# Programming Assignment 3 Report

Brandon DeAlmeida (bdealmeida@tamu.edu)

*UIN: 127008543*

*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. -Brandon DeAlmeida (3/29/2020)*

## Objective and Class Functions –

This programming assignment had me create the BSTree class, which contains a single BST binary tree. The code provided in this submission contains three C++ files: BSTree_main.cpp, BSTree.cpp, and BSTree.h. BSTree_main.cpp contains the main function for this program which tests my BSTree class (accessing the class by including BSTree.h), while BSTree.cpp and BSTree.h detail the class itself.

The BSTree class allows for the insertion of as well as searching for an item and includes a move assignment/constructor, copy assignment/constructor, and a deconstructor. There are other helper functions within the class, such as deallocateTree, that aid the functions listed above.

## Data Structure Description –

The BSTree class holds a single binary search tree. This tree is the most basic type of binary search tree, and doesn't include and balancing or node identification methods in order to cut down on insertion or search times. In addition to basic functionality, the BSTree class also keeps track of search times themselves by recording the number of comparisons in order to reach each node. These search times allows for average search times to be computed for a tree in order to adequately and numerically describe the class' efficiency with different input types.

## Implementation –

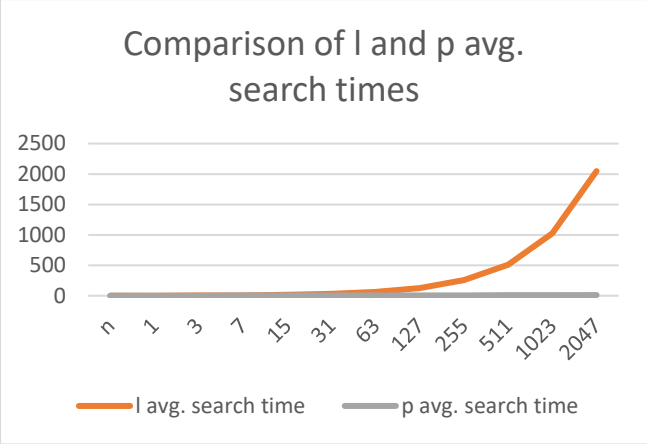
As each node's search time is related to its parent I found it easiest to traverse the tree, in an arbitrary order, assigning search times for each node's child(ren) where these search times are $T_{child} = T_{parent} + 1$ when the function update_search_times is run. Each node's search time is also assigned when it's first inserted into the BSTree using the same relation. After these values are assigned, the average search cost is simply $(\sum_{i=0}^{n} N_i)/n$ where $N_i$ is each node's search time and n is the size of the BSTree. Because when updating each node's search time and computing the average search time involves all nodes, both of these time complexities are in **O(n)**.

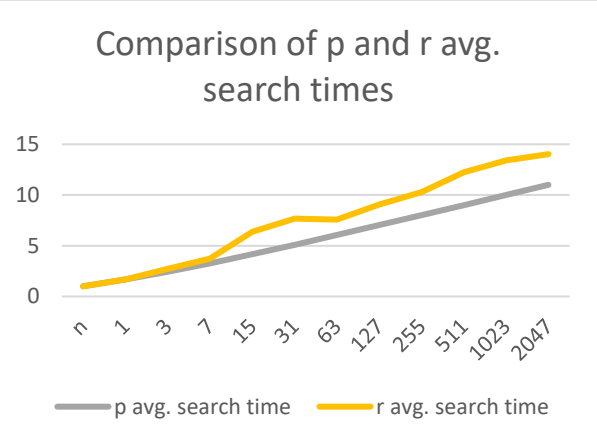
## Search Cost –

For a perfect binary tree in these tests the tree itself is balanced so that each height of the tree that is used is completely filled, this results in the best-case runtime for searching through the tree. For each comparison, the number of potential nodes that could be a match is halved. This results in a worst case runtime for a perfect binary tree of **O($\log_2$(n)**. A linear binary tree, however, is skewed so that its search times are the worst possible case for a binary tree. As every single node has to potentially be compared against this results in a runtime in **O(n)**.

## Data –

| N | L AVG. SEARCH TIME | P AVG. SEARCH TIME | R AVG. SEARCH TIME |
|---|---|---|---|
| **1** | 1 | 1 | 1 |
| **3** | 2 | 1.66667 | 1.66667 |
| **7** | 4 | 2.42857 | 2.71429 |
| **15** | 8 | 3.26667 | 3.73333 |
| **31** | 16 | 4.1619 | 6.3871 |
| **63** | 32 | 5.09524 | 7.66667 |
| **127** | 64 | 6.05512 | 7.59055 |
| **255** | 128 | 7.03137 | 9.06667 |
| **511** | 256 | 8.01761 | 10.3033 |
| **1023** | 512 | 9.00978 | 12.2463 |
| **2047** | 1024 | 10.0054 | 13.3972 |
| **4095** | 2048 | 11.0029 | 14.0237 |




From the data gathered from running my tests it's abundantly clear just how more efficient searching through both p and r trees are than l. As the difference between the p and r plots are negligible when compared to the l plot I've split the data into two graphs. From these the runtime trends of both the l and p cases are clear, showing $O(\log_2(n))$ and $O(n)$ respectively.