Übung 10

Aufgabe 1: Sortierverfahren

Die Idee: Gegeben sind Zeichenketten (z.B. Namen) in einer beliebigen Reihenfolge. Diese Zeichenketten fügen wir in einen Suchbaum ein. Wenn wir die Knoten des Suchbaumes in der richtigen Reihenfolge ausgeben, erhalten wir die Namen in sortierter Reihenfolge.

Die Datenstruktur: Der Suchbaum hat folgende Datenstruktur:

```
1 const int N = 30;
2
3 struct SuchKnoten {
4   char schluessel[N];
5   SuchKnoten *links, *rechts;
6 };
```

Die Suchbaumeigenschaft ist:

- Der linke Teilbaum eines Knotens enthält nur Elemente, die kleiner sind als das Schlüsselelement des Knotens oder ihm gleich.
- Der rechte Teilbaum eines Knotens enthält nur Elemente, die grösser sind als das Schlüsselelement des Knotens.

Die Einfügeprozedur: Die Einfügeprozedur ist rekursiv und implementiert die Suchbaumeigenschaft. Ein Beispiel für eine Einfügeprozedur finden Sie unten. Diese Prozedur muss auf die gegebene Datenstruktur angepasst werden.

```
 1 void Insert( int z, TreeNode* &r )
 2 {
 3   if ( r == 0 )
 4   {
 5     r = new TreeNode;
 6     r->left  = 0;
 7     r->right = 0;
 8     r->key   = z;
 9     return;
10   }
11
12   if ( z < r->key )
13     Insert( z, r->left );
14   else
15     Insert( z, r->right );
16 }
```

Die Ausgabeprozedur: Die Ausgabeprozedur ist auch rekursiv. Sie gibt den Baum mittels Inorder-Traversierung aus. Ein Beispiel für eine solche Prozedur ist unten angefügt. Auch diese Prozedur muss auf die gegebene Datenstruktur angepasst werden.

```
1 void Inorder( TreeNode* r )
2 {
3   if ( r != 0 )
4   {
5     Inorder( r->left );
6     cout << r->key << endl;
7     Inorder( r->right );
8   }
9 }
```

Aufgaben:

a) Implementieren Sie für die gegebene Datenstruktur die Einfügeprozedur
   `void einfuegen(SuchKnoten *&baum, const char name[])`
   die den Namen `name` in den Suchbaum `baum` einfügt.
   Hinweis: In der Header-Datei `string.h` stehen Ihnen sehr viele Operationen zur Zeichenkettenbehandlung zur Verfügung. Sie werden folgende zwei Operationen benötigen:
   `void strcpy(char *dst, const char *src)`
   Diese Prozedur kopiert die Zeichenkette `src` in die Zeichenkette `dst`.
   `int strcmp(const char *s1, const char *s2)`
   Diese Funktion vergleicht die Zeichenkette `s1` mit der Zeichenkette `s2` und gibt folgendes Resultat zurück:

$= 0$	falls beide Zeichenketten gleich sind,
> 0	falls s1 grösser als `s2` ist,
< 0	falls s1 kleiner als `s2` ist.

b) Implementieren Sie für die gegebene Datenstruktur eine Inorder-Traversierung
   `void ausgabe(SuchKnoten *baum)`
   die die Schlüssel des Suchbaumes `baum` ausgibt.

c) Fügen Sie die beiden Prozeduren aus den Teilaufgaben a) und b) zur einer Sortierprozedur
   `void sortieren(const char Namen[][N], int n)`
   zusammen.

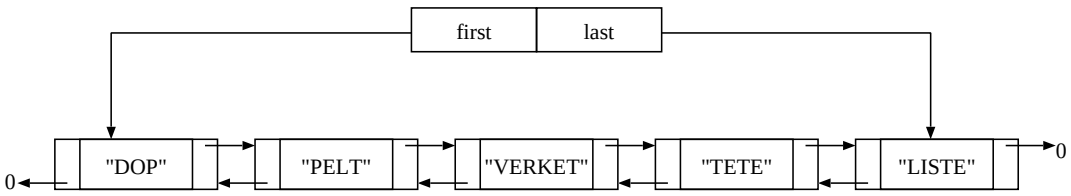
d) Testen Sie Ihre Prozedur Sortieren mit folgendem Hauptprogramm:

```
 1 const int M = 8;
 2
 3 char namen[M][N] = {
 4   "Laura Baudis",
 5   "Roland Engfer",
 6   "Hans-Werner Fink",
 7   "Christof Aegerter",
 8   "Daniel Wyler",
 9   "Juerg Osterwalder",
10   "Nicola Chiapolini",
11   "Ulrich Straumann"
12 };
13
14 int main()
15 {
16   sortieren(namen,M);
17 }
```

Aufgabe 2: Doppeltverkettete Listen

In dieser Aufgabe konfrontieren wir Sie mit einer weiteren Variante von Listen: Doppelt verkettete Listen. Sie müssen in dieser Aufgabe nur den Datentyp mit seinen Operationen implementieren.

Definition: In einer doppelt verketteten Liste haben die Elemente Zeiger in beide Richtungen: Zum nächsten Element und zum vorhergehenden Element:



Der Datentyp einer solchen Liste sieht z.B. wie folgt aus:

```
 1 const N = 30;
 2
 3 struct List_item {      // Listenelement
 4   char inhalt[N+1];     // Inhalt
 5   List_item *pred;      // Vorgaenger
 6   List_item *succ;      // Nachfolger
 7 };
 8
 9 struct List {           // Die Liste
10   List_item *first;     // erstes Listenelement
11   List_item *last;      // letztes Listenelement
12 };
```

Eigenschaften

Einfaches Löschen: Der Hauptvorteil doppelt verketteter Listen besteht darin, dass ein Listenelement, das durch einen Pointer zugänglich ist, ohne nochmaliges Durchlaufen der Liste gelöscht werden kann.

Einfaches Aneinanderhängen: Das Zusammenhängen zweier Listen kann ebenfalls ohne weiteres Durchlaufen der beteiligten Listen erreicht werden.

Aufgaben: Implementieren Sie folgende Operationen für doppelt verkettete Listen:

a) `List init()`. Diese Funktion initialisiert eine neue (leere) doppelt verkettete Liste.

b) `bool empty(List list)`. Diese Funktion testet, ob die gegebene Liste `list` leer ist.

c) `void push(List_item* list_item, List &list)`. Diese Prozedur fügt das gegebene Element `list_item` am Anfang der Liste `&list` ein.

d) `void append(List_item* list_item, List &list)`. Diese Prozedur fügt das Element `list_item` am Ende der Liste `&list` ein.

e) `void print(List list)`. Diese Prozedur gibt den Inhalt der Liste vom ersten zum letzten Element am Bildschirm aus.

f) `void del(List_item* list_item, List &list)`. Diese Prozedur entfernt das gegebene Listenelement `list_item` aus der gegebenen Liste `&list`.

g) `List conc(List list1, List list2)`. Diese Funktion hängt die Liste `list2` an das Ende der Liste `list1` und gibt das Resultat zurück.

Schreiben Sie ein Hauptprogramm so, dass Sie die Operationen testen können und von der Funktionstüchtigkeit Ihrer Implementation überzeugt sind.