Exercise 13 – Linked Lists

Objective

The objective of this session is to use a linked list to re-implement the word sorting code completed in the Searching and Sorting practical. This code was slow because the word array needed to be resorted after each insertion. This is not necessary with a linked list since the insertion point maintains the ordering of the list.

Reference Material

The material is based entirely on the Linked List chapter. This practical session is located in the following directory:

	Microsoft Windows	*Linux*
Directory:	**c:\qacadv\linkedlist**	**~/qacadv/linkedlist**
Solution directory:	**c:\qacadv\linkedlist\Solution**	**~/qacadv/linked/Solution**

Overview

There are two questions in this session. In the first question, we present a file called **listcode.c**, which contains all the code to manage a list of `int`s. We also provide a file called **readkbd.c**, which exercises this list code. The program reads `int`s from the keyboard and stores them in a list. Your task is to convert the program to deal with strings rather than `int`s.

In the second question, you will use the string version of the linked list to store words read from a text file. We provide the necessary file handling code; your mission is to extend the program to insert the words into a list.

There is a final optional question that implements a self-adjusting linked list. This is where the nodes are held in order of popularity, rather than being sorted by a data key.

Practical Outline

On Microsoft Windows open **readkbd.sln, on Linux** change your current working directory

This program contains a header file and two source files. Take a look at these files:

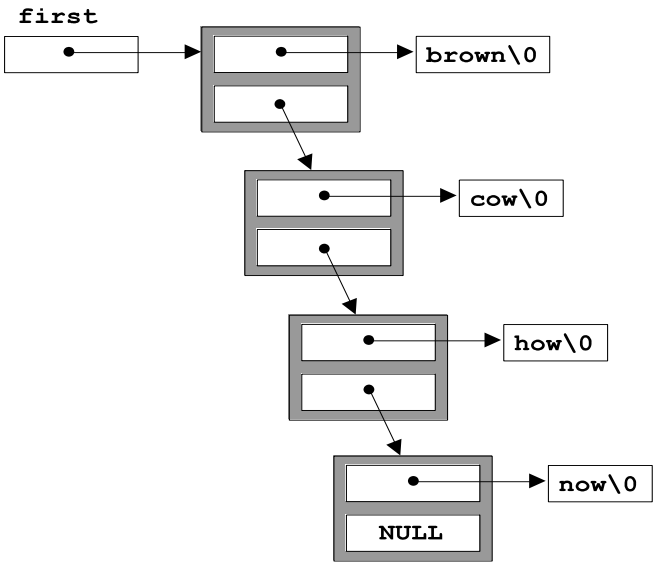
listcode.h Declares a struct template to represent a node in a linked list (note that it's a list of `int`s). The file also declares a variety of list-related functions.

listcode.c Defines the list-related functions. Take a look at these functions, to convince yourself that they are similar to the functions discussed during the lecture!

readkbd.c Contains the `main()` function for the program. `main()` reads `int`s from the keyboard, and inserts each one into the list (using the `insertList` function).

Modify the linked-list code in **listcode.h** and **listcode.c**, so that it stores words rather than `int`s. Don't put a fixed size character array into the nodes, instead use a `char*` and dynamically allocate the storage for each new word.

Thus, the four words "how now brown cow" would build a list as follows:



You will also need to change the test harness in **readkbd.c**, so that it reads strings rather than `int`s from the keyboard. Build and run the program; enter plenty of words, to make sure each word is stored at the correct position in the list.

On Microsoft Windows open **readfile.sln, on Linux** stay in the same working directory.

This program contains the following files:

listcode.h	Your modified file from Question 1 above.
listcode.c	Your modified file from Question 1 above.
readfile.c	Contains the `main()` function for the program. This program reads words from a text file, rather than from the keyboard.

Before making any changes to the code, build and run the program as it stands. The program asks you for a filename – you can use the text file **long.txt** as trial input. The program reads all the words from the file, and just displays them in the order they appeared in the file.

Now take a look at the code in **readfile.c**, to see what it's doing. Examine the `process()` function, which breaks the input file into separate words. Each word from the file is read via the `getNextWord()` function into a character array. Modify the program to build up a linked list of words, using your code in **listcode.c**. Print this list on the screen once everything has been read from the file.

Build and run the program again, and use **long.txt** as trial input as before. This time, the words should be displayed in sorted order (the list ensures this).

When you have the code working you might like to incorporate the timing routines from the arrays practical. You should find the program is noticeably quicker than the equivalent from the Sorting and Searching practical, in which the array was resorted using `qsort` after each insertion.

Optional Question

Linux only

In this exercise we will initially create a simple stack-like singly linked list of names and telephone numbers. We wish to concentrate on linked list handling, so our application will not have many features. Next we will refine the list so that the most popular items will have the shortest search time.

For simplicity we will read the names and numbers from a file, with the default file name **phone.txt**. The linked list will be populated from that file by merely adding each item onto the list as it is read. This will result in the records being stored on the list in reverse order, i.e. the record at the head of the list will be the last one read.

Most of the code has already been written for you in **Link2.c** – it is assumed that you are familiar with the basic principles. It is worth scanning the code to get an idea how the application fits together.

The records on the list are stored in a typedef'ed struct called **Node**, which is defined in **Link2.h**. The records are read from the file by function **iPopulate**, and added to the list in **AddItem**. The head of the list is indicated by a global called **g_pAnchor**.

The main program asks for a name to retrieve, finds and displays it, then prints the entire list as it is stored.

Part 1

Initially, write the code to find an item in the list by name (function **GetNode**). Don't be too ambitious, use **strcmp** for the comparison.

Part 2

Now change **GetNode** to implement a self-adjusting linked list. This is a type of linked list that stores the most recently requested items near its head. Whenever an item is retrieved (**GetNode**), move it to the head of the list. Over time the most popular items will migrate near the head of the list, giving faster retrieval times.

Test with the supplied phone.txt file, and add further data if you wish (the names and numbers are comma separated).