Exercise 14 – Binary Trees

Objective

The objectives of this session are to consolidate your understanding of binary tree manipulation.

Reference Material

This practical session is based on the Binary Trees chapter, concentrating in particular on how to generate a binary tree. This practical session is located in the following directory:

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

Overview

There are two questions in this session, following a similar pattern to the Linked List practical session. In the first question, we present a file called **treecode.c**, which contains all the code to manage a binary tree of `int`s. We also provide a file called **readkbd.c**, which exercises this tree code. The program reads `int`s from the keyboard and stores them in a tree. 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 binary tree 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 binary tree.

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 for handling binary trees (these files are the binary-tree equivalents of the files used in the Linked List practical exercise):

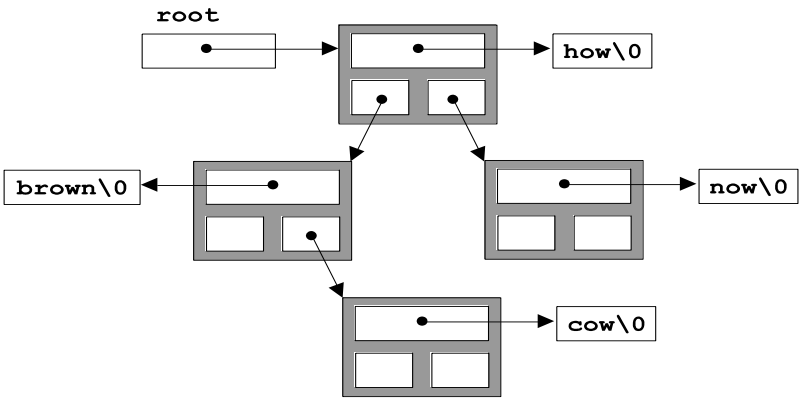
treecode.h Declares a struct template to represent a node in a binary tree (note that it's a tree of `int`s). The file also declares a variety of tree-related functions.

treecode.c Defines the tree-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 tree (using the `insertTree` function).

Modify the binary tree code in **treecode.h** and **treecode.c**, so that it stores words rather than `int`s. Just as in the Linked List exercise, 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 tree 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 tree.

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

This program contains the following files:

treecode.h Your modified file from Question 1 above.

treecode.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. As before, each word from the file is read via the `getNextWord()` function into a character array. Modify the program to build up a binary tree of words, using your code in **treecode.c**. Print this tree 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 tree 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 quicker than the equivalent from the Linked List practical; you will also remember that this program was quicker still than the equivalent from the Sorting and Searching practical!