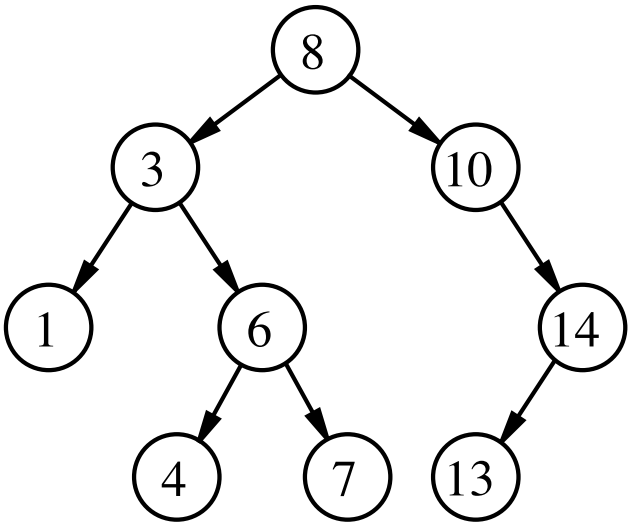
# Binary Trees
## CS2002 Practical C2

**Due 9pm Friday 6th March**

This practical carries the same share of the coursework credit for the module as the others. You should submit your work on MMS by the deadline.

## Introduction

First we shall build a binary tree, and functions for inserting nodes into it and cleaning up the tree at the end. A binary tree is a data structure that stores sorted data. Given the value at a node, all child nodes to the left will be smaller, and all child nodes to the right will be larger. It is therefore possible to find if a particular node is in the tree by starting at the root node and at each node looking if the value stored at this node is equal, smaller or larger than the required value. An example of a binary tree is given in the diagram below (diagram by Derrick Coetzee).




Our binary trees will store a void* at each node, so we can use them to store any type. You do not have to worry about making your trees **balanced**.

## 1 Part A - Basics of Binary Trees

We shall begin by designing a basic binary tree, and functions for inserting a new node. You will find a header, with prototypes for all these structs and functions, in `treenode.h`.

The struct `TreeNode` represents a node of your binary tree. It contains a `void*` member for storing the data at each node, and a pointer for the left and

right children of the node.

`struct TreeBase` stores the base of a tree, and some function pointers. It contains:

1. A `TreeNode*` representing the base of the tree.
2. A function `int(*comp)(void*, void*)` which is used to order nodes of your tree. This comparison function should return -1,0 or 1 (like strcmp).
3. A function `void(*clean)(void*)` which is used to free the memory of the `void*` member of a TreeNode. We accept this as a function rather than just calling `free` to allow users to manage their memory however they like.
4. A function `void(*print)(void*)` which prints out the value of a node. For example for an `int` node, a valid print function could be:
   `void print(void *v) { int* i = v; printf("%i", *i);}`

Implement the methods whose prototypes are in `treenode.h`, described below:

1. Construct a new tree (which takes a comparison, print and clean method, as described above)
2. Look a value up in the tree.
3. Add a value to the tree (this can assume the value is not already present).
4. Print a tree (from smallest to largest member, using the `print` function from (d) above).
5. Free all memory from a tree, including calling the `clean` method on the data stored at each node of the tree.

Write a program, called `test` in a file `test.c` which tests your methods, by building some trees, checking if values are present, and printing out the trees. This program should use `assert` where possible to check methods return correct values. Try to test your tree works correctly before progressing, and discuss the tests you ran in your submission.

Place `test` in a correct `Makefile`. Make sure this program is in your final submission (do not edit/break it while doing Part B or extensions).

# 2 Part B – Implementing word counting

We shall now use our code from part A to count the number of different words in a file. We shall do this by building a tree which, at each node contains:

1. A word (represented by a char*) and;

2. the number of occurrences of that word.

The first time a word is encountered is it added to the tree, after that we just increment its count. The `comp` method should just look at the word, not the count (so a word is only added to the tree once).

Method:

1. Write a function which reads from a file and returns a `TreeBase` containing all the words and their counts.
2. The code should read user input a word at a time. For the purposes of this project, treat a word as any continuous string of lower case characters, upper case characters and apostrophes, turned into lower case (so `cat, Cat` and `CAT` are all the same word. `cat's` is a single, different word)
3. Print out how many times every word occurs, in alphabetical order.

Test your code first on "tiny.txt" (the output your program should produce is in tiny-out.txt), and then on the file "pride.txt".

# 3 Extensions

For extra credit, ensure that your code accepts any length of word, rather than using a hard-wired buffer size. Also, include a method of calling a function on every node of the tree, and use this method to find the most used word in `pride.txt`, and also to give some other interesting statistics (for example, average word length or total number of words).

# 4 Submission

Submit your program as a tar file containing your source. Submit a short report covering your design, implementation and testing, and any problems encountered and lessons learned as PDF.

## Marking

Your submission will be marked according to the standard mark descriptors published in the Student handbook at

`https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Mark_Descriptors`

## Lateness Penalties

You should submit your work on MMS by the deadline. The standard lateness penalties apply to coursework submitted late. As indicated at:

`https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#lateness-penalties`

## Good Academic Practice

I would remind you to ensure you are following the relevant guidelines on good academic practice as outlined at:

`https://studres.cs.st-andrews.ac.uk/Library/Handbook/academic.html#/Good_Academic_Practice`