Prof. Dr. Uwe Schmidt

ptlwedel
PRIVATE BERUFSFACHSCHULE

1. Februar 2011

Aufgaben zur Klausur **C** im WS 2010/11 (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.

Tipp: Bei der Entwicklung der Lösung kann einen kleine Skizze manchmal hilfreich sein.

Viel Erfolg!

Diese Klausur besteht einschließlich dieses Deckblattes aus 8 Seiten.

Aufgabe 1:

In der Vorlesung sind mehrere Datenstrukturen zur Laufzeit- und Speicherplatz-effizienten Implementierung von Mengen von ganzen Zahlen entwickelt worden. In dieser Aufgabe soll eine weitere Möglichkeit teilweise entwickelt werden. Eine Menge von ganzen Zahlen kann in zusammenhängende Bereiche (Intervalle) aufgeteilt werden. Diese Intervalle können in einer sortierten, einfach verketteten Liste gespeichert werden. So kann die Menge $m_1 = \{1, 2, 3, 5, 7, 8, 9\}$ durch eine dreielementige Liste von Bereichen $l_1 = [1..3, 5..5, 7..9]$ repräsentiert werden.

Für eine effiziente Implementierung ist zu beachten, dass immer mit der minimalen Anzahl von Intervallen gearbeitet wird. So sollte nach dem Einfügen des Wertes 4 in die Menge m_1 eine neue Liste l_2 mit nur noch 2 Intervallen entstehen. Wird dann anschließend noch die 6 eingefügt, genügt ein Intervall zur Darstellung des Resultats.

Die folgende C Header Datei enthält Deklarationen für eine solche Implementierung.

```
typedef struct {
  int lb;
  int ub;
} Range;

typedef struct node * IntSet;

struct node {
  Range info;
  IntSet next;
};

#define isEmpty(s) ((s) == (IntSet)0)

extern int min(int i, int j);
extern int max(int i, int j);
extern unsigned int length(IntSet s);
extern unsigned int card(IntSet s);

extern int overlap(Range e1, Range e2);

extern int invIntSet(IntSet s);

extern int isIn(int e, IntSet s);
extern IntSet union1(IntSet s1, IntSet s2);
extern IntSet insert(int e, IntSet s);
extern IntSet mk1Range(int lb, int ub);
```

Die hier eingeführten Größen sind bei der Lösung der folgenden Aufgaben zu verwenden.

Einige einfache Funktionen sind wie folgt implementiert:

```
#include "IntSet.h"

int max(int i, int j) {
   return (i > j) ? i : j;
}
int min(int i, int j) {
   return (i < j) ? i : j;
}
unsigned int length(IntSet s) {
   return (isEmpty(s)) ? 0 : 1 + length(s->next);
}
unsigned int card(IntSet s) {
   return (isEmpty(s)) ? 0 :
      (s->info.ub - s->info.lb + 1) + card(s->next);
}
```

Implementieren Sie als erstes eine Hilfsfunktion *overlap*. In dieser soll getestet werden, ob zwei Intervalle sich überlappen oder aneinander stoßen.

Die *overlap* Funktion:

```
#include "IntSet.h"

int overlap(Range e1, Range e2) {

   .....

   .....

   .....

   .....

   .....

   .....
}
```

Im nächsten Schritt soll die Datenstruktur-Invariante für diese Implementierung realisiert werden.

```
#include "IntSet.h"

int invIntSet(IntSet s) {
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
    ..........
}
```

Weiter ist eine Konstruktorfunktion *mk1Range* zu implementieren:

```
#include "IntSet.h"
#include <stdlib.h>

IntSet mk1Range(int lb, int ub) {
    IntSet res =

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................
   return res;
}
```

und der Test, ob ein Element in der Menge enthalten ist:

```
#include "IntSet.h"

int isIn(int e, IntSet s) {

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

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

Der interessante Teil dieser Implementierung ist die Mengenvereinigung. Hierbei müssen 2 sortierte Listen von Intervallen gemischt werden und sich überlappende oder aneinander angrenzende Intervalle zu einem Intervall verschmolzen werden. Außerdem muss der Speicher für nicht mehr verwendete Bereiche freigegeben werden. Tipp: Die Hilfsfunktionen können hier gewinnbringend eingesetzt werden.

```
#include "IntSet.h"
#include <stdlib.h>

IntSet union1(IntSet s1, IntSet s2) {

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

   ..........................................................

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

Als letzte Funktion etwas zur Entspannung (`:-)`), das Einfügen eines Elements in eine Menge:

```
#include "IntSet.h"

IntSet insert(int e, IntSet s) {

   ..................................................

   ..................................................

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

Mit welcher Zeitkomplexität arbeitet das Suchen in einer Menge s?
n sei dabei die Anzahl der Elemente in der Menge s.

. . . . . .

Mit welcher Zeitkomplexität arbeitet die Mengenvereinigung?
n_1 und n_2 seien dabei die Anzahl der Elemente in den beiden zu vereinigenden Mengen s_1 und s_2.

. . . . . .

Mit welcher Zeitkomplexität arbeitet das Einfügen in eine Menge s?
n sei dabei die Anzahl der Elemente in der Menge s.

. . . . . .

Wäre es möglich, anstatt einer sortierten Liste für die Speicherung der Intervalle einen binären Suchbaum zu verwenden und so die Laufzeit zu verbessern? Wenn ja, welche Laufzeit könnte für das Suchen erreicht werden?

. . . . . .

. . . . . .

Die Speicherplatzeffizienz einer Mengenimplementierung kann abgeschätzt werden, indem berechnet wird, wieviel Speicherzellen pro Element benötigt werden. Die Anzahl der Speicherzellen für eine Variable oder einen Datentyp kann in C sehr einfach berechnet werden.

Entwickeln Sie eine Funktion *sizePerValue*, mit der berechnet wird, wieviel Speicher pro Element für ein Menge benötigt wird.

```
#include "IntSet.h"

double sizePerValue(IntSet s) {

   .....

   .....

   .....

   .....

   .....

   .....
}
```
