Song Chen
Johnathon A Van-Atta

CS240 Lab 6 Report

The goal was to create and run tests that would compare the run times of basic operations for balanced and unbalanced binary trees.

A templated container that would use a binary tree and implement simple insert, remove, find, and display functions. insert, remove, and find take a reference to a value, and use it for the respective value in the tree was created(seeing as values aren't repeated, there's only one response per possible value).

For the most part, the tree is implemented using recursive functions. Insert, find, and display all use recursion to do their respective operations. Remove has two helper functions, Locate and FindMax that utilize recursion to help remove find the node it wants to delete, and what to replace it with, if needed.

Special care had to be taken where the node to remove was the root node, in those cases you had to be careful because you couldn't make your parent simply point to your child, instead you had to promote the child's value to the root and point past the original child.

Anyone trying to use our container needs to be aware we take advantage of the <, >, ==, and = operators of the templated type. Without versions of these operators defined, the program will not compile. Also, any two items of the templated class who are equal by the == operator are viewed as indistinguishable by the container, the first instance will have precedence over all others inserted afterwards.

Four tests were ran, each consisting of three trees. The 1st tree refers to the bars labeled sorted in the charts. The tree consisted of integers from 0 to up to one less than the number of nodes in that particular tree. The integers were inserted in sorted order, resulting in the “worst” case unbalanced tree. The 2nd tree refers to bars labeled random in the charts. This tree consisted of the elements 0 to 9 inserted in random order, utilizing c++11 random_shuffle function. The last set of bars labeled balanced refers to the last tree, a balanced tree. This tree was created by sorting an array such that inserting the elements in order would result in a balanced tree at every insert.

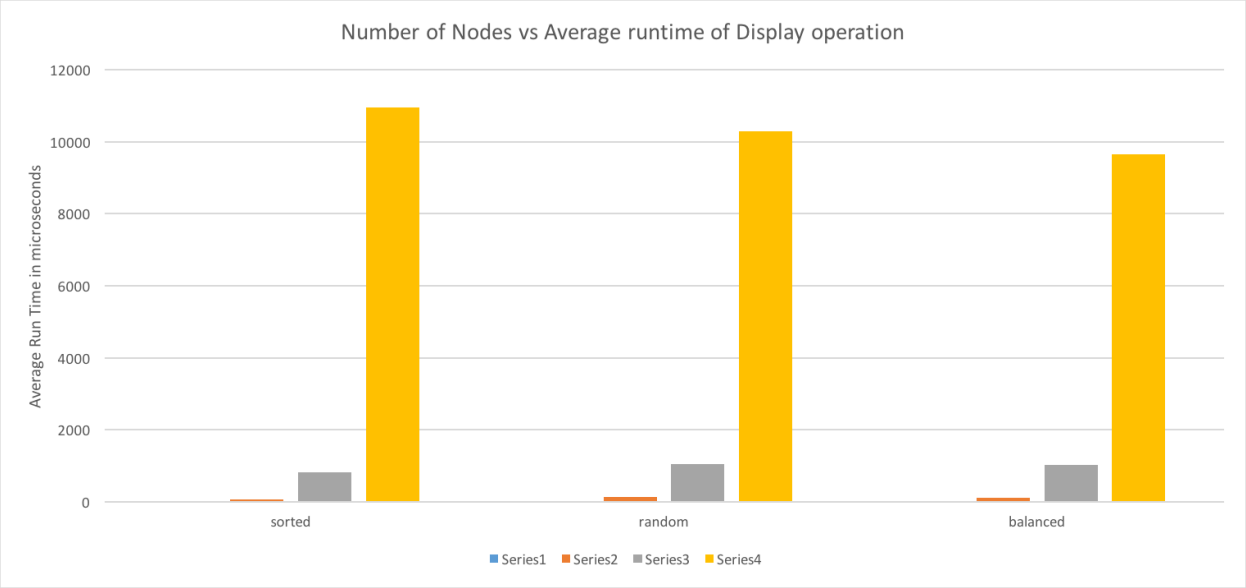
The first test ran was the display test. First ten nodes were inserted into an empty tree. The current time was recorded and the display member function was called 100 times and then the current time was taken again. Taking the difference of these two and dividing by 100 resulted in the average run time for a display operation for a tree of 10 nodes. The above steps were then repeated for 100, 1000, and 10000 nodes.

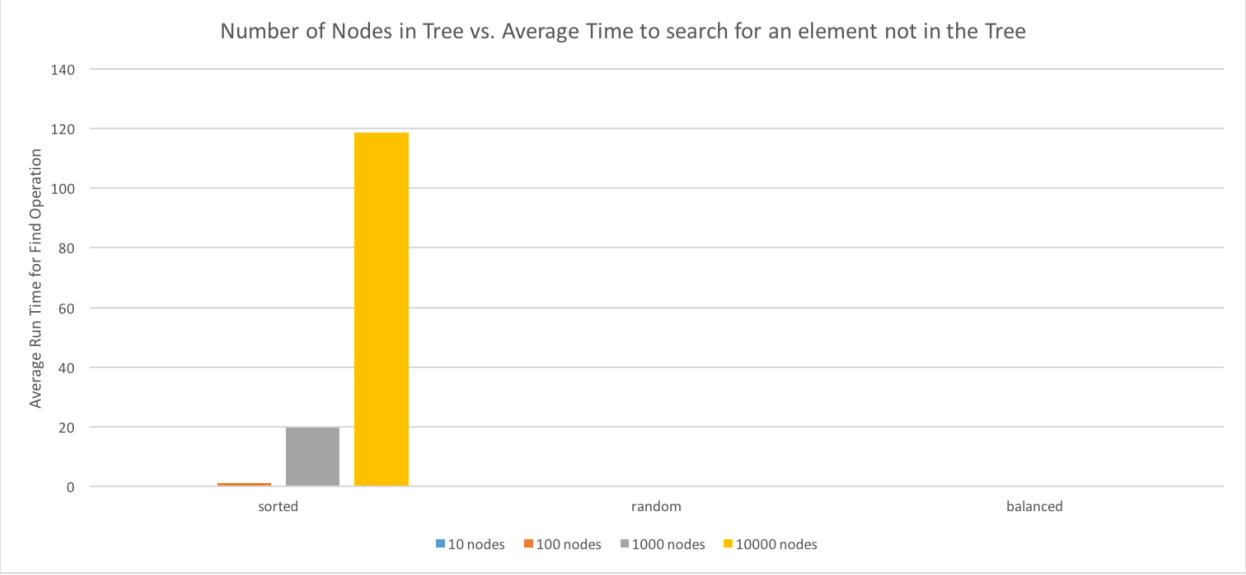
The next test ran was insert operation test. Like the display test, this test was done for 10, 100, 1000, and 10000 nodes. First the current time was taken, then the nodes were inserted, and then the time was taken again. Taking the difference and dividing by the number of nodes inserted results in the average time it takes to insert a node.

The last two tests were the find and remove tests. These tests are also done with 10, 100, 1000, and 10000 nodes. For the find tests, the current time was recorded, 100 find operations were performed for an element not inside the tree, the current time was taken again and the difference divided by 100 results in the average time it takes to find a node for a given number of nodes currently in the tree. The same tests were run for the delete operation. The

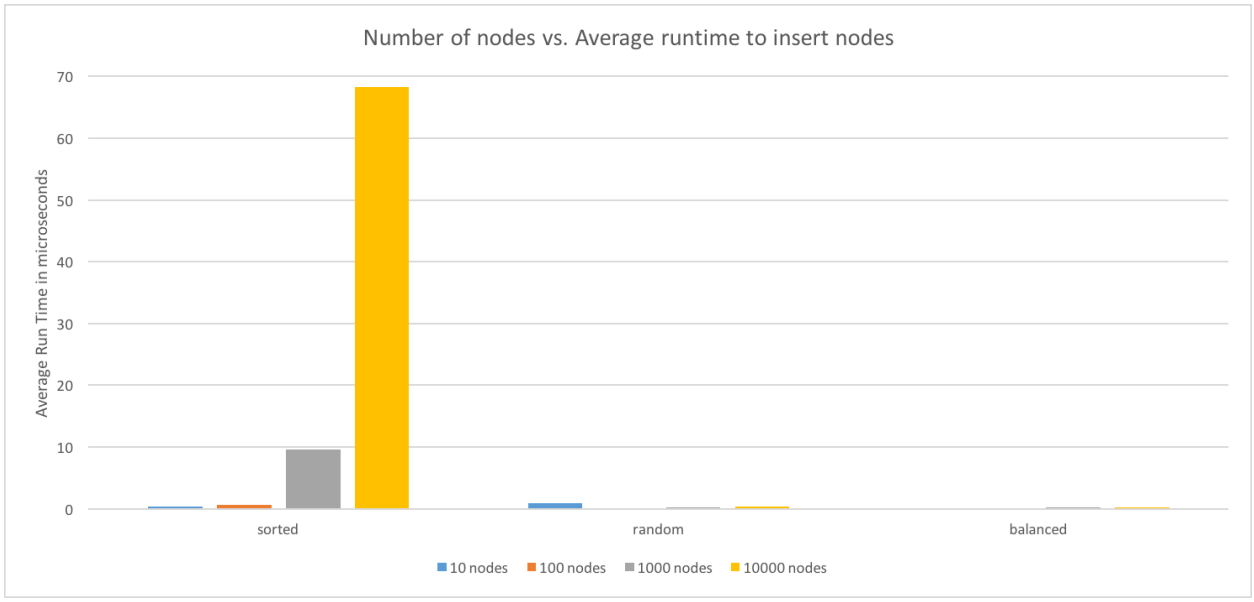
reason the find and remove operations were performed for an element not in the tree was because these cases represented “worst cases” as the entire depth of the tree would have to be searched to find or delete the element.

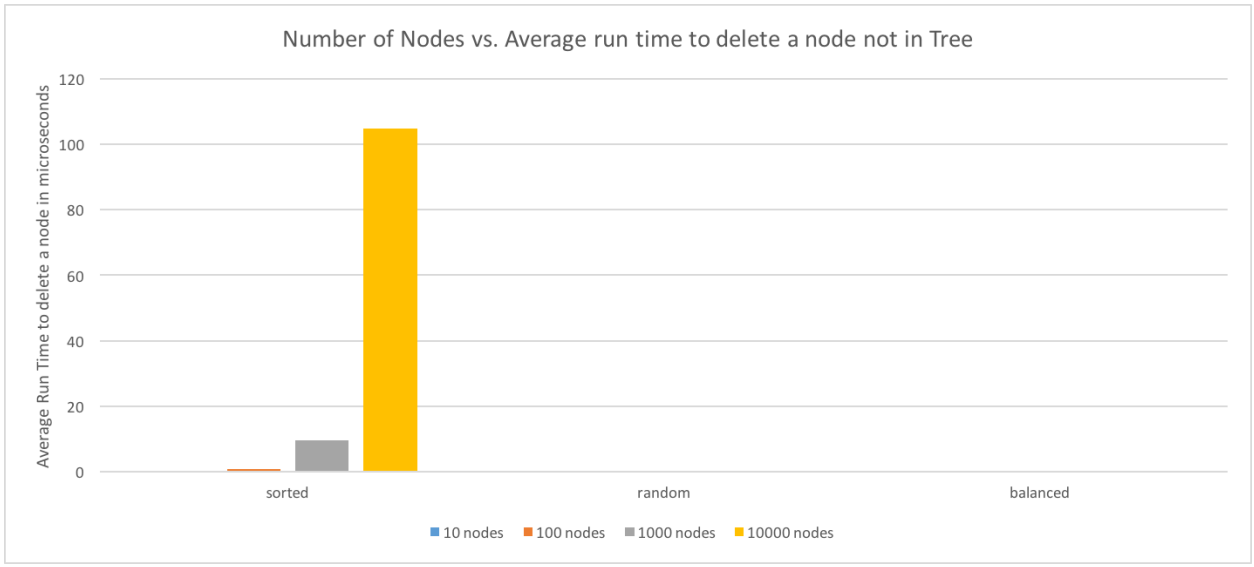
We hypothesized that for every tests besides the display tests, the runtime of the operations for a “sorted” (the worst case unbalanced tree) would grow much faster than the balanced tree. We’re basically correct. To see this you can refer to the charts. However, we expected the random tree’s runtimes to grow at least somewhat faster than the balanced trees. But no significant differences, even for 10000 nodes, was found. Since the Big Oh run time to display the elements of a binary search tree is order n for both balanced and unbalanced trees, we did not expect the run times of the three trees to be significantly different.





(excel was being glitchy, the bars are there for random and balanced, they are just really small)





(excel was being glitchy, the bars are there for random and balanced, they are just really small)