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 that utilize these types of optimizations are AVL trees and Red-Black trees. Comparatively, AVL trees are said to be faster than their Red-Black tree counterpart. In order to test this, we solved two separate problems using an AVL tree and a Red-Black tree. The primary goal of this research is to see which tree is faster in three fields: searching, sorting, and modifying.*

1 Introduction

In its most basic form, a tree is a 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 tree: O(logn) and Red-Black tree: O(logn).

In order to test the time complexities of these trees, we solved two separate problems. The first problem involved a phone directory in which the user can add, modify, find, and remove names and their associated numbers. This was done to test the traversal, addition, deletion, and modification times of the trees. For the second problem, we considered a movie script word counter. For this problem, we fed a movie script to both trees. Then the script was processed, collected and sorted for each word. This was performed to test the sort, traversal, and addition times of the trees.

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). In the subsequent sections, we discuss 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, our results show which tree is faster and more useful and we explain why that is the case.

2 Background

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.

AVL trees are self-balancing, i.e., at most, the level of the nodes will differ by at most one. If this is not the case, then the AVL program will take the initiative and balance itself.

Red and Black tree, 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 speed because it is more balanced where Red and Black has better insert and deletion speed. The AVL tree stores heights for each node (O(n)), while Red and Black tree just stores 1 bit for each node (O(1)).

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 one uses a structure like a list, stack, or a queue, the search and insert times will be much slower.

As far as lists are concerned, one will be able to sort them based on names but you cannot jump around them unless you implement something like a binary search. The main reason a binary search would not work is that one would have to bounce around "chopping" the list in half each time to find the person. A stack wouldn't work because one 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 data set.

So, in the end, an advanced tree is the only logical thing to work with since it is possible to look down one side of a tree. A basic tree would not be sufficient because one would be lowering the 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 made a script analyzer that takes in a large sum of words (which can repeat quite a bit) and calculated how many times they appear in the script. We chose to solve this problem because we thought it was a great example of seeing how a Red and Black Tree can handle repeating data along with inserting a large amount of data.

Some may argue that a map would be a better data type for this problem, but 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. Since our goal was to compare the performances of trees, that was our chosen data structure.

4 Related works

The basics of the trees are constructed from sanfoundry.com[1] and TutorialHorizon[2]. While most of the algorithms came from geeksforgeeks.com[3, 4]. 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.

5 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.

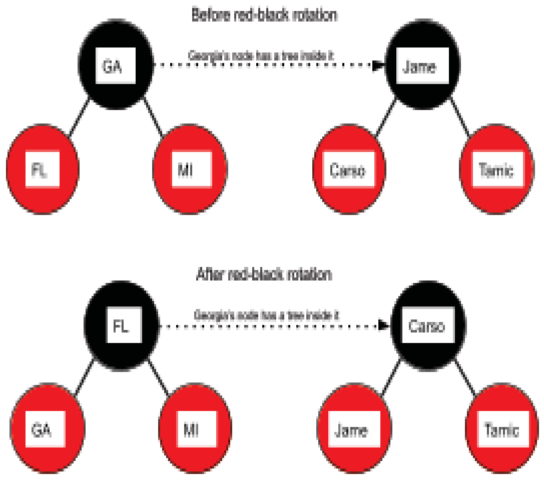

Before red-black rotation
Georgia's node has a tree inside it
GA
Jame
FL
MI
Carso
Tamic
After red-black rotation
Georgia's node has a tree inside it
FL
Carso
GA
MI
Jame
Tamic

Fig 1: This figure is showing how the tree will take in a few node and how it will reoriented itself so that it is self balancing

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

6 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 in 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 were then stored in String nodes because through our trial error that is what was easy to work with because of the data that we were 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 basic tree programs.

For the first problem, we constructed a phone directory and used two different types of trees - an AVL tree and a red black tree to see if one tree had a better time complexity than the other. The red black tree that was used in problem one has another red black tree placed inside of each node. What this means is when we grab a person we check their state, travel to the state node and delve further. Once the State node has been picked we then access the person red black tree. It sorts people based on name or phone number depending on if there is another person with that same name. Once it places the person it will then sort itself using the red black algorithm.

7 Implementations

Phone Directory with Red Black Tree

The red black tree is similar to the sanfoundry[1] implementation, but we took it a step further. We implemented a multilevel tree, i.e., each node inside the state tree is a small tree itself.

The way the program works is it takes in information from the GUI, such as a name, state, phone number and age (optional), and pulls in a state node. Once it pulls in a state node, it opens up its information and creates a person object out of the rest of the information, and inserts it into the state tree.

To start the tree, if a tree has not been made before, it will create a temp person to place as a temporary root node. Once a tree has been made, it treats the person that was just created from the GUI information and places it in like a Binary Search Tree. From there it moves into where the red black tree is a lot more organized.

If the parent is red then we start sorting and making rotating nodes with other nodes starting with grand. On the other hand if the grand node is in the right place, we check the great node. Once everything has been sorted and it has turned into a semi complete tree, we set the root to black and return to the GUI.

Phone Directory with AVL Tree

The first step is to get the program to read the text file. After that it is a matter of how the information is stored in the nodes and how the nodes are then compared. The information in the nodes are compared using the "compareTo" method. The compareTo method compares the two strings and determines which node it should go to. If it is greater than it goes to the right and if it is less than if goes to the left. The last part is the display. You go deep in the left subtree until you find the leaf node, when that has been exhausted then it starts on the right leaf nodes.

Phone Directory with Binary Search Tree

The BST implementation found on TutorialHorizon[2] served as a control tree, after modification to work with all-String nodes, either using the String.compareTo() method or parsing String phone numbers into Long as needed for sorting purposes.

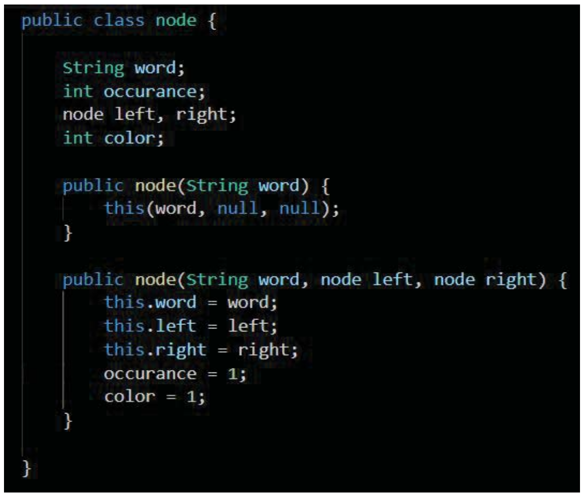
The Binary Search Tree works in a similar fashion as its cousin, Binary Search, dividing its dataset by two until the target index has been reached.

Script Analyzer with AVL Tree

We followed a similar approach to reference [4] tree, however implemented it differently. In [4], they used a tree that focused around an int key. However, for the movie script problem we used a string key so the keys can automatically balance alphabetically. We compared strings using compareTo() in the insertion and rotation methods. In normal BST insertion one typically ignores the duplicate. The main point of our second problem was to get the frequency of the words. So rather than just do nothing when dealing with a frequency, we increment an int variable named frequency, which is located inside the node.

Script Analyzer with Red Black Tree

For this problem, our first step was to create a red black tree. For this tree, we made our node initially send in a string which is the word. From there, it sent another constructor which took five parameters: word, occurrence, color, left node, and right node.



```
public class node {

    String word;
    int occurance;
    node left, right;
    int color;

    public node(String word) {
        this(word, null, null);
    }

    public node(String word, node left, node right) {
        this.word = word;
        this.left = left;
        this.right = right;
        occurance = 1;
        color = 1;
    }

}
```

Fig.2: This is an example of our node class which stores the name, occurrences, children nodes, and the color of a word in within the individual spots of the red-black tree

After the tree was made and had the proper nodes, our next step was to create a search that could find the word in the tree. To do this, we just used a simple binary search based on the ASCII value of the first letter. Next was creating the different types of tree traversals just to check if everything was being sorted into the proper spots. To save some time, we did the same thing with counting how many nodes are in the tree (using PreOrder traversal). The last step in making our tree was to be able to use files to scan in the script and to create some kind of output file with all the times taken to perform the tasks.

8 Experimental Results

Experimental Results(Red Black Tree Problem 1)

We tested three aspects for problem 1 to test for the time complexities for red black tree calculations. We tested how long it would take to insert 13 people and to see if the time goes up as each time we add another person. Then we tested how long it would take to search for certain people that we knew would fall further down the tree such as James or Sam. Then finally we gathered the time table for inorder traversal with 13 people .

As we can see in the figure 4, the time it takes to insert people is almost instant. The most that we were getting for times were 0-1 milliseconds. Whereas for searching it depends on how many people we have to search through.

Experimental Results(Red Black Tree Problem 2)

To test for the time complexities for the Red Black Tree for problem 2, we tested four different aspects. The first one was how long it took to insert an entire movie script (the Bee Movie). The second test was the time taken to do three different traversals. The third test was to search for three separate words to test the search time. The words tested were: bee, the, and zero, with zero not appearing in the script. The final test was finding the time to get the size of the tree. All of the results for these tests can be seen in Fig.5.

Experimental Results(AVL Tree Problem 1)

We recorded the time to test how fast the trees were being formed. Even though we used the same file to create the trees, there were different times.

Experimental Results (Control BST Problem 1)

Similar to above, we used nanoseconds to test the insertion, search, and display times of the control binary search tree.

While the initial insertion was close to 600,000 nanoseconds, the other twelve insertions dramatically dropped below

100,000 and continued to stay under that number, with the proverbial floor being 12,301 nanoseconds and the ceiling 83,244, the insertion directly after the initial, costlier insertion.

The BST's performance is comparable to the Red Black Tree, matching the sub-zero millisecond search times while having faster search and display times.

Compared to the AVL Tree, however, the BST is far more efficient, with all the AVL Tree's times in the millions of nanoseconds compared to the max BST time of just under 600,000 nanoseconds.

Experimental Results (AVL Tree Problem 2)

Figure 6 shows the time log for the AVL tree for the second problem. The insert method in the Red-Black tree is more efficient than the AVL one. The AVL takes ten times as long to insert the same script, which is the bee movie script in this situation. The search times are very quick in both trees as they range from 0-2 milliseconds. The traversals in the Red-Black are also more efficient as they take half the time as the AVL traversals.

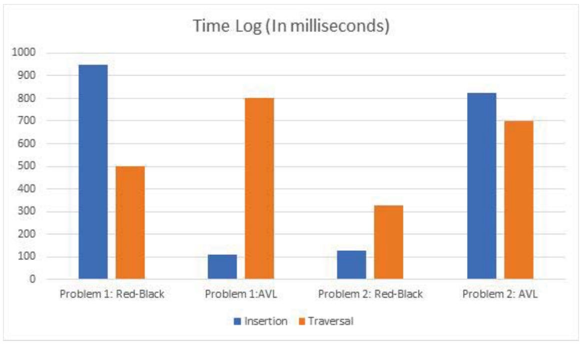


Fig.3: This is a chart that shows off the average insertion (blue) and traversal (red) times with both problems and both types of trees.

10 Conclusions

To conclude problem 1, we could see that the regular binary search tree that we used was actually faster than both the red-black tree and the AVL tree.

For problem 2, we can see that the red-black tree is faster than the AVL tree in pretty much every action, and if it was not faster, then they were tied.

11 References

[1]https://www.sanfoundry.com/java-program-implement-red-black-tree. [Accessed Nov. 4, 2018].
[2]https://algorithms.tutorialhorizon.com/binary-search-tree-complete-implementation/. [Accessed Nov. 6, 2018].
[3]https://www.geeksforgeeks.org/red-black-tree-set-3-delete-2. [Accessed Nov. 6, 2018].
[4]https://www.geeksforgeeks.org/avl-tree-set-1-insertion/+. [Accessed Nov. 7, 2018].