Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

21.August 2007

Aufgaben zur Klausur **Algorithmen und Datenstrukturen in C** im SS 2007 (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

---

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 .....