Programming Assignment 4 Report CSCE 221

Oneal Abdulrahim

1. In this assignment, we were tasked with implementing a Binary Search Tree with proper methods. Then we loaded various trees with data from differently organized files of integers. The integer order differed, making the trees perfectly balanced, randomly balanced, and very unbalanced. In my implementation, I used many of the methods on the slides to create a node class, which held pointers to the left and right children and integer data, as well as the search cost (which is the number of comparisons it would take to reach this node, also the "level" of the node). Then, I combined these nodes in a BinaryTree class, which held a root node that the tree would grow from. The insert function in BinaryTree would do all the work, and insert nodes in the correct order and make comparisons. When printing the tree, the three traversal methods (pre-order, in-order, and post-order) would simply print the elements and their search values. When printing the tree itself to a file, I used the mentioned breadth-first-search algorithm, using a queue to store data and print the tree level-by-level.

   The files included have been precompiled. Simply use the command `./BinaryTree` to run the program. For different files, the program will need to be rerun. Many result files have already been included.

2. A Binary Search Tree is a data structure that holds nodes that have left and right children, and grows from a root node. Each node holds some data, and in our case, we used integers. To implement this in C++, we must use an individual node structure. Each individual node contains pointers to its left and right child. Other implementations allow the node to have a pointer to its parent as well. Then, these nodes are combined and the root node (the node with no parent) is a property of a BinaryTree structure.

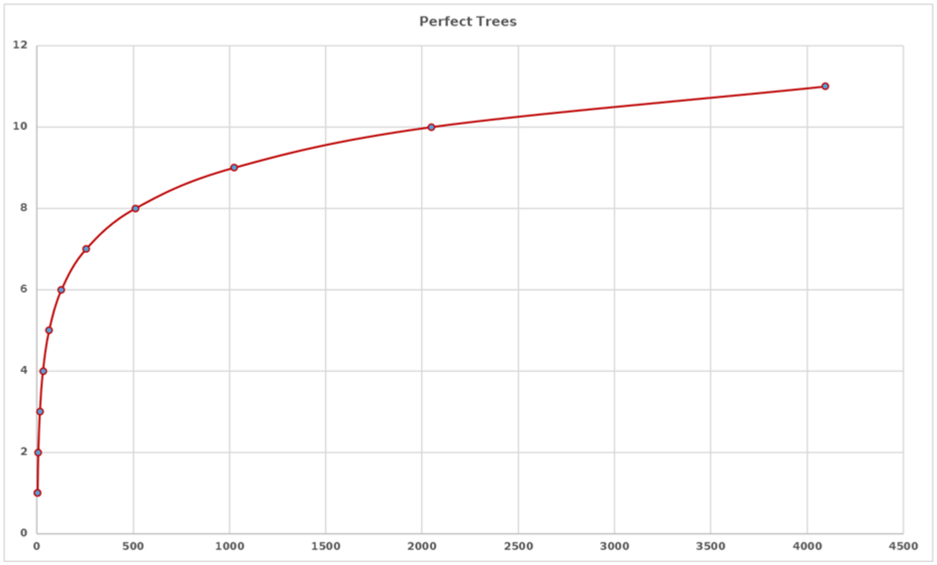
   To traverse, print, or insert elements for BinaryTrees, the concept of recursion is needed. Most methods to perform these operations are implemented easiest using recursive calls. It is possible to perform these functions iteratively as well (likely through the usage of a stack or queue). In the recursive case, we must choose the order of traversal, then make decisions revolving around starting at the root node, then checking left and right children, and so on.

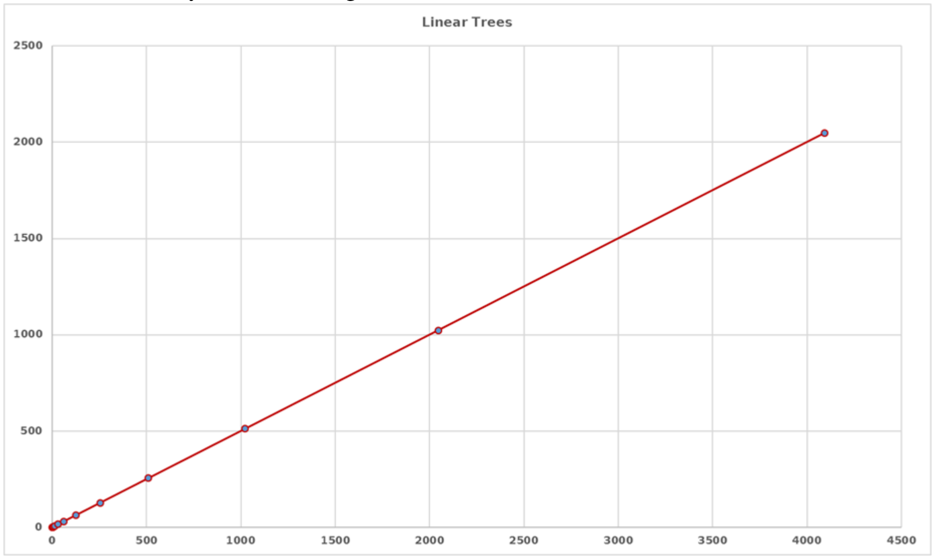
3. Some descriptions of how I implemented:
   (a) individual search cost
   After nodes have been inserted, a call to the `set_costs` method is made. This performs in-order traversal recursively, and when the root is visited, calls a method called `depth` which calculates the height of the current node, then this is subtracted from the depth of the root. The depth function is $O(log_2 n)$. Since both of these methods visit every node for each call, and one calls another within itself, we can conclude that this method is $O(n^2)$

   (b) average search cost
   The tree's average search cost is calculated within `set_costs`, which simply adds the current node's cost to the tree's running sum property. Just this aspect (the summing of all the search costs) of the method is linear time complexity $O(n)$.

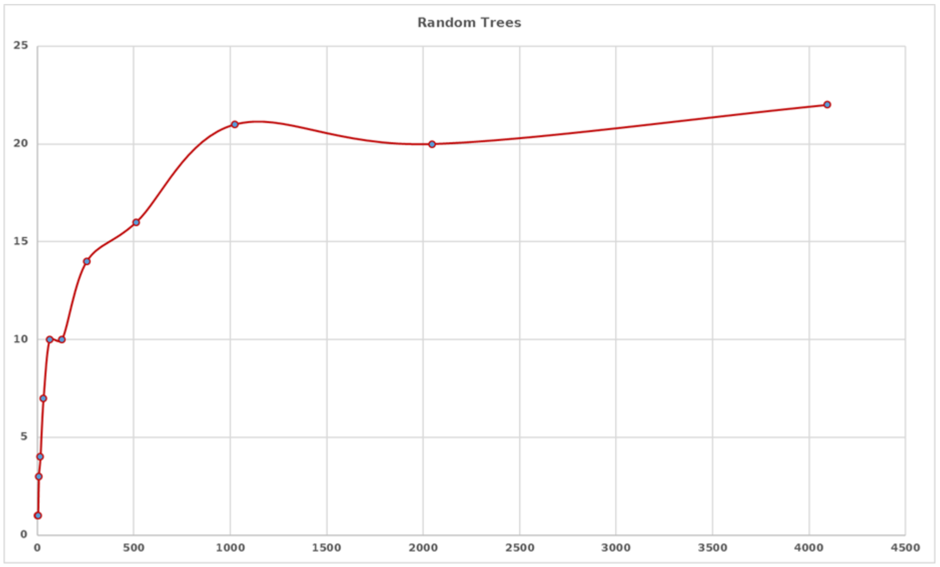
   (c) updated search cost
   The `set_costs` method is called after a node has been removed. As seen above, this method is $O(n^2)$

4. Give individual search cost in terms of n using big-O notation. Analyze and give the average search costs of a perfect binary tree and a linear binary tree using big-O notation, assuming that the following formulas are true (n denotes the total number of integers).
   $\sum_{d=0}^{log_2(n+1)-1} 2^d(d+1) \approx (n+1)\cdot log_2(n+1) - n$ and $\sum_{d=1}^{n} d \approx n(n+1)/2$
   The formula above represents the sum of all of the search costs in our Binary Tree. If we have n nodes, we arrive at $O(log_2 n)$ for perfect trees, $O(n)$ for linear and random trees.

5. 

   We can see clearly that this is logarithmic, similar to our theoretical results.
   

   This is linear, exactly our previous result



This is not linear and is shaped more logarithmic. Since big-O is an upper bound, we can say this is linear time complexity, which is consistent with our previous result.