Prof. G. Zachmann
Dipl.-Inf. C. Giesemann

TU Clausthal
Institut für Informatik
27. Januar 2006

Wintersemester 2005/2006

Übungen zu Grundlagen der Programmierung in C - Blatt XI

Abgabe vom 1.2.2006 bis 7.2.2006 in der angemeldeten Übung

Aufgabe 1 (Doppelt verkettete Liste, *10 Punkte*)

Im Gegensatz zu der in der Vorlesung vorgestellten einfach verketteten Liste enthalten die Knoten einer *doppelt verketteten Liste* nicht nur einen Zeiger auf den nachfolgenden Knoten, sondern auch einen Zeiger auf den vorausgehenden Knoten.

In dieser Aufgabe soll der folgende Struct verwendet werden, welcher einen Knoten der Liste repräsentiert:

```
struct Node
{
    int data;
    Node* left;
    Node* right;
}
```

data ist ein im Knoten gespeicherter Datenwert. left ist der Zeiger auf den vorausgehenden Knoten der Liste, right ist der Zeiger auf den nachfolgenden Knoten.

Bei einer doppelt verketteten Liste ist der left-Zeiger des Listenkopfes (also desjenigen Elementes der Liste, welches wir als erstes Element auszeichnen) ein `NULL`-Pointer, ebenso ist auch der right-Zeiger des Listenendes ein `NULL`-Pointer, s. Abb. 1 .

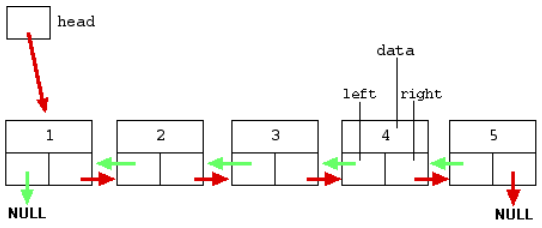

Abbildung 1: Eine doppelt-verkettete Liste

Grundlegende Operationen auf Listen

1. Schreiben Sie eine Funktion

   ```
   Node* newListNode( int data )
   ```

welche eine ein-elementige doppelt-verkettete Liste erzeugt (s. Abb. 2), und dessen Kopf (welcher der einzige Knoten dieser Liste ist) zurückgibt. Benutzen Sie dabei den **new**-Operator, um den Knoten der Liste dynamisch zu allozieren:

```
Node* n = new Node;
```

Der Funktionsparameter data bestimmt den Inhalt des Datenfeldes dieses Knotens.

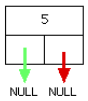

Abbildung 2: Eine ein-elementige doppelt-verkettete Liste

2. Schreiben Sie eine Funktion

   ```
   Node* appendLists( Node* listHead1, Node* listHead2 )
   ```

   welche zwei Listen hintereinanderhängt, deren Köpfe durch die Parameter listHead1 und listHead2 gegeben sind und einen Zeiger auf den Kopf der resultierenden zusammenhängenden Liste zurückgibt. Wird der Funktion ein NULL-Zeiger als einer der beiden Parameter übergeben, soll dies als eine leere Liste interpretiert werden.

3. Schreiben Sie eine Funktion

   ```
   void printList( Node* listHead )
   ```

   welche die in der verketteten Liste enthaltenen Datenwerte der Reihe nach auf der Standardausgabe ausgibt, beginnend mit dem als Parameter listHead übergebenen Listenkopf.

4. Schreiben Sie eine Funktion

   ```
   void printInversList( Node* listHead )
   ```

   welche die in der verketteten Liste enthaltenen Datenwerte in umgekehrter Reihefolge auf der Standardausgabe ausgibt, beginnend mit dem Datenwert des letzten Knotens der Liste. (Hinweis: Benutzen Sie vom Listenende aus die left-Zeiger um die Liste rückwärts zu durchlaufen.)

5. Schreiben Sie ein Hauptprogramm, welches von der Standardeingabe `Integer`-Werte einliest und dabei eine Liste aufbaut, welche diese Werte in der eingegebenen Reihenfolge enthält. Verwenden Sie dabei die Funktionen newListNode() und appendLists(), um neue Knoten an die Liste anzuhängen. Sobald eine negative Zahl eingelesen wurde, soll die Konstruktion der Liste abgebrochen und der Inhalt der Liste dann zuerst mittels der Funktion printList () wieder ausgegeben werden. (Hier sollen die eingelesenen Werte wieder in derselben Reihenfolge ausgegeben werden.) Danach soll der Inhalt der Liste mittels der Funktion printInversList () rückwärts ausgegeben werden. (Hier soll der zuletzt eingelesene Wert zuerst erscheinen und der zuerst eingelesene zuletzt ausgegeben werden.)
   Anm.: In dieser Aufgabe müssen Sie den allozierten Speicher nicht wieder freigegeben.

Aufgabe 2 (Zirkulär doppelt verkettete Liste, *10 Punkte*)

Eine Erweiterung der doppelt verketteten Liste ist die *zirkulär* doppelt verkettete Liste.

In dieser Aufgabe soll (wie in Aufgabe 1) der folgende Struct verwendet werden, welcher einen Knoten der Liste repräsentiert:

```
struct Node
{
    int data;
    Node* left;
    Node* right;
}
```

data ist ein im Knoten gespeicherter Datenwert. left ist der Zeiger auf den vorausgehenden Knoten der Liste, right der Zeiger auf den nachfolgenden Knoten.

In dieser Aufgabe soll eine *zirkulär* doppelt verkettete Liste verwendet werden, d.h. der left-Zeiger des Listenkopfes (also desjenigen Elementes der Liste, welches wir als erstes Element auszeichnen) zeigt auf das Listenende und der right-Zeiger des Listenendes auf den Listenkopf, s. Abb. 3 (Somit sind die left / right-Zeiger aller Knoten der Liste ungleich NULL.)

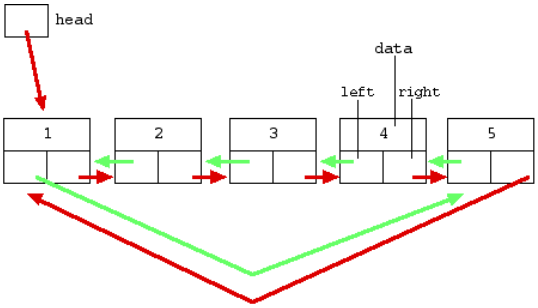

Abbildung 3: Eine zirkulär doppelt-verkettete Liste

a) Grundlegende Operationen auf Listen

1. Schreiben Sie eine Funktion

   `Node* newListNode( int data )`

   welche eine ein-elementige zirkulär doppelt-verkettete Liste erzeugt (s. Abb. 4), und dessen Kopf (welcher der einzige Knoten dieser Liste ist) zurückgibt. Benutzen Sie dabei den **new**-Operator, um den Knoten der Liste dynamisch zu allozieren:

   `Node* n = new Node;`

   Der Funktionsparameter data bestimmt den Inhalt des Datenfeldes dieses Knotens.

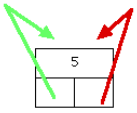

Abbildung 4: Eine ein-elementige zirkulär doppelt-verkettete Liste

2. Schreiben Sie eine Funktion

   `Node* appendLists( Node* listHead1, Node* listHead2 )`

   welche zwei Listen hintereinanderhängt, deren Köpfe durch die Parameter listHead1 und listHead2 gegeben sind, und einen Zeiger auf den Kopf der resultierenden zusammenhängenden Liste zurückgibt. Wird der Funktion ein NULL-Zeiger als einer der beiden Parameter übergeben, soll dies als eine leere Liste interpretiert werden.

3. Schreiben Sie eine Funktion

   ```
   void printList( Node* listHead )
   ```

   welche die in der verketteten Liste enthaltenen Datenwerte der Reihe nach auf der Standardausgabe ausgibt, beginnend mit dem als Parameter listHead übergebenen Listenkopf.

4. Schreiben Sie ein Hauptprogramm, welches von der Standardeingabe `Integer`-Werte einliest und dabei eine Liste aufbaut, welche diese Werte in der eingegebenen Reihenfolge enthält. Verwenden Sie dabei die Funktionen newListNode() und appendLists(), um neue Knoten an die Liste anzuhängen. Sobald eine negative Zahl eingelesen wurde, soll die Konstruktion der Liste abgebrochen und der Inhalt der Liste mittels der Funktion printList () wieder ausgegeben werden.

   Anm.: Auch in dieser Aufgabe müssen Sie den allozierten Speicher nicht wieder freigegeben.

b) Bubble Sort

Ergänzen Sie das Programm aus Aufgabenteil a) wie folgt:

Schreiben Sie eine Funktion

```
Node* sortList( Node* listHead )
```

welche die übergebene Liste aufsteigend sortiert und einen Zeiger auf den Kopf der sortierten Liste (welcher den kleinsten Datenwert der Liste enthält) zurückgibt.

Verwenden Sie dazu den folgenden Algorithmus (Bubble Sort):

```
sorted ← false
while not sorted do
    sorted ← true
    for all E ∈ L do
        if (E is not last node in L) ∧ (E.x > E_next.x) then
            swap(E,E_next)
            sorted ← false
        end if
    end for
end while
```

Dabei bezeichnet E jeweils einen Knoten der Liste $\mathcal{L}$ und E_{next} den jeweiligen nachfolgenden Knoten (welcher über den right-Zeiger erreicht wird). swap(E,E_{next}) besagt demnach, dass die Reihenfolge zweier benachbarter Knoten der Liste vertauscht werden sollen. Hinweis: Im swap()-Schritt können Sie einfach die Datenwerte der beiden Knoten vertauschen. (Alternativ können Sie aber auch die Zeiger umhängen.) Beachten Sie, dass der letzte Knoten in der Liste derjenige ist, dessen right-Zeiger auf den Listenkopf zeigt.

Ergänzen Sie das Hauptprogramm aus Aufgabenteil a) um den Aufruf von sortList (), so dass die eingelesene Liste in sortierter Reihenfolge wieder ausgegeben wird.