S. Kurtz, T. Schwabe,
G. Gonnella, D. Willrodt

Universität Hamburg
Zentrum für Bioinformatik

Programmierung für Naturwissenschaften
Sommersemester 2015
Übungen zur Vorlesung: Ausgabe am 07.05.2015

Punkteverteilung: Aufgabe 6.1: 5 Punkte, Aufgabe 6.2: 2 Punkte, Aufgabe 6.2: 5 Punkte

Bitte lösen Sie Aufgabe 1 zuerst, anschließend kopieren Sie für Aufgabe 2 den Ordner von Aufgabe 1 und ergänzen diesen. Eventuelle Fehler aus der Aufgabe 1 werden natürlich nur einmal bewertet.

Es gibt dieses mal etwas mehr Punkte, dafür haben Sie auch 2 Wochen Zeit die Aufgaben zu lösen.

Abgabe bis zum 20.05.2015, 10:00 Uhr.

Aufgabe 6.1 In dieser Aufgabe sollen Sie eine alternative Implementierung des Datentyps `IntSet` (aus einer vorherigen Aufgabe) zur Repräsentation einer geordneten Menge von nicht-negativen ganzen Zahlen $\leq m$ erstellen. Dabei soll die folgende bekannte Eigenschaft ausgenutzt werden:

> Für jede ganze Zahl p und jede positive ganze Zahl d, gibt es zwei eindeutig bestimmte ganze Zahlen q und r, so dass $p = d \cdot q + r$ mit $0 \leq r < d$.

$q = p \operatorname{div} d$ ist also der ganzzahlige Quotient und $r = p \bmod d$ der Rest der ganzzahligen Teilung. Die Idee ist es nun, die ganzzahligen Reste der Elemente der einzufügenden Zahlen in einem Array zu speichern und sich für alle möglichen Quotienten q zu merken, in welchen Bereichen die Restwerte der Zahlen p stehen, für die $p \operatorname{div} d = q$ gilt. Sei $m =$ `maxvalue`. Da m eine obere Schranke für die einzufügenden Elemente darstellt, ist der Quotient höchstens $\frac{m}{d}$.

Um die Divisions- und Modulo-Operation effizient durchführen zu können, wählt man d als Zweier-Potenz. In Ihrer Implementierung soll $d = 2^{16} = 65536$ sein. Dann sind die Restwerte kleiner als 2^{16} und es reicht ein Array $elements$ der Länge $\ell =$ `nofelements` über dem Basistyp `uint16_t` aus, um die Restwerte zu speichern. Das Typsynonym `uint16_t` ist in `inttypes.h` definiert.

Das Array $elements$ ist nun virtuell unterteilt in $\frac{m}{d}+1$ Abschnitte. Nach dem Einfügen aller Elemente $p_0, p_1, \ldots, p_{\ell-1}$ in sortierter Reihenfolge enthält für alle $q, 0 \leq q \leq \frac{m}{d}$ der q-te Abschnitt die Reste der Werte p_j mit $q = p_j \operatorname{div} d$. Um die Abschnitte zu identifizieren, braucht man ein zweites Array $sectionstart$ der Länge $\frac{m}{d} + 2$ über dem Basistyp `unsigned long`. Für alle q, $0 \leq q \leq \frac{m}{d} + 1$ ist

$$sectionstart[q] = \min\left(\{\ell\} \cup \{j \mid 0 \leq j < \ell, q \cdot d \leq p_j\}\right).$$

Falls $sectionstart[q] < sectionstart[q+1]$ ist, dann ist der q-te Abschnitt von $elements$ nicht leer und liegt im Indexbereich $sectionstart[q], \ldots, sectionstart[q+1] - 1$. Falls $elements[j]$ zum q-ten Abschnitt von $elements$ gehört, ist der j-te ursprünglich eingefügte Wert $p_j = d \cdot q + elements[j]$.

Beispiel: Wir betrachten 10 positive ganze Zahlen, die in der folgenden Tabelle in der Zeile *original* dargestellt sind und von $j \in \{0, \ldots, 9\}$ indiziert werden. In der Zeile $elements$ findet man die Restwerte zu den darüber stehenden Originalwerten. Die letzte Zeile zeigt die jeweiligen Quotienten bzgl. $d = 2^{16}$. Es gibt zwei Abschnitte, die aus mehr als einer Zahl bestehen, nämlich der Abschnitt für den Quotienten 5 (grau) und der Abschnitt für den Quotienten 9 (hellgrau).

j	0	1	2	3	4	5	6	7	8	9
original (p_j)	48747	124290	185664	231747	343991	360151	595178	628803	824326	901437
elements	48747	58754	54592	35139	16311	32471	5354	38979	37894	49469
quotients	0	1	2	3	5	5	9	9	12	13

Da 901437 der maximale Wert ist, hat das Array $sectionstart$ $901437/d + 2 = 15$ Einträge, die in der folgenden Tabelle gezeigt sind. Die letzte Zeile zeigt die Länge der einzelnen Abschnitte. So haben die Abschnitte für $q = 5$ und für $q = 9$ jeweils die Länge 2.

q	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$sectionstart[q]$	0	1	2	3	4	4	6	6	6	6	8	8	8	9	10
$sectionlength[q]$	1	1	1	1	0	2	0	0	0	2	0	0	1	1	

Implementieren Sie nun in einer Datei `intset.c` auf der Basis der hier beschriebenen Ideen eine alternative Repräsentation `IntSet` von geordneten Mengen von nicht negativen ganzen Zahlen. Die zu implementierenden Funktionen sollen so benannt werden wie in der entsprechenden voherigen Aufgabe und sie sollen das Gleiche leisten.

Geben Sie in Abhängigkeit von m (Maximalwert) und ℓ (Anzahl der Werte) an, unter welcher Bedingung obige Repräsentation weniger Speicher benötigt als die in der vorherigen Aufgabe implementierte Repräsentation.

Aufgabe 6.2 In einer vorherigen Aufgabe wurde eine effiziente Repräsentation des Datentyps `IntSet` für eine Menge S von Werten zwischen 0 und m realisiert. Implementieren Sie auf der Basis von `IntSet` die folgende Funktion:

```
unsigned long intset_number_next_larger(const IntSet *intset,
                                        unsigned long value);
```

Sei `value` ein Wert zwischen 0 und m, der nicht in S vorkommt. Die genannte Funktion liefert für `value` den Index q des kleinsten Elementes in S, das echt grösser ist als `value`. Wenn es ein solches Element nicht gibt, dann wird `numofelems` zurückgegeben. Die Funktion soll zunächst durch entsprechende *assertions* für `value` die genannten Bedingungen überprüfen. Dann wird der Abschnitt ermittelt, zu dem `value` gehört und schliesslich mit dem Restwert von `value` auf den Restwerten, die zu diesem Abschnitt gehören, eine modifizierte binäre Suche durchführt.

Die folgende Tabelle gibt für die Beispielmenge aus der vorherigen Aufgabe und einige Werte von `value` die entsprechenden Abschnitte und die Ergebnisse an:

`value`	0	4	52342	274565	637383	800000
Abschnitt	0	0	0	4	9	12
Ergebnis	0	0	1	4	8	8

Verwenden Sie die gleichen Materialien wie zur vorherigen `IntSet`-Aufgabe, sowie Ihren bereits geschriebenen Code.

Fügen Sie in der Datei `intset-run.sh` in der Zeile `./intset.x ${maxvalue} ${numofelems}` nach `./intset.x` den String `-s` ein. Damit werden auch die Ergebnisse von `intset_number_next_larger` verifiziert.

Führen sie dann wie gewohnt `make test` aus.

Der Zweck dieser Funktion, bezogen auf die Speicherung von Separatorpositionen einer Konkatenation von Sequenzen (siehe erste `Intset`-Aufgabe), ist die Ermittlung einer Sequenznummer zu einer gegebenen Position, die hier als `value` übergeben wird.

Aufgabe 6.3 Zur sortierten Speicherung von Daten haben wir in einer vorherigen Aufgabe doppelt verlinkte Listen kennen gelernt. Eine weitere, effizientere, Methode zur Speicherung von Daten sind Bäume, z.B. Rot-Schwarz-Bäume oder unbalancierte Binärbäume. Die Suche nach einem bestimmten Element in diesen Strukturen wird durch die Baumstruktur beschleunigt.

Ihre Aufgabe ist es einen generellen sortierten Binärbaum zu implementieren. Da dieser Baum beliebige Elemente enthalten soll, enthalten die Knoten des Baums nur einen `void`-Zeiger auf die zu speichernden Werte. Damit der Größenvergleich der Elemente funktioniert, enthält die Baumstruktur einen Zeiger auf eine Vergleichsfunktion `cmp_node_value`.

Duplikate werden durch die Funktion `combine_node_value` abgefangen. Diese Funktion könnte z.B. eine Fehlermeldung erzeugen, oder wie im Beispiel-Code einen Zähler aktualisieren.

Das Löschen eines Elementes aus dem Baum gibt den Speicher des darin enthaltenen Wertes mit hilfe der Funktion `free_node_value` wieder frei.

Soll der gesamte binäre Baum freigegeben werden, wird diese Funktion ebenfalls verwendet. Sie gibt den Speicher des Inhaltes frei, bevor der Speicher der einzelnen Knoten frei gegeben wird.

Nutzen Sie folgende Dateien und implementieren Sie alle Funktionen:

bintree.h

```
#ifndef BINTREE_H
#define BINTREE_H
#include <stdbool.h>

/* The general Binary Tree class, used to store arbitrary objects in a sorted
   (log(n) access) manner. */
typedef struct GenBinTree GenBinTree;

/* Funktion used by <GenBinTree> to compare two objects. Return values should be
   as follows
      < 0 if first value is smaller than second
      > 0 if first value is larger than second
      0 if the values are considered equal */
typedef int (*Cmpfunction)(const void *, const void *);

/* Funktion used by <GenBinTree> to handle duplicate entries. First parameter
   <old> will be the value stored in the <GenBinTree>, second <new> will be the
   value to add. */
typedef void (*Combinefunction)(void *old, const void *new);

/* Funktion used by <GenBinTree> to free objects stored within a node */
typedef void (*Freefunction)(void *);

/* Funktion called for every object stored within the nodes of a <GenBinTree>
   when gbt_enumvalues is called */
typedef void (*Applyfunction)(const void *, void *);

/* Return a pointer to a new <GenBinTree> object. <cmp_node_value> will be used
   to compare and sort the values to be stored in the <GenBinTree>,
   <combine_node_value> is called if <cmp_node_value> indicates a duplicate
   value during a gbt_add()-call, <free_value> will be called for every value in
```

```
   the <GenBinTree> if it is deleted. */
GenBinTree *gbt_new(Cmpfunction cmp_node_value,
                    Combinefunction combine_node_value,
                    Freefunction free_node_value);

/* add a new element to binary tree, uses cmp_node_value. Return
   true if node corresponding to value was added. Otherwise
   combine_node_value function was called and the gtb_add returns false */
bool gbt_add(GenBinTree *bintree,void *new_value);

/* free the memory of the whole tree, also of the values inside the nodes,
 * uses free_node_value */
void gbt_delete(GenBinTree *bintree);

/* iterate in sorted order over the values stored in the binary tree.
   Apply the function apply_node_value to the values. data is supplied
   as second argument to this function. */
void gbt_enumvalues(const GenBinTree *bintree,Applyfunction apply_node_value,
                    void *data);

#endif
```

`bintree.c(.template)` Vervollständigen Sie die Datei `bintree.c.template` und benennen Sie diese um:

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

#include "bintree.h"

struct bin_tree_node {
  void *value;
  BinTreeNode *left,
              *right;
};

struct generic_bin_tree {
  BinTreeNode *t_root;
  Cmpfunction *cmp_node_value; /* pointer to compare function, returns
                                  < 0 if value1 < value2,
                                    0 if value1==value2 * and
                                  > 0 if value1 > value2 */
  Combinefunction combine_node_value; /* pointer to function that handles
                                         duplicate values */
  Freefunction *free_node_value; /* pointer to Funktion that frees the contents
                                    of a node pointed to by value */
};
```

`wordstat.c` Diese Datei muss nicht mehr vervollständigt werden, sie wird für den Test benötigt. Schauen Sie sich den Code aber mal an, besonders die Funktionen, die an den Bintree übergeben werden.

Anschließend benötigen Sie noch den Tokenizer aus einer der vorheringen Aufgaben, damit das Testprogramm compilieren kann. Kopieren Sie dazu einfach ihre (möglicherweise korrigierten) `tokenizer.[ch]`-Dateien in den Ordner zu dieser Aufgabe.

Das Mitgelieferte Makefile und `make test` sollten dann den Code erfolgreich testen. Vergessen Sie

nicht auch mit Valgrind zu testen, ob ihr Code Fehlerfrei läuft.

Die Lösungen zu diesen Aufgaben werden am 21.05.2015 besprochen.