zur einheitlichen Verarbeitung aller Knoten eines Baumes.

Die Methode **toString** dient hier zur Testausgabe. Sie ist bei der Entwicklung der anderen Methoden nicht zu verwenden.

Die Methode **noOfElements** soll die Anzahl der Knoten in einem Baum berechnen.

Die Methode **map** soll aus einem Baum durch Anwenden einer Funktion von der Art **Function** auf alle Knoten einen neuen Baum berechnen. Es soll hierbei der Knoten selbst als erstes verarbeitet werden, anschließend die Kinder in der Reihenfolge, in der sie im Feld abgespeichert sind.

Die Methode **copy** soll für eine komplette Baumstruktur eine physikalische Kopie anlegen, es sollen also alle Exemplare von **NTree** dupliziert werden. Die Knoteninformation selbst soll unverändert bleiben.

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

Tipp: Bitte vermeiden Sie die Verdopplung von Algorithmen(–teilen).

Die Schnittstelle für die Representation von einstelligen Funktionen:

```
interface Function {
    public Object apply(Object o);
}
```

**Die Klasse für die Baumstruktur:**

```
public class NTree {
    private Object node;
    private NTree [] children;

    private NTree(Object n,
                  NTree [] cs) {
        node = n;
        children = cs;
    }
```

NTree (cont.): Die erzeugenden Funktionen und die Testausgabe

```
private static final NTree [] ecs = new NTree[0];

public static NTree mkLeave(Object n) {
    return
        mkTree(n, ecs);
}

public static NTree mkTree(Object n,
                           NTree [] cs) {

    return
        new NTree(n, cs);
}

public static NTree mkTree1(Object n,
                            NTree c) {
    return
        mkTree(n, new NTree []{c});
}

public static NTree mkTree2(Object n,
                            NTree c1,
                            NTree c2) {
    return
        mkTree(n, new NTree []{c1, c2});
}

public String toString() {
    String cs = "";
    for (int i = 0; i < children.length; ++i) {
        cs += children[i].toString();
        if (i < children.length -1) {
            cs += ",";
        }
    }
    return
        node.toString() + "(" + cs + ")";
}
```

NTree (cont.): Die Methode zur Berechnung der Anzahl der Knoten

```
public int noOfNodes() {

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

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

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

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

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

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

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

NTree (cont.): Die Verarbeitung aller Knoten

```
public NTree map(Function f) {
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
    ..........................
}
```

NTree (cont.): Das Klonen

```
    public NTree copy() {
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
        ..........
    }

}
```

Ende von **NTree**

**Eine Klasse für eine Funktion zur Berechnung der Anzahl Zeichen eines String–Objekts.**

Hier soll angenommen werden, dass die Klasse von Anwendern immer *vernünftig* verwendet wird.

```
public class StringLengthFunction
    implements Function {

    public Object apply(Object o) {

        ..........

        ..........

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

**Eine Klasse zum Durchnummerieren aller Knoten eines Baumes.**

Hierbei soll jedes Objekt auf eine Zahl abgebildet werden, und zwar auf die Stelle an der das Objekt beim Durchlauf verabeitet wird. Es soll beim Zählen mit 1 begonnen werden, der Wurzelknoten wird also auf die 1 abgebildet.
Tipp: Bitte vorher das Testprogramm genau durcharbeiten.

```
public class NumberNodesFunction
    implements Function {

        ..........

        ..........
    public Object apply(Object o) {

        ..........

        ..........

        ..........

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

**Ein einfaches Testprogramm:**

```
public class Test {
    public static void main(String [] argv) {
        NTree t1 =
            NTree.mkTree2("tooLess",
                          NTree.mkTree1("ham",
                                        NTree.mkLeave("and")),
                          NTree.mkLeave("eggs")
                          );

        System.out.println(t1.noOfNodes());

        System.out.println(t1.copy());

        System.out.println(t1.map(new NumberNodesFunction()));

        System.out.println(t1.map(new StringLengthFunction()));

    }
}
```

Welche 4 Zeilen gibt dieses Testprogramm aus?
Tipp: Eine kleine Skizze über den Aufbau des Baumes $t1$ kann hier und für die Entwicklung der Algorithmen hilfreich sein.

1) .....

2) .....

3) .....

4) .....

---