**A TIME COMPARISON BETWEEN AVL TREES AND RED BLACK TREES**

**Carson Davis, James Jackson, Johnnie Oldfield, Tamikal Johnson, and Matthew Hale**

***Abstract*** *- One of the most basic and common examples of a tree is a binary search tree. While efficient, there have been strides to make it even more optimized with the implementation of self-balancing and self-sorting. Two examples of this type of optimization are AVL trees and Red-Black trees. Comparatively, AVL trees are said to be faster than the Red-Black tree counterpart.*

*For this paper, we are going to discuss if this is, in fact, true. In order to test this, we are solving two separate problems using an AVL tree and a Red-Black tree. The main focus is to see how well the trees can keep up in each scenario and compare the time taken for both.*

**1 Introduction**

In its most basic form, a tree is a form of data structure that is used to store some kind of information. The simplest type of tree is a binary tree, which takes in information (with the first piece being the root) and compares it to the root, and places it to the left or right of the root accordingly. Although it will place it to the left and right, it will not sort the numbers properly. One step up from a binary tree is a binary search tree. It is very similar to a binary tree with the exception that it will sort the data while inserting it into the tree. The issue with this type of tree is that it will not auto balance the tree, and it could potentially leave the tree lopsided. Traversing a lopsided tree could provide linear search times, while balancing the tree can improve those times to O(log(n)). An AVL tree and a Red-Black tree is a modification of a binary search tree which will self-sort and self-balance. The projected time complexity for both of these actions in each type of tree are as follows: AVL: O(logn) and Red-Black tree: O(logn).

In order to test the time complexities of these trees, we are going to solve two separate problems. The first problem will deal with a phone directory in which the user can add, modify, find, or remove names and their associated numbers. This will test the trees’ traversal, addition, deletion, and modification times. The second problem is a movie script word counter. For this problem, we are going to feed a movie script to both trees. They will then process the script, then collect and sort each word. This is to test the trees’ sort, traversal, and addition times.

The purpose of this paper is to establish which tree is truly faster in each of the three scenarios mentioned before (adding, searching and modifying). We will also be discussing how these two types of trees are different from one another, and how they are more beneficial than a regular binary search tree. By the end of the trials, we should be able to definitely conclude which tree is faster and more useful and why this is.

**2 Background**

**2.1 Model and Notation**

AVL trees are named after two Soviet mathematicians, Georgy Adelson-Velsky and Evgenii Landis. AVL trees are self-balancing trees, which make them useful for sorting and display functions. This self-balancing means that, at most, the nodes will differ by at least one. If this is not the case, then the AVL program will take the initiative and balance itself.

Red and Black trees, so named for designating individual nodes as being red or black, was created by German computer scientist Rudolf Bayer as an offshoot of his earlier B-Tree. Red and Black trees often get compared to AVL trees because of their time complexity and self-balancing nature.

AVL trees and Red and Black trees are very similar. The main differences are that AVL has faster search speeds, store because it is more balanced where Red and Black has better insert and deletion speeds. The AVL tree stores heights for each node (O(n)), while Red and Black trees just store 1 bit for each node (O(1)).

**2.2 Related works**

The basics of the trees are constructed from sanfoundry.com[1]. While most of the algorithms came from geeksforgeeks.com[2]. These sites were able to help us construct our trees, and helped us to fix our red black trees when they broke. Being able to use other works as a foundation was very helpful to help us understand what we were working with and not having to struggle/brute force our work.

**2.3** **Motivation**

We decided to create a notional phone directory to help show the time complexity of search and insert/rotations of the red-black tree.

When working with a phone directory a tree seems like the proper data structure to use. The reason for this is if you use a structure like a list, stack, or a queue you will be getting slow times.

As far as lists are concerned, you are able to sort them based on names but you can’t jump around them unless you implement something like a binary search. The main reason a binary search would not work is that you would have to bounce around “chopping” the list in half each time to find the person. A stack wouldn’t work because you can only see what the last person put on a list would be. Finally, a queue wouldn’t work because you can only see the head and tail of the data which doesn’t help anything in a large amount of data.

So, in the end, an advanced tree is the only logical thing to work with since you are able to look down one side of a tree. A basic tree would not be sufficient because you would be lowering your odds of quick finding since it does not rotate. But trees such as AVL or Red-Black have an autorotation built into the insert methods.

For the second problem, we are making a script analyzer that takes in a large sum of words (which can repeat quite a bit) and seeing how many times they appear in said script. We chose to do this problem for two reasons: we thought it would be interesting to make/test and it is a great example of seeing how a Red and Black Tree can handle repeating data along with inserting a large amount of data.

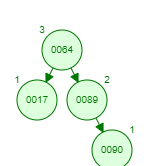
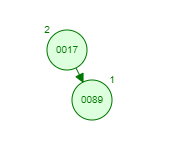
Although a map would be a better data type for this problem, a tree works just as fine. The reason a map would be more efficient is because elements in a map can natively be searched by their index, while elements stored in a tree cannot.

**2.4 Model**

A red-black tree takes in a node and attaches it as basic tree rules, which is where the red-black tree rotation comes into play.



For example, in an AVL tree if the numbers 17, 89, 64, and 90 were inputted into an AVL tree in this order the 17 would be introduced in the root at first. The 89 would be made into the right leaf because 89 is larger than 17. This is the part where it gets a little interesting. The 64 is inserted as a left leaf of 89 because it is less than 84 but 17 is also less than 64, so 64 is moved up to the root and 17 becomes the left leaf and 84 becomes the right leaf. The last number 90 is inserted as the right node to 84 and that will conclude a balanced AVL tree.



**3 Algorithm**

The first initial root is the easiest to understand because it takes the first input and puts it in the root space to be compared to anything that comes afterward. The program we have compares Strings in a way to make the input file alphabetical order. For example, if your first input was John Adam and your second was John Lewis. The John Lewis would be the leaf on the left side. The data is then stored in String nodes because through our trial error that is what was easy to work with because of the data that we are working with.

The program uses the String.compareTo() method which returns an integer. That integer is then figured out to be less than or greater than zero. If it is less than zero then it goes to the left node and if it is greater than zero then it goes to the right node.

Red and Black trees are very similar to AVL trees because they are both self-balancing trees and they both have the time complexity of O(logn) and in the worst case scenario O(n). That simply means these are among the fastest programs when used properly and even if not they are just as fast as normal or basic running programs.

The phone directory using a red-black tree takes a generic version of the tree so it could be sorted by any data type. The project consists of a multilevel red-black tree. The reason for this is to demonstrate how well the sorting works. Its time complexity would still be O(log n) for the fact that even if you are searching for a specific person you are only having to go through so many state nodes, then search another half of a tree just to find a person. This process still comes out faster than most algorithms.

**7 References**

[1]https://www.sanfoundry.com/java-program-implement-red-black-tree

[2]https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2