Prof. Dr. Uwe Schmidt

fhwedel
UNIVERSITY OF APPLIED SCIENCES

14. August 2014

Aufgaben zur Klausur **Algorithmen und Datenstrukturen in C** im SS 2014 (B_ECom 17b, B_Inf v201 od. 17b, B_TInf v201 od. 17b, B_MInf v201 od. 17b, B_WInf v201 od. 17b)

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.

Nutzen Sie die Rückseiten der Klausur zur Entwicklung der Lösungen und übertragen die fertigen Lösungen in das Aufgabenblatt.

Sollten Unklarheiten oder Mehrdeutigkeiten bei der Aufgabenstellung auftreten, so notieren Sie bitte, wie Sie die Aufgabe interpretiert haben.

Tipp: Bei der Entwicklung der Lösung können kleine Skizzen manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 12 Seiten.

Aufgabe 1:

Gegeben sei das folgende (unvollständige) C–Programmstück für die Implementierung von binären Suchbäumen. Alle Programmteile, 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 echt kleiner als der 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:

Rot–Schwarz–Bäume sind binäre Suchbäume, die gewisse Ausgewogenheitskriterien erfüllen müssen. Diese Bedingungen werden mit Hilfe einer Invarianten formuliert. Eine Bedingung ist die, dass keine roten Knoten einen roten Kindknoten besitzen.

Die Datenstruktur–Definition für einen als Rot–Schwarz–Baum realisierten Mengendatentyp habe folgendes Aussehen:

```
typedef int Element;

typedef struct Node * Set;

struct Node
{
  enum { RED, BLACK } color;
  Element info;
  Set l;
  Set r;
};

static struct Node finalNode = {BLACK, 0, 0, 0};

#define mkEmptySet() (&finalNode)
#define isEmptySet(s) ((s) == &finalNode)

extern unsigned int maxPathLength(Set s);
extern unsigned int minPathLength(Set s);

#define isBlackNode(s) ((s)->color == BLACK)
#define isRedNode(s) (! isBlackNode(s))

extern int hasRedChild(Set s);
extern int invNoRedNodeHasRedChild(Set s);
```

In diesem Codefragment sind einige Makros und einige Funktionen deklariert. Die Bedeutung der Funktionen geht aus dem Namen hervor. Benutzen Sie bitte diese Makros und Funktionen zur Formulierung Ihrer Lösung.

Man erkennt an den Makros *isEmptySet* und *mkEmptySet*, dass in dieser Implementierung der leere Baum durch einen Zeiger auf einen speziellen Knoten repräsentiert wird, nicht durch den 0–Zeiger.

Es soll als erstes die Hilfsfunktion *hasRedChild* entwickelt werden. Dieses Prädikat soll überprüfen, ob für einen beliebigen Baum ein möglicher Wurzelknoten einen roten Kindknoten besitzt.

Die Funktion *hasRedChild*:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Mit dieser Hilfsfunktion kann der Teil der Invariante für Rot–Schwarz–Bäume formuliert werden, der überprüft, dass an keiner Stelle in einem Baum ein roter Knoten einen roten Kindknoten besitzt. Dieses überprüft die Funktion *invNoRedNodeHasRedChild*

Die Funktion *invNoRedNodeHasRedChild*:

Aufgabe 3:

Gegeben sei das folgende C–Programm zur Verarbeitung von Mengen als Bitstrings.

```
#include <stdio.h>

typedef unsigned char Menge;
#define MengeMax 8

void printMenge(Menge s) {
    unsigned int i = MengeMax;
    while ( i-- != 0 ) {
      printf("%1u", (unsigned int)((s >> i) & 1));
      if (i == 4)
        printf("  ");
    }
}
static unsigned int linecnt = 0;
#define PRINT(s) { printf("%2u)  ", ++linecnt); printMenge(s); printf("\n"); }

#define einStueck(n,m) (dieErsten(m+1) ^ dieErsten(n))
#define dieErsten(n) (einElement(n) - 1)
#define einElement(i) ( (Menge)(1 << (i)) )

int main(void) {
    Menge s1;

    PRINT( einElement(2) );
    PRINT( einElement(MengeMax) );

    PRINT( (Menge)1 );
    PRINT( (Menge)0xbc );

    PRINT( einStueck(5,5) );
    PRINT( einStueck(3,1) );
    PRINT( einStueck(0,MengeMax-1) );

    PRINT( 0x22 & 0x20 );
    PRINT( 0x22 && 0x20 );

    s1 = einStueck(1,4) | ~einStueck(2,6); PRINT(s1);
    s1 = einStueck(1,4) ^ (Menge)((128 - 1) * 4); PRINT(s1);

    s1 = 8 + 16 + 32;
    s1 = s1 ^ (s1 & (~s1 + 1)); PRINT(s1);
    return 0;
}
```

Die Mengen sind in diesem Beispiel 8 Bits lang, können also die Elemente $0, 1, \ldots, 7$ enthalten. *printSet* gibt eine Menge im Binärformat aus. Die Menge, die nur die 1 enthält würde als $0000\ 0010$ ausgegeben werden. Das *PRINT* Makro gibt jeweils eine Menge pro Zeile aus und numeriert die Zeilen durch.

Welche 12 Ausgabezeilen erzeugt dieses Programm?

1) .....

2) .....

3) .....

4) .....

5) .....

6) .....

7) .....

8) .....

9) .....

10) .....

11) .....

12) .....

Aufgabe 4:

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`

mkEmpty()->table .....

*m .....

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

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

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

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

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

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

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

compare .....

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

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

Aufgabe 5:

Seien f_1, f_2, g_1, g_2 Funktionen vom Typ $\mathbb{N} \to \mathbb{R}$.
Weiter gelte $f_i(n) \in O(g_i(n))$ und $c_i \in \mathbb{R}$ für $i \in \{1, 2\}$.

1. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * (f_1(n) + c_2 * n^2)$

   .....

2. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * f_1(n) * n + c_2 * f_1(n) * n$

   .....

3. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) * (f_2(n) + c_2 * n)$

   .....

4. Aus welcher Komplexitätsklasse ist $f(n) = (f_1(n))^2 + f_1(n)$

   .....
