Prof. Dr. Uwe Schmidt

ptlwedel
PHYSIKALISCH-TECHNISCHE LEHRANSTALT

10. August 2005

Aufgaben zur Klausur **C** im SS 2005 (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 8 Seiten

---

Aufgabe 1:

In dieser Aufgabe geht es darum, die Laufzeit von Operationen für verschiedene Datenstrukturen abzuschätzen. Die Laufzeit von Operationen auf Listen und Bäumen hängt üblicherweise von der Anzahl der Elemente in einer Liste oder in einem Baum ab. In dieser Aufgabe sollen $n, n_1, n_2, ...$ die Anzahl der Elemente in den Listen $l, l_1, l_2, ...$ oder Bäumen $b, b_1, b_2, ...$ bezeichnen. Verwenden Sie bitte die *Groß–O*–Notation.

1. Laufzeitkomplexität für das Einfügen eines Elementes e in eine unsortierte, einfach verkettete Liste mit Duplikaten: $insert(e, l)$

   .....

2. Laufzeitkomplexität für das Einfügen eines Elementes e in eine unsortierte, einfach verkettete Liste ohne Duplikate: $insert(e, l)$

   .....

3. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete Liste mit Duplikaten: $insert(e, l)$

   .....

4. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete Liste ohne Duplikate: $insert(e, l)$

   .....

5. Laufzeitkomplexität für das Anhängen eines Elementes e am Ende einer einfach verketteten Liste: $append(l, e)$

   .....

6. Laufzeitkomplexität für das Konkatenieren zweier einfach verketteter Listen: $concat(l_1, l_2)$

   .....

7. Laufzeitkomplexität für das Mischen aller Elemente zweier unsortierter, einfach verketteter Listen ohne Duplikate: $merge(l_1, l_2)$

   .....

8. Laufzeitkomplexität für das Mischen aller Elemente zweier sortierter, einfach verketteter Listen ohne Duplikate: $merge(l_1, l_2)$

   .....

9. Laufzeitkomplexität für das Suchen eines Elementes e in einer sortierten, einfach verketteten Liste mit Duplikaten: $isIn(e, l)$

. . . . . .

10. Laufzeitkomplexität im Mittel für das Suchen eines Elementes e in einem binären Suchbaum: $isIn(e, b)$

. . . . . .

11. Laufzeitkomplexität im schlechtesten Fall für das Suchen eines Elementes e in einem binären Suchbaum: $isIn(e, b)$

. . . . . .

12. Laufzeitkomplexität im Mittel für das Einfügen eines Elementes e in einen binären Suchbaum: $insert(e, b)$

. . . . . .

13. Laufzeitkomplexität im schlechtesten Fall für das Einfügen eines Elementes e in einen binären Suchbaum: $insert(e, b)$

. . . . . .

14. Laufzeitkomplexität für das Anhängen eines Elementes e am Ende einer einfach verketteten, als Ring implementierten Liste: $append(l, e)$

. . . . . .

15. Laufzeitkomplexität für das Konkatenieren zweier einfach verketteter, als Ring implementierten Listen: $concat(l_1, l_2)$

. . . . . .

16. Laufzeitkomplexität für das Einfügen eines Elementes e in eine sortierte, einfach verkettete, als Ring implementierte Liste: $insert(e, l)$

. . . . . .

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 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

      ..........................................
}
```