Lösung der 7. Übung – Informatik I für VIW

Fakultät Verkehrswissenschaften
Fachrichtung Verkehrsingenieurwesen

Zeitraum: 24.01.2011 bis 04.02.2011 (WS 2010/11)

Aufgabe 1

(a)

```
wurzel = &s4;
s4.next = &s2;
s2.next = &s3;
s3.next = &s1;
s1.next = &s5;
s5.next = NULL;
```

(b)

```
wurzel = &s1;
s5.next = &s4;
s3.next = NULL;
```

(c)

```
struct el *neu = (struct el *) malloc (sizeof(struct el));
neu->wert = 6;
neu->next = &s2;
s4.next = neu;
```

Aufgabe 2

(a)

```
struct datum {int tag, monat, jahr; struct datum *next;};
```

(b) Als erstes sollte man die Jahreszahlen vergleichen. Sind diese unterschiedlich, dann weiß man sofort, welches Datum vor dem anderen liegt. Sind die Jahreszahlen jedoch gleich, sollte man im nächsten Schritt die Monate vergleichen. Wieder gilt: sind diese unterschiedlich, dann kann man auch wieder sofort die Reihenfolge der beiden Datumsangaben bestimmen. Bei Gleichheit von Jahr und Monat bestimmt man die Reihenfolge der beiden Angaben über den Tag.

Die folgende Funktion macht sich diese Vorgehensweise zu Nutze.

```
int vergleiche(struct datum *datum1, struct datum *datum2) {
   if (datum1->jahr < datum2->jahr)
      return 1;
   if (datum1->jahr > datum2->jahr)
      return 0;

   if (datum1->monat < datum2->monat)
      return 1;
   if (datum1->monat > datum2->monat)
      return 0;

   if (datum1->tag <= datum2->tag)
      return 1;

   return 0;
}
```

(c) Die folgende Funktion arbeitet mit einer rekursiven Lösung. Eine Liste ist sortiert, wenn sie leer ist oder aus nur einem Element besteht (siehe die ersten beiden If-Statements).

Besteht die Liste aus mehr als einem Element, dann ist sie sortiert, wenn das erste Element kleiner oder gleich dem zweiten Element ist und wenn die Restliste (also die Liste, die bei dem zweiten Listeneintrag beginnt) ebenfalls sortiert ist.

```
int istSortiert(struct datum *wurzel) {
   if (wurzel == NULL)
      return 1;
   if (wurzel->next == NULL)
      return 1;

   if ( vergleiche(wurzel, wurzel->next) == 0 )
      return 0;

   /*Wenn wir hier sind, dann ist das erste Datum kleiner oder gleich
   dem zweiten Datum. Die Liste ist jetzt sortiert genau dann wenn die
   Restliste sortiert ist.*/

   return istSortiert(wurzel->next);
}
```

(d) Die folgendes Funktion sucht zuerst nach dem kleinsten Datum und gibt am Ende die Jahresangabe des kleinsten Datums zurück. Sie sucht das kleinste Datum sucht mit Hilfe einer For-Schleife, in der sie sich das bisher kleinste Element mit Hilfe des Pointers `kleinstes` merkt. Die aktuelle Position, in der sie sich beim Durchsuchen der Liste befindet, merkt sie sich mit dem Pointer `aktuelles`.

Eine Besonderheit ist, dass zum Anfang der Minimum-Suche das erste Datum in der Liste zunächst als das kleinste Datum angesehen wird (deshalb der Befehl `kleinstes = wurzel`). Bei der Minimum-Suche braucht sie deshalb nur beim zweiten Listenelement zu beginnen (deshalb wird `aktuelles` mit `wurzel->next` initialisiert). In der Schleife wird die aktuelle Datumsangabe mit der bisher kleinsten gefundenen verglichen und bei Bedarf das kleinste Datum neu definiert.

```
int kleinstesDatum(struct datum *wurzel) {
   struct datum *aktuelles, *kleinstes;

   if (wurzel == NULL)
      return -1;

   kleinstes = wurzel;

   for (aktuelles = wurzel->next;
        aktuelles != NULL;
        aktuelles = aktuelles->next) {
      if ( vergleiche(kleinstes, aktuelles) == 0 )
        kleinstes = aktuelles;
   }

   return kleinstes->jahr;
}
```

Aufgabe 3

(a)

```
struct el s1,s2,s3,s4;
struct list myList;

s1.wert = 4; s2.wert = 2; s3.wert = 5; s4.wert = 6;
s1.next = &s2; s2.next = &s3; s3.next = &s4; s4.next = NULL;
myList.laenge = 4;
myList.wurzel = &s1;
```

(b)

```
void einfuegen(struct list *lp, int zahl) {
   struct el *neu = (struct el *) malloc(sizeof(struct el));

   neu->next = lp->wurzel;
   neu->wert = zahl;
   lp->wurzel = neu;
   lp->laenge++;
}
```

Beispielaufruf: `einfuegen(&myList, 6);`

(c) Die folgende Funktion überprüft zunächst die beiden Sonderfälle, dass die Liste leer ist oder nur ein Element enthält. In beiden Fällen ist nichts zu tun.

In den drei Befehlen nach den beiden If-Statements wird zunächst ein Hilfspointer `erstes` definiert, der im gesamten restlichen Teil der Funktion auf das ursprünglich erste Listenelement verweist (dieser Eintrag soll dann ja ans Ende der Liste verschoben werden). Als nächstes wird die Wurzel verändert, so dass die Liste dann beim zweiten Listeneintrag beginnt, und danach wird der Next-Pointer des ursprünglich ersten Elementes so verändert, dass es keinen Nachfolgerelement mehr hat.

Als letztes muss nur noch der Next-Pointer des letzten Listenelementes so verändert werden, dass es auf den ursprünglich ersten Listeneintrag verweist. Dazu muss das letzte Element der Liste zuerst mit einer While-Schleife aufgefunden werden. Das letzte Element in der Liste zeichnet sich dadurch aus, dass sein Next-Pointer der Null-Pointer ist.

```
void anfangAnsEnde(struct list *lp) {
   struct el *erstes, *aktuelles;

   if (lp->wurzel == NULL)
      return;

   if (lp->wurzel->next == NULL)
      return;

   erstes = lp->wurzel
   lp->wurzel = erstes->next;
   erstes->next = NULL;

   aktuelles = lp->wurzel;
   while(aktuelles->next != NULL)
      aktuelles = aktuelles->next;

   aktuelles->next = erstes;
}
```

Beispielaufruf: `anfangAnsEnde(&myList);`

(d) Die folgende Funktion allokiert eine neue Struktur vom Typ `list`. Mit Hilfe der Funktion `einfuegen` füllt sie die neue Liste mit den Einträgen der Argumentliste (gegeben durch `lp`) auf. Da die Funktion `einfuegen` immer am Anfang einer Liste einfügt, genügt es, die Argumentliste einmal von vorn nach hinten zu durchlaufen und den jeweiligen Eintrag mit der Funktion `einfuegen` in die neue Liste einzutragen. Dadurch entsteht automatisch eine Liste mit umgekehrter Reihenfolge.

Damit das ganze korrekt funktioniert, muss die neue Liste am Anfang zu einer leeren Liste initialisiert werden (durch den Befehl `neu->wurzel = NULL;`). Die Länge der neuen

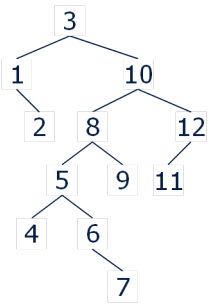
Liste ist die Länge der alten Liste. Am Ende wird der Zeiger auf die neue Listenstruktur zurückgegeben.

```
struct list *umdrehen(struct list *lp) {
   struct el *aktuelles;

   struct list *neu = (struct list *) malloc(sizeof(struct list));
   neu->laenge = 0;
   neu->wurzel = NULL;

   for (aktuelles = lp->wurzel;
        aktuelles != NULL;
        aktuelles = aktuelles->next) {
      einfuegen(neu, aktuelles->wert);
   }

   return neu;
}
```

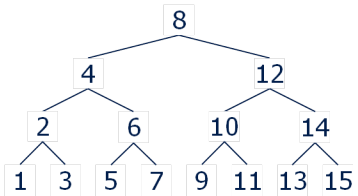
Aufgabe 4 (Zusatz)

(a) 6, 2, 4, 3, 9, 8, 4, 1, 3, 5, 7, 4, 2, 6, 9

(b)

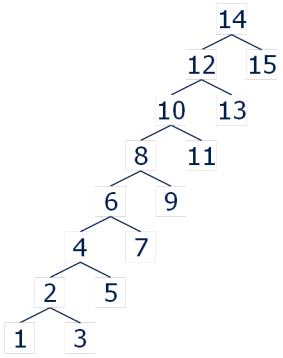


(c) Ein balancierter Baum entsteht zu Beispiel bei der Folge 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15.

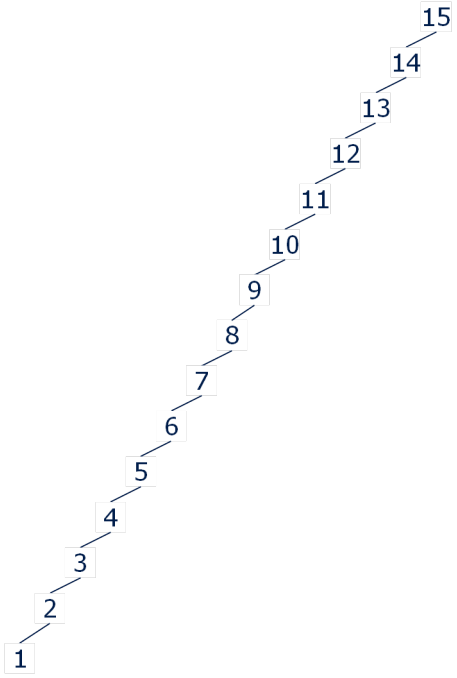


Ja! Auch die Folge 8, 12, 4, 10, 9, 14, 13, 2, 6, 5, 7, 3, 15, 1, 11 führt zu dem gleichen Suchbaum. Es gibt noch viele weitere.

Ein nach links abfallender Kamm entsteht zum Beispiel durch die Folge 14, 12, 15, 10, 11, 13, 8, 6, 4, 7, 9, 5, 2, 3, 1.



Ein noch extremerer Baum entsteht durch die Eingabefolge 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1



(d) Ausgabe der Folge:

```
void folge(struct knoten *wurzel) {
   if (wurzel == NULL)
      return;

   folge(wurzel->left);
   printf("%d, ", wurzel->wert);
   folge(wurzel->right);
}
```

Test auf Suchbaum; zwei Hilfsfunktionen `minimum` und `maximum` werden definiert:

```
int minimum(struct knoten *wurzel) {
   /*ermittelt den Minimalwert eines Suchbaumes;
   Achtung: die Eingabe muss ein Wurzelzeiger auf einen Suchbaum sein*/
   if (wurzel == NULL)
      return -1;
   if (wurzel->left == NULL)
      return wurzel->wert;
   return minimum(wurzel->left);
}

int maximum(struct knoten *wurzel) {
   /*ermittelt den Maximalwert eines Suchbaumes;
   Achtung: die Eingabe muss ein Wurzelzeiger auf einen Suchbaum sein*/
   if (wurzel == NULL)
      return -1;
   if (wurzel->right == NULL)
      return wurzel->wert;
   return maximum(wurzel->right);
}

int istSuchbaum(struct knoten *wurzel) {
   if (wurzel == NULL)
      return 1;

   if (istSuchbaum(wurzel->left) == 0)
      return 0;
   if (istSuchbaum(wurzel->right) == 0)
      return 0;

   if (wurzel->left != NULL && maximum(wurzel->left) > wurzel->wert)
      return 0;
   if (wurzel->right != NULL && minimum(wurzel->right) < wurzel->wert)
      return 0;

   return 1;
}
```