Prof. Dr. Uwe Schmidt

ptlwedel
PRIVATE BERUFSFACHSCHULE

31. Januar 2013

Aufgaben zur Klausur **C** im WS 2012/13 (IA 302)

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 11 Seiten.

Aufgabe 1:

Gegeben sei das folgende C–Programmstück zur Berechnung und Verarbeitung von Bitstrings und Bit-Masken.

```
#include <limits.h>
#include <assert.h>

typedef unsigned long int BS;

#define LEN_BS (CHAR_BIT * sizeof (BS))

typedef BS Mask;

#define emptyBS ((BS)0)

#define singleton(i) ((BS)1 << (i))

int invMask(Mask m) {
  return
    m != emptyBS
    &&
    (m ^ (m & (~m + 1))) == emptyBS;
}
```

Ein Bitstring (BS) wird als vorzeichenlose lange ganze Zahl repräsentiert. Eine Maske (Mask) ist ein Bitstring, in dem genau ein Bit gesetzt ist (*invMask*). Die Länge des Bitstrings wird mit Hilfe vordefinierter Makros aus *limits.h* fest gelegt.

Tipp: Entwickeln sie die Code-Stücke für die Operationen auf diesen Datentypen auf den Rückseiten der Klausur-Blätter und übertragen die fertigen Lösungen an die gekennzeichneten Stellen.

Implementieren Sie als erstes eine Funktion *onesBS*, die zu einem beliebigen Bitstring, in dem mindestens ein Bit gesetzt ist, einen Bitstring berechnet, bei dem alle Bits vom höchstwertigen gesetzten Bit bis zum niederwertigsten Bit (gerade/ungerade-Bit) gesetzt sind. Beispiel: Alle Bitstring-Werte zwischen 128 und 255 werden auf den Wert 255 abgebildet. Für den Wert *emptyBS* soll die Berechnung abgebrochen werden.

```
BS onesBS(BS bs) {
    ..................................................
    ..................................................
    ..................................................
    ..................................................
    ..................................................
    ..................................................
    ..................................................
   return bs;
}
```

Implementieren Sie eine Funktion *maskBS*, mit der aus einem Bitstring ungleich *emptyBS* eine Bit-Maske berechnet wird, in der genau nur das höchstwertige Bit gesetzt ist. Für den Wert *emptyBS* soll die Berechnung wie oben abgebrochen werden. Tipp: Vermeiden Sie Schleifen.

```
Mask maskBS(BS bs) {
    ..................................................
    ..................................................
    ..................................................
}
```

Implementieren Sie eine Funktion *commonPrefixMask*, mit der zu zwei beliebigen unterschiedlichen Bitstrings eine Bit-Maske berechnet wird, die anzeigt, welches das höchstwertige Bit ist, in dem die Bitstrings sich unterscheiden. Sind die Argumente gleich, soll die Berechnung abgebrochen werden. Tipp: Vermeiden Sie Schleifen.

```
Mask commonPrefixMask(BS bs1, BS bs2) {
    ..................................................
    ..................................................
    ..................................................
}
```

Implementieren Sie eine Funktion *getPrefix*, mit der aus einem beliebigen Bitstring und einer Maske m ein Bitstring berechnet wird, in dem alle niederwertigen Bits ab dem in der Maske m gesetzten Bit (und einschließlich dieses Bits) gelöscht werden. Wird die Funktion mit nicht erlaubten Argumenten aufgerufen, so soll die Berechnung abgebrochen werden. Tipp: Vermeiden Sie Schleifen.

```
BS getPrefix(BS bs, Mask m) {
    ..................................................
    ..................................................
    ..................................................
}
```

Aufgabe 2:

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.

```
#include <stdlib.h>
#include <assert-h>
typedef struct {
  int lb;
  int ub;
} Range;
typedef struct node * IntSet;
struct node {
  Range info;
  IntSet next;
};
#define isEmpty(s) ((s) == (IntSet)0)
#define mkEmpty() ((IntSet)0)
extern int min(int i, int j);
extern int max(int i, int j);

extern int isEmptyRange(Range r);
extern int less(Range r1, Range r2);
extern int overlap(Range r1, Range r2);
extern Range mkRange(int lb, int ub);
extern Range unionRange(Range r1, Range r2);

extern unsigned int length(IntSet s);
extern unsigned int card(IntSet s);

extern int invIntSet(IntSet s);
extern int isIn(int e, IntSet s);

extern IntSet mk1Set(Range r);
extern IntSet insertRange(Range r, IntSet s);
extern IntSet insertInt(int e, IntSet s);
extern IntSet unionIntSet(IntSet s1, IntSet s2);
```

Die im Headerfile eingeführten Größen sind bei der Entwicklung der folgenden Routinen 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;
}
Range mkRange(int lb, int ub) {
   Range res = {lb, ub};
   return res;
}
int isEmptyRange(Range r) {
   return r.lb > r.ub;
}
int less(Range r1, Range r2) {
   return r1.ub < r2.lb;
}
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);
}
```

Vorsicht: Aufrufe von *mkRange* in der Form *mkRange(5,4)* sind möglich, solche Intervalle sind aber in *IntSet*-Werten nicht sinnvoll.

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

Die *overlap* Funktion:

```
#include "IntSet.h"

int overlap(Range e1, Range e2) {

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

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

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

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

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

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

Eine weitere nützliche Hilfsfunktion ist *unionRange* zum Verschmelzen zweier Intervalle:

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

Range unionRange(Range r1, Range r2) {
   assert(! isEmptyRange(r1));
   assert(! isEmptyRange(r2));
   assert(overlap(r1, r2));

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

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

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

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

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

```
#include "IntSet.h"

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

Weiter ist eine Konstruktorfunktion *mk1Set* zu implementieren. Zu beachten ist, dass für das Resultat die Invariante gelten muss.

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

IntSet mk1Set(Range r) {
  IntSet res;

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

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

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

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

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

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

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

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 das Einfügen eines Intervalls und die Mengenvereinigung. Das Einfügen eines Intervalls wird durch *insertRange* implementiert:

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

IntSet insertRange(Range r, IntSet s) {

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

   ..........

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

Tipp: Achten Sie auf mögliche Speicherfreigaben.

Für das Hinzufügen eines Elementes gibt es die sehr einfach zu realisierende Funktion *insertInt*:

```
#include "IntSet.h"

IntSet insertInt(int e, IntSet s) {

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

Bei der Mengenvereinigung 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 bisherigen Funktionen können hier gewinnbringend eingesetzt werden.

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

IntSet unionIntSet(IntSet s1, IntSet s2) {
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
  ..........................................................
}
```
