Prof. Dr. Uwe Schmidt

ptlwedel
PRIVATE BERUFSFACHSCHULE

21. August 2012

Aufgaben zur Klausur **C** im SS 2012 (IA 302)

Zeit: 90 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 9 Seiten.

Aufgabe 1:

Gegeben sei das folgende Programm

```
#include <stdio.h>

int x1 [] = {1,2,3};
int x2 [] = {4,2,6};
int x3 [] = {9,7,5};

int * a[] = {x3,x2,x1};
int ** pp, *p, i;

int main(void) {

  pp = a, i = * * pp;
                                        printf("%d\n",i);
  p = *(++pp), i = *(++p);
                                        printf("%d\n",i);
  i = *( *(a + 1 ) +1 );
                                        printf("%d\n",i);
  pp = a + 1, i = pp[1][2];
                                        printf("%d\n",i);
  pp = a + 1, p = pp[-1] + 1; i = p[-1];
                                        printf("%d\n",i);
  pp = a + 2, --pp, i = (*pp == a[1]);
                                        printf("%d\n",i);
  p = a[2] + 1, i = *p;
                                        printf("%d\n",i);
  pp = 1 + a, p = 2 + *pp, i = *(1 + p);
                                        printf("%d\n",i);
  return 0;
}
```

Tipp: Skizzen sind manchmal hilfreich.

Wird in diesem Programm an irgend einer Stelle eine implizite Konversion durchgeführt?

ja ☐ nein ☐

Begründung:

. . . . . .

Wie sieht die Ausgabe des Programms aus ?

1) . . . . . .

2) . . . . . .

3) . . . . . .

4) . . . . . .

5) . . . . . .

6) . . . . . .

7) . . . . . .

8) . . . . . .

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 (*Rel) (Element e1, Element e2);

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:

Entwickeln Sie Makros zur Erzeugung von Bitmasken. Die Makros sollen Ausdrücke vom Typ **unsigned int** erzeugen. Sie sollen unabhängig von der Zahlendarstellung in der Maschine, 1–er oder 2–er-Komplement, sein.

1. Ein Makro `low_zeroes(n)` zum Setzen der `n` niederwertigen Bits auf 0, alle anderen Bits sollen auf 1 gesetzt werden.

   .....

   .....

   .....

2. Ein Makro
   low_ones(n) zum Setzen der `n` niederwertigen Bits auf 1, alle anderen Bits sollen auf 0 gesetzt werden.

   .....

   .....

   .....

3. Ein Makro
   mid_zeroes(width,offset), das die niederwertigen `offset` Bits auf 1 setzt, die folgenden `width` Bits auf 0, und alle übrigen wieder auf 1.

   .....

   .....

   .....

4. Ein Makro
   mid_ones(width,offset), das die niederwertigen `offset` Bits auf 0 setzt, die folgenden `width` Bits auf 1, und alle übrigen wieder auf 0.

   .....

   .....

   .....