**Tree Experiment Summary**

Writing a binary tree is essentially a rite of passage in computer science. They have many real-world use cases such as quick lookup times and are even the backbone of the C standard library Hash Map. When referring to binary trees, one always thinks of the fast Big O(logn) search time. However, this is only true if you have a balanced binary tree. An unbalanced binary tree can eventually degrade to Big O(n) and might as well be a linked list if it isn’t balanced properly. The purpose of this experiment today is to pit a normal binary search tree against a self-balancing binary tree such as red-black tree or an AVL tree. Mine and Loic’s experiment will run both trees through the same three datasets and compare how they perform on each individual run. We will then prompt the user to test how well searching works for both trees such as the number of comparisons it takes to search for the requested number.

Loic and I approached this program in a similar manner to the sorting experiment and tried to reuse a lot of the same code that we had used in the prior assignment. This strategy saved us a lot of time in some areas, but also led to some massive headaches where some things didn’t mesh as well as we had hoped. For example, we were not able to implement the trees in namespace files and instead had to use header files and declare the actual objects. This was a slight hiccup to work around. We also borrowed some strategy that I used on the sorting experiment which involved having a “Tree Experiment” object that controls doing an instance of the experiment and then having a “Tree Experiment Controller” class that creates multiple experiment objects to dictate the program and handle running multiple tests.

We wrote another C++ file to generate 1,000 random number for us between 1 and 10,000 and made it to where we would follow Dr. Johnson’s suggestion of numbers for the root. As a result, our first data file has 5,000 as the first value which will be assigned to the root node, our second data file has 2,500 has the root value, and our third data file is truly a random assortment of numbers. Additionally, there is a method in place to generate random numbers for the data set as well without having to rely on our number list. Loic and I both collaborated on the binary search tree file whereas he did more of the work and sourced some helper code to get the red-black tree working. It was then my job to cram them into the format I had with the controller and experiment classes.

When it comes to counters and comparisons, we kept computation counters to keep track of how many comparisons it took to insert a number. Since the way we handle a balanced binary tree was somewhat left to our discretion, we chose to increment the counter for comparisons that are also performed during the rotations. As a result, the balanced trees tend to perform more comparisons per insertion. As far as the experiment, we did 10 runs on each of the 3 data sets for both a binary search tree and our red-black tree. As of typing this, we seem to be having a problem getting the averaged results, so at the moment, each individual test result is placed in the terminal similar to our sorting experiments so you can clearly see the results on a test-by-test basis for both trees.

The results of our tree experiment were interesting because they did not go exactly how we were expecting with the exception of tree height. Tree height was right in line with our expectations as when we were loading the trees with datasets containing 1,000 numbers, the red-black tree far outperformed the generic binary search tree. In fact, the red-black tree was around 2/3rds the height of the binary tree on average. However, one result in particular that was a bit surprising was that when it came to searching for a number, the results were almost identical between the two trees. The red-black tree seemed to perform slightly better on larger numbers as they approached the top end of our data set’s number range, however on lower and middle number ranges, the binary search tree was just as fast or in some cases faster than the red-black tree. Another result that was mildly surprising was that the time difference between loading the values into the different trees was negligible. Our initial hypothesis was that the red-black tree would be considerably slower because of all the rotation work. However, it became clear that the time lost to rotations on the red-black tree made up for the longer traversal times when inserting on an unbalanced binary search tree, so we found that interesting.

This was a really cool project and I enjoyed working on it. I will say though, both Loic and I kind of agree that with more time we feel we could have done a better job on a few more things and been able to go above and beyond a bit more. On paper we had plenty of time to work on this, but like many others at this time of the semester we had a lot of projects and classes to juggle so we had to prioritize a bit. As a result, we thought we would be able to just replace our sorting algorithms with the trees and otherwise everything would mesh pretty well together so we left it a little late. Still, I think we put together a good program overall.

A picture containing chart

Description automatically generated

Chart, bar chart

Description automatically generated

Chart, bar chart

Description automatically generated