# CSCE 221 Assignment 4 Cover Page

Asa Hayes UIN: 525003952

User Name: AsaHayes E-mail address: asahayes@tamu.edu

Please list all sources in the table below including web pages which you used to solve or implement the current homework. If you fail to cite sources you can get a lower number of points or even zero, read more on Aggie Honor System Office website: `http://aggiehonor.tamu.edu/`

| Type of sources | |
|---|---|
| People | |
| Web pages (provide URL) | http://www.cplusplus.com/reference/<br>Reference on use of input and output operators |
| Printed material | Textbook for this class<br>Reference on binary tree structure |
| Other Sources | |

I certify that I have listed all the sources that I used to develop the solutions/codes to the submitted work. *On my honor as an Aggie, I have neither given nor received any unauthorized help on this academic work.*

Asa R Hayes Date 23 April 2018

# Programming Assignment 4: Report

**Program: Due April 2 at 11:59 pm**

# 1 Introduction

## 1.1 Assignment Objective

Write a C++ program to create binary search tree. Your program should empirically calculate the average search cost for each node in a tree and output a tree, level by level, in a text format.

## 1.2 Compile and Run Instructions

Code should be built and run with "make all && ./Main". When run, it will ask for user input for filename (do not include directory, just filename), will proceed on input of a valid name of any of the included data files.

## 1.3 Program structure

The two main classes used in this project are the node class, used as regular except with a left and right child node instead of next and/or previous node pointers, and a binary tree based on those nodes. The tree is a container for a root node and its child nodes, with functions to insert new nodes into the tree, multiple output functions, and other utilities.

# 2 Description of Data Structure

A binary tree is a collection of nodes containing a value and pointers to left and right child nodes, starting at the root node. To be a correct binary tree, the left child of any node must have a lesser value than its parent, and the right child must have a greater value than its parent. This arrangement is able to achieve search times much better than many of the other structures we have learned about in average cases, as it minimizes the amount of elements it has to search thru to reach a value. If deletion were a part of this assignment then there would be more complexity, as removing elements form the tree usually requires a restructuring of the tree to keep the qualities of the tree as valid to a binary tree.

# 3 Search Costs

## 3.1 Calculation of Individual and Average Search Cost

The search cost for each node was implemented to work during the insert function. Whenever a new node was inserted, the tree was traversed level by level to give each node its proper depth/search cost. This is most likely an inefficient way to do this, but the implementation was simple. To get the average search cost, the function get_total_search_time() did the same traversal as the setting of the search times for each nodes but adding each search cost to a total. After all search costs were added, that amount was simply divided by the size of the tree (a member variable of BTree) to get the average search time.

## 3.2 Time Complexity

Time Complexity for individual search cost is O(log n), the same as a search operation for a single node. As the sum search cost operation visits all nodes, it has a runtime of O(n).

# 4 Cost Analysis

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).

$\sum_{d=0}^{\log_2(n+1)-1} 2^d(d+1) \simeq (n+1)\cdot\log_2(n+1) - n$ and $\sum_{d=1}^{n} d \simeq n(n+1)/2$

Linear Tree: A linear binary tree has been unoptimized far enough to where it essentially functions as a linked lists, as all elements need to be traversed in order, giving it an average runtime of O(n).

Perfect Tree: The best case of a binary tree is when the tree is perfectly balanced, which gets us to the lower end of O(log n). Both the average and best cases of a binary tree of any kind are O(log n).

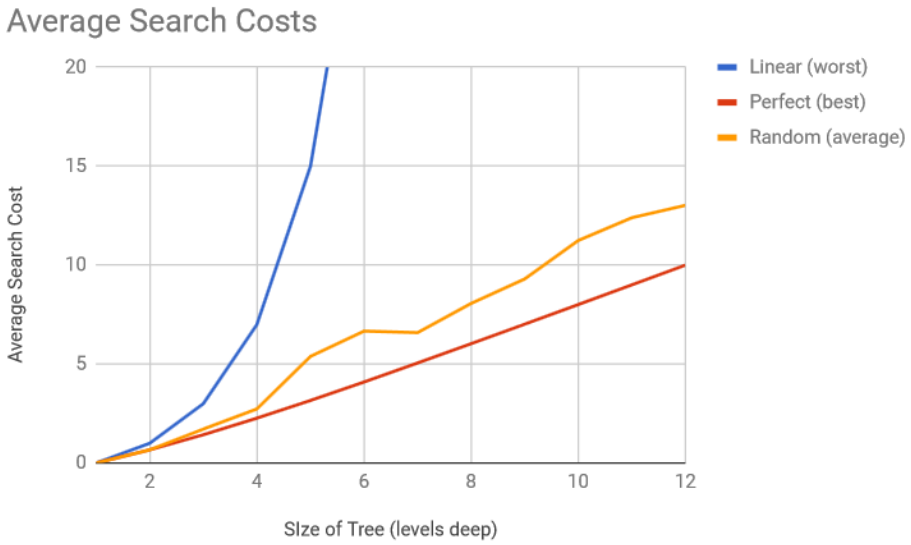
# 5 Average Search Cost Figures

## 5.1 Average Search Cost Table

| Size (levels deep) | Linear (worst) | Perfect (best) | Random (average) |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 1 | 0.66667 | 0.66667 |
| 3 | 3 | 1.42857 | 1.71429 |
| 4 | 7 | 2.26667 | 2.73333 |
| 5 | 15 | 3.16129 | 5.3871 |
| 6 | 31 | 4.09524 | 6.66667 |
| 7 | 63 | 5.05512 | 6.59055 |
| 8 | 127 | 6.03137 | 8.06667 |
| 9 | 255 | 7.01761 | 9.30333 |
| 10 | 511 | 8.00978 | 11.2463 |
| 11 | 1023 | 9.00537 | 12.3972 |
| 12 | 2047 | 10.0029 | 13.0237 |

## 5.2 Average Search Cost Plot

The vertical axis is cut off at 20 to keep the average/random and best/perfect lines readable. The linear/worst case line is seen to have exponential growth equal to $2^{n-1} - 1$




# 6 Discussion/Conclusion

The results of the running of the program against the theoretical time complexities was entirely in-line. I had several issues with the coding of this assignment, mostly to do with large issues with the in- and outstreams, as the actual implementation of the tree itself was fairly easy in hindsight.