Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

19.August 2008

Aufgaben zur Klausur **Algorithmen und Datenstrukturen in C** im SS 2008 (BInf 201, BTInf 201, BMInf 201, BWInf 201)

Zeit: 75 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.

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?

. . . . . .