CSCE 221 Assignment 1

Casey Brooks

Due 9-9-2013

Program Description.

This program is a simple program that creates a Binary Tree and performs operations on this data, such as the size of the tree, the height, and the searh cost of individual elements and of the whole tree. It is also implemented as a binary search tree, so that data will be organized numerically as it is inserted for rapid searching.

Purpose.

The purpose of this assignment was to get experience in implementing and using a binary serach tree to store integer data. It gives us more practice with pointers, as well as experience in writing recursive algorithms to more efficiently solve problems.

Data Structures.

The data structure used is a binary search tree, which is a heirarchal way to organize data. The tree contains a root, implemented as a node which contains data and pointers to other nodes. The node points to one on the left, indicating that data is less than the root, and a node on the right, which is greater. These nodes then do the same thing, pointing to nodes of data greater and less than itself. In structuring the tree in this way, any element can be found quite quickly, $O(log_2 n)$ if the tree is nicely balanced, and it is easy to access data within it, simply by calling the recursive function that returns the appropriate node.

Algorithm.

Most of the algorithms used in this assignment were recursively defined. Starting at the root node of the tree, we call a function that iterates through the lower nodes in the same way until the desired operation is completed, usually terminating when a specific node is found or when the next call would be on a node that is null. Some operations are ones that find the height of a tree, by determining the max of the height of a specific node's children. An element can be found quickly by searching for the key by following the tree's path of elements that are greater or less than the desired element. Since each node contains the value for how many comparisons it takes to search for it, by iterating through the entire tree, in this case with an inorder traversal, we get the total number of comparisons for the whole tree. LIkewise, if we just incrememnt a counter for

each node in a traversal, we get the size of a tree, and if we divide the number of comparisons by the size, we get the average search cost for any given node in the tree.

Classes and Organization.

The binary search tree was implemented using binary nodes and pointers as opposed to an array. Each node contains the value of the element to be stored as well as the search cost the that particular element, and pointers to nodes containing the left and right subtrees. These nodes are enclosed in the BinarySearchTree class, which contains functions that augment those in the BinaryNode class, as well as some that simply mask a call to the same function in the Node.

Compiling Instructions.

There is not special commands for compiling, just the normal "g++-4.7 -std=c++11 *.cpp" and then run it with "./a.out".

Input and Output.

There is no input necessary for this program, as all the test files are read and executed programatically with a single run. The data for these files is put into trees, and data about these trees are inserted into a .csv file which can be opened as a spreadsheet in Excel. The final structure of the tree is then printed into files matching the name of the input with "_tree" appended to the title.

Exceptions.

I did not implement any exceptions, but in the case where an exception could be thrown I added a cout message or just left it blank.

C++ Object-Oriented Features.

Apart from reading and storing the data from all files correctly, I implemented a feature to write the data to be plotted, the number f nodes, the number of comparisons for each node, etc., to a .csv file. This makes it very easy to make a table out of the data, which can then be used to make a graph of this data.

Tests.

The results of the test were as expected. The perfectly balanced trees performed the best, with the average search cost being less than the heigh of the tree, which in itself was a perfect $log_2 n$ where n is the number of nodes in the tree. The randomly inserted nodes did well too, though not as well as the perfect trees: the heights being an average of twice that of the perfect tree. But considering the cost of searching for an element in a random tree is still $O(log_2 n)$, it is still very efficient. And when the data is input already sorted, with elements increasing linearly, the operations took $O(n)$ time to finish. While for a tree this is really slow, it basically has become a linked list, an as such is still an optimal data structure, even moreso considering that the data is sorted.

Resources.

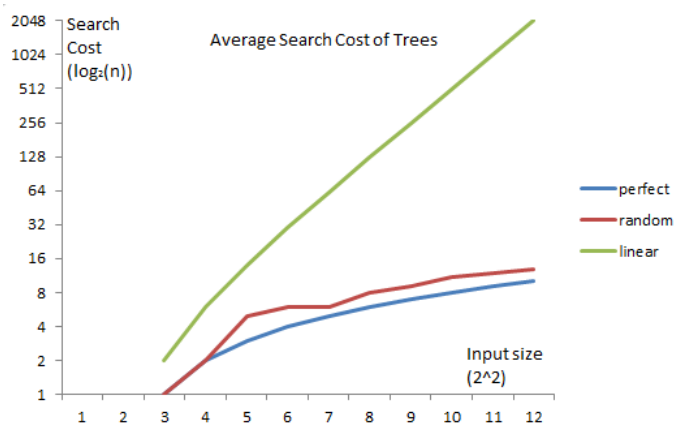

Figure 1: Graph of input size versus average search cost

Class lectures and slides for theory of the binary tree and code snippets for some of the features.