Prof. Dr. Uwe Schmidt

ptlwedel
PHYSIKALISCH-TECHNISCHE LEHRANSTALT

17. August 2006

Aufgaben zur Klausur **C** im SS 2006 (IA 302)

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 seien die folgenden Datentypdefinitionen und Funktionsdeklarationen für die Verarbeitung von Matrizen:

```
typedef double Element;

typedef Element *Row;
typedef Row *Rows;

typedef struct
{
  Rows rows;
  Element *elems;
  int width;
  int height;
} *Matrix;

/* constructor functions */

extern Matrix newMatrix (int w, int h);
extern Matrix zeroMatrix (int w, int h);
extern Matrix unitMatrix (int w, int h);

/* destructor */

extern void freeMatrix (Matrix m);

/* matrix ops */

extern Matrix addMatrix (Matrix m1, Matrix m2);
extern Matrix transposeMatrix (Matrix m);

/* element access ops */

extern Element at (Matrix m, int i, int j);
extern Matrix setAt (Matrix m, int i, int j, Element v);
```

Die *newMatrix*–Funktion sei wie folgt implementiert:

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

/*—————————*/

Matrix
newMatrix (int w, int h)
{
  Matrix res = malloc (sizeof (*res));
  if (res)
    {
      res->rows = malloc (h * sizeof (Row));
      res->elems = malloc (h * w * sizeof (Element));
      res->width = w;
      res->height = h;
      {
        Rows rs = res->rows;
        Row r = res->elems;
        if (rs && r)
          {
            while (h--)
              {
                *rs++ = r;
                r += w;
              }
            return res;
          }
      }
    }
  /* heap overflow */
  perror ("newMatrix:  can't allocate matix");
  exit (1);
}
```

Wie viel Platz wird für eine Matrix der Größe $m*n$ mit einem Aufruf von $newMatrix(m, n)$ auf der Halde alloziiert?

. . . . . .

Implementieren Sie die Routinen *at* und *setAt* für den lesenden und schreibenden indizierten Zugriff, und zwar so, dass eine Indexüberprüfung mit *assert* vorgenommen wird.

Die Funtion *at*:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Die Funktion *setAt* soll einen Wert v an einer Stelle in der Matrix setzen und die veränderte Matrix als Wert zurückgeben. Die Funktion *setAt*:

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Entwickeln Sie die Funktion *zeroMatrix*, und zwar so, dass keine geschachtelten Schleifen benötigt werden. *zeroMatrix* soll die Matrix mit 0-en initialisieren.

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

. . . . . .

Aufgabe 2:

Seien $f_1, f_2, f_3, g_1, g_2, g_3$ 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 $i \in \{1, 2, 3\}$.

1. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) - f_2(n)$

   .....

2. Aus welcher Komplexitätsklasse ist $f(n) = f_1(n) * f_2(n) * f_3(n)$

   .....

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

   .....

4. Aus welcher Komplexitätsklasse ist $f(n) = c_1 * n^2 + c_2 * n * \log n + c_3 * n$

   .....

Aufgabe 3:

Rot–Schwarz–Bäume sind binäre Suchbäume, die gewisse Ausgewogenheitskriterien erfüllen müssen. Diese Bedingungen werden mit Hilfe sogenannter 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*: