Ryan Russell

Dr. Leyk

CSCE.221.511

1 April 2020

Assignment Three Report

1. In this assignment, the objective is to create a C++ class that represents the binary search tree data structure. In addition to all the basic functions such as insert, search, delete, and others, the program is supposed to be able to give each node a integer value titled search_cost (which is defined to be the number of comparisons it takes to find the node in the tree), print a node from a tree along with that node's search cost, updating all the search costs for each of the individual nodes in the tree, and output the average search cost of all the nodes in the tree. The inorder traversal and the level-by-level representations of the tree should also be printed onto the screen or be put into a file. These programs are compiled by using a makefile which is able to link multiple files together and are able to run by running the one file created from the makefile. The files that are used include BSTree.h, BSTree.cpp, and BSTree_main.cpp. The file BSTree.h serves as the header file for the assignment which declares the class and the methods involved with it but does not have any implementation. The file BSTree.cpp is the C++ source file which provides the implementation of the methods defined in the header file, BSTree.h. Finally, BSTree_main.cpp serves as the C++ source file where the user is able to create trees and use the methods defined by the previously two mentioned files.

2. The data structure that is created in these files is a binary search tree. A binary search tree is made up of nodes like data structures such as the linked list. These nodes can either have a comparable value or be null and are pointed to each other which in this case it is where a node points to two “children " nodes to its left and right. This creates a tree-like structure if you draw the tree out physically. What makes this data structure different from other binary trees is that if any node has children nodes to its left and/or right that are not null, the left child has a smaller key and the right child has a bigger key than the parent node. These trees can have any comparable objects as nodes such as doubles, floats, strings, etc. In this case of this assignment, the key of these nodes are integers. Therefore, if a node has children nodes to its left and/or right, the integer key value of the left child is smaller and the right child has a bigger integer key value than the parent node’s integer key value.

3. The way that I implemented the calculation of individual search cost of a node is by recursively going down the tree. I started by testing if the node passed through the function is not null. If it was not null, I assigned the node passed through initially its search cost of one by having an integer parameter called num and then if the node had a child to its left or right, I would call the function to assign that node’s search cost num + 1. This approach gives the root a search cost of one then recursively gives its children a search cost of two and then recursively gives their children a search cost of three which continues until all nodes are visited. This assigns the 1 + depth level of each node in the tree and as depth level increases, so does the search cost for that node. The way that I implemented the calculation of average search cost is by finding and getting the sum of all the search costs of each individual node in the tree then dividing it by

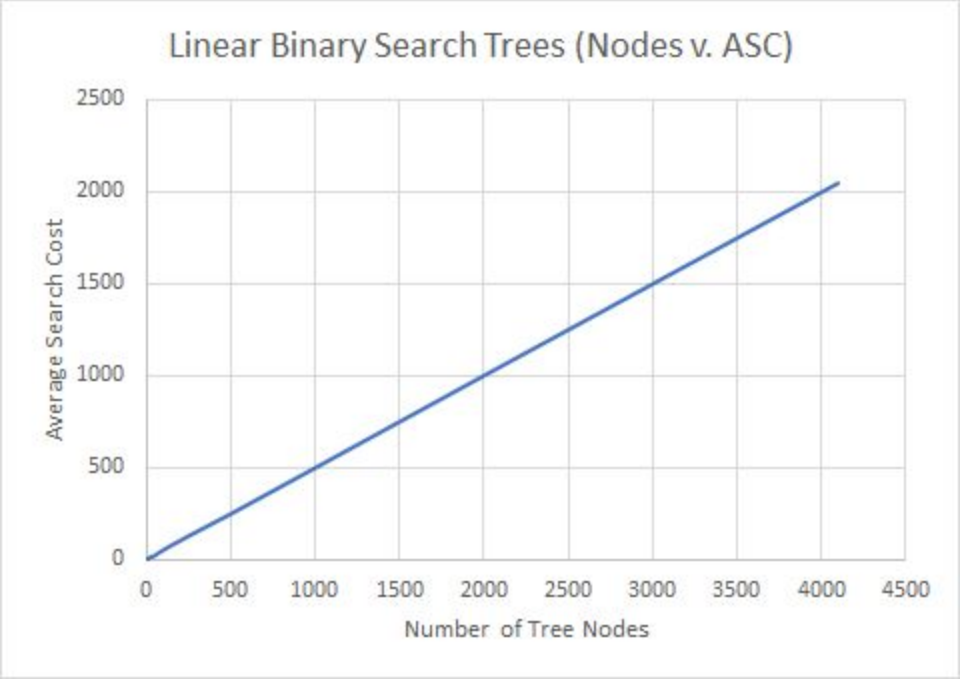
the total number of nodes in the tree. The tree operations insert and update_search_times were helpful in doing this. Insert allowed for the nodes to be added to the tree which increased depth and search time. Update_search_times allowed for after a node's insertion for a traversal of the tree and making sure that each node's search cost/depth was correct which made it possible to calculate the total search time of the tree. The time complexity of calculating individual search cost is O(log n). The reason why it is O(log n) is because the insert function gives the binary search tree the ability to go either left or right (depending on the key of the node) which cuts the search time down by half which can be represented as f(n/2), f(n/4),...f(n/(2^k)) where n represents the number of nodes and k represents the number of times the function is called. The time complexity of calculating the total search cost of all the nodes in a binary search tree is O(n) because n represents the number of nodes and each of the nodes in the tree is visited once to get its search cost and add it to a sum.

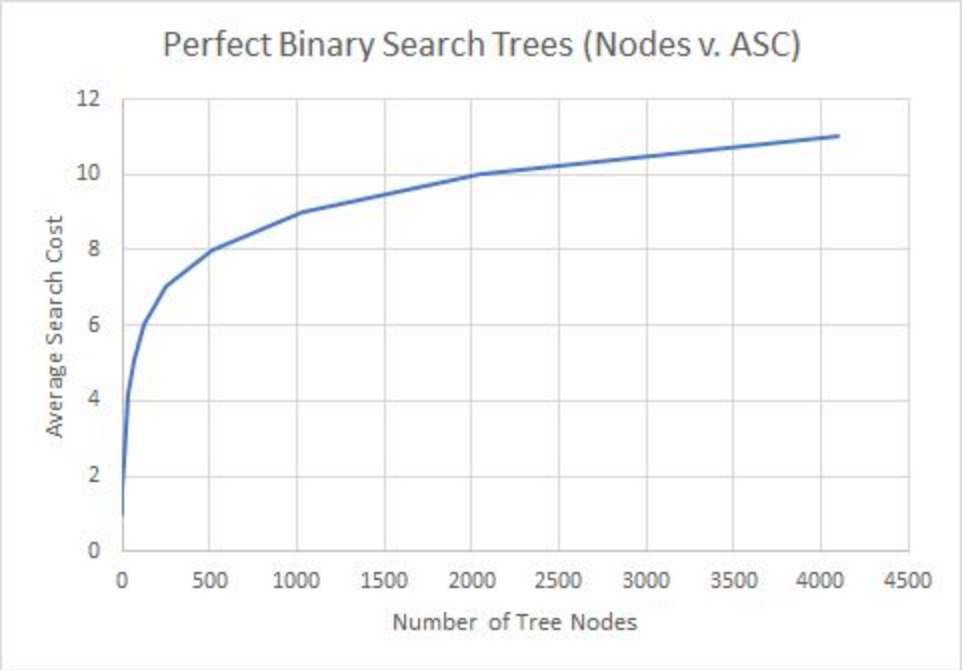
4. The individual search cost of a perfect binary search tree is O(n*log n) using the formula $\sum_{d=0}^{log_2(n+1)-1} 2^d(d+1) \simeq (n+1) * log_2(n+1) - n$. The individual search cost of a linear binary search tree is O(n^2) using the formula $\sum_{d=1}^{n} d \simeq n * (n+1)/2$. If we take these time complexity and divide them by the size of the tree or the total number of nodes, n, then we get the average search cost for a perfect binary search tree is O(log n) and O(n) for a linear binary search tree.

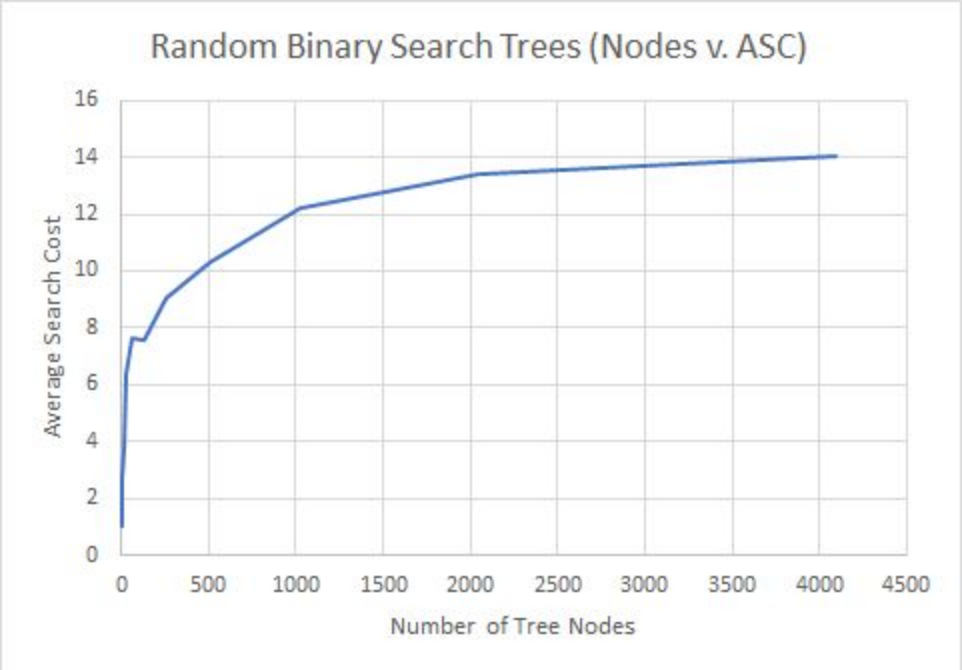
5.

Number of Nodes	Linear Average Search Cost	Perfect Average Search Cost	Random Average Search Cost

1	1	1.000	1.000
3	2	1.667	1.667
7	4	2.429	2.714
15	8	3.267	3.733
31	16	4.161	6.387
63	32	5.095	7.667
127	64	6.055	7.591
255	128	7.031	9.067
511	256	8.018	10.303
1023	512	9.010	12.246
2047	1024	10.005	13.397
4095	2048	11.003	14.024







As previously stated, the time complexity of summing up the search costs over all the nodes is O(n). As shown in the Linear Binary Search Trees (Nodes v. ASC) graph, the relationship between the number of tree nodes and average search cost in linear binary search

trees is linear. Since linear binary search trees are considered to be the worst case when it comes to having a binary search tree, this proves that the worst case time complexity of searching for a node in a binary search tree is O(n). Also previously stated, the time complexity of calculating individual search cost is O(log n). As shown in the Perfect Binary Search Trees (Nodes v. ASC) and Random Binary Search Trees (Nodes v. ASC) graphs, the relationship between the number of tree nodes and average search cost in perfect binary search trees is logarithmic. Since perfect binary search trees are considered to be the best case and random binary search trees are considered to be the average case when it comes to having a binary search tree, this proves that the best and average case complexity of searching for a node in a binary search tree is O(log n).