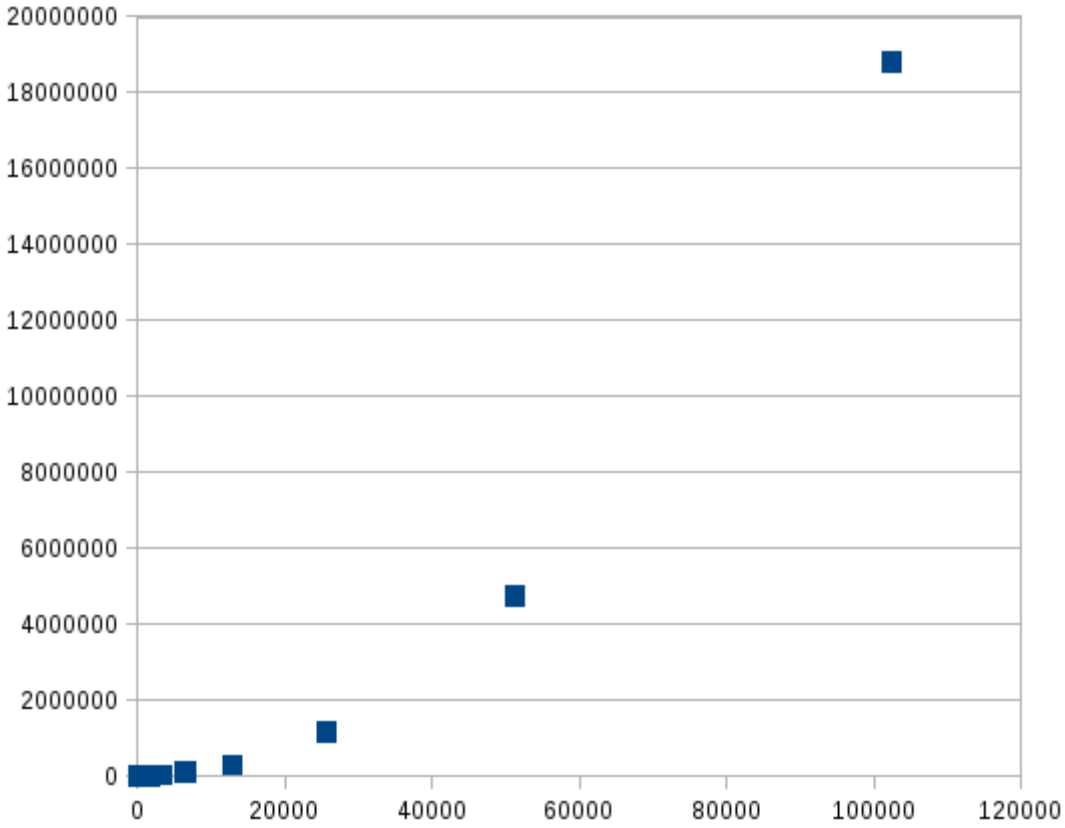
Josiah Bailey and Shawn Bailey
Lab 6 Report

We set out to test the time complexity of our BST methods with a worst case, average case, and best case analysis. To time each case, we used #include <sys/time.h> and calculated time values.

Note, the x axis is the number of nodes we have in our BST, and the y axis is the amount of microseconds it took to run each respective method. The axis labels were not cooperating for LibreOffice Calc. Sorry in advance!

Worst Case

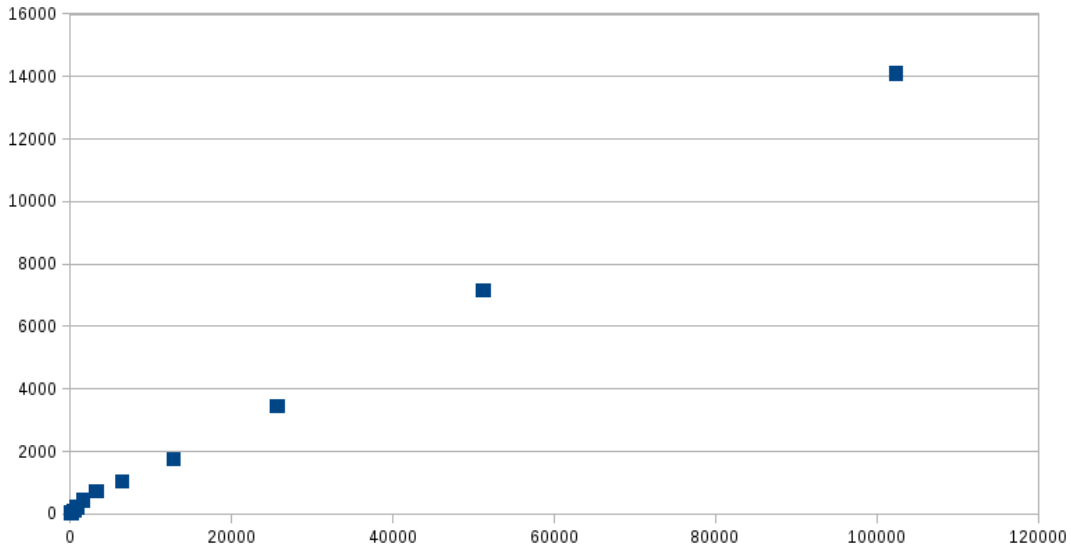
(Insert method) We created a totally unbalanced BST, starting with a root of zero and adding 1 to j in ascending order, where j started at 100 and doubled until it reached 102,400. We timed how long it took for the insert method to build the BST for each j value. The result came out linear, as we argue with the following graph:



Note that the graph looks like O(N^2) which is to be expected because we timed the building in a for loop of size N, implying each insert in the for loop will have O(N) time complexity. Thus, the insert time complexity itself will be bounded above by O(N) or linear growth.

(*Find and Remove methods)* For each unbalanced tree we created, we reasoned that the worst case would occur if it had to search the whole tree for a value not there. With this in mind, we found and removed such a value (in particular j+1). We did this in a loop 100000 times, because the operations are actually fast, and wanted to get some times that would make a relevant graph. With that in mind, here

are the results:

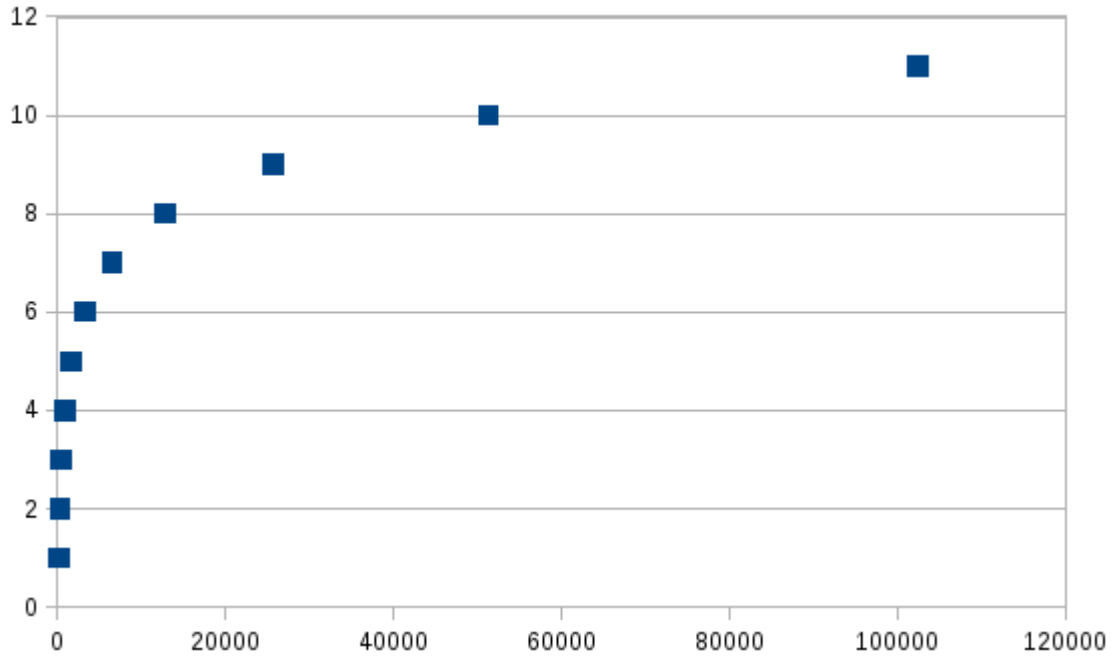


Note that in this case the fact that we put the remove and find functions in a for loop does not effect our time complexity analysis, because we used a constant coefficent of 20 for our loop, so it will gives us O(20N) time complexity, which means in the worst case, our find and remove methods are O(N).

Best Case

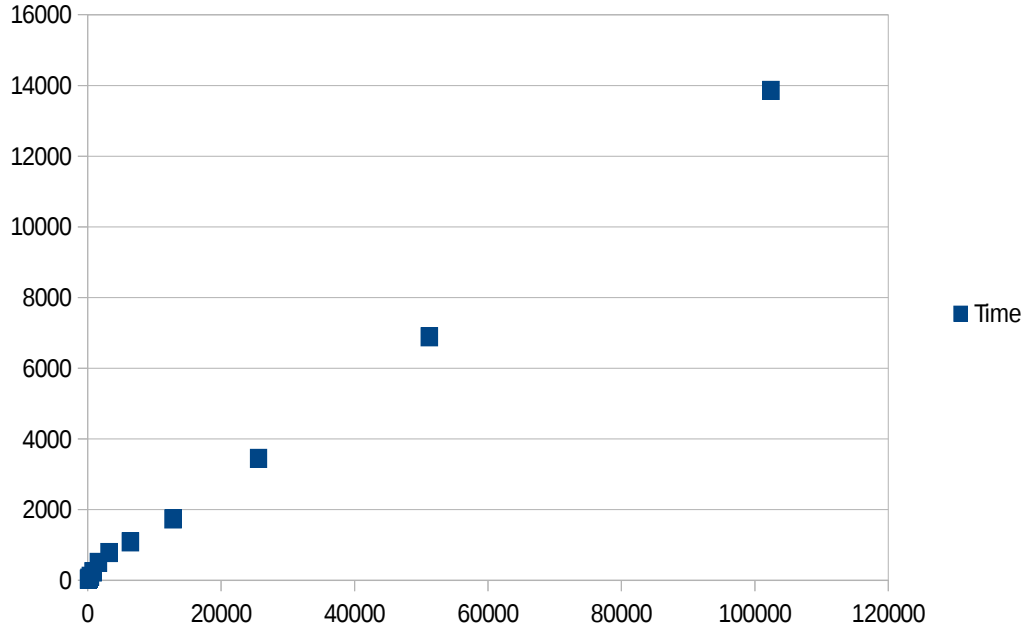
We decided that the first non-trivial example of the Best Case, would be with a balanced BST. We did not set out to analyze the time complexity for a single noded or empty tree, as that seemed too trivial.

(Insertion Method) We created a recursive function which would build balanced trees, and in the same manner, we built many trees for different j values, starting at 100 and again ending at 102,400, doubling each time. Here are our results:



We can see clearly that the time complexity of our insertion method for a balanced tree is O(lg N). In this case, we did NOT put the recursive function in a for loop of size N (as we did with our worst case building analysis), which is why this plotting came out to be O(lg N) and not O(N lg N).

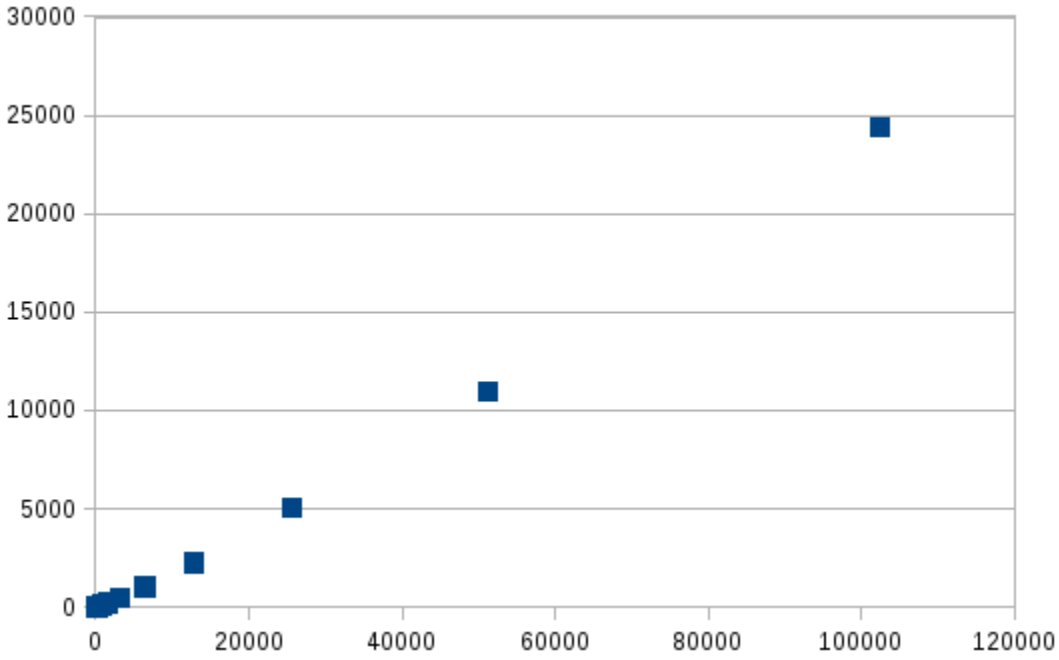
(Find and Remove Method) We analyzed find and remove in the same manner as we did above, in a for loop of size 100000. Why such a big loop? Because the functions are really fast it turns out for a balanced tree and we want a better feel for how long they are taking on a significant amount of runs. By significant here, we mean that 1000,000 is going to be sufficiently large to influence the plots, relative to the size of the trees we implemented. We also tried to find and remove an element that isn't in the tree on purpose to avoid trivially fast cases, and get a feel for how they behave compared to an unbalanced tree. Here are our results:



The plot here looks linear, but is in reality O(N lgN). The argument is relatively straight forward. Note that the big 100,000 for loop will take O(N) time to get through. We saw earlier that find and remove have O(N) for an unbalanced tree. Here however, they appear to have O(1) time complexity. But, for our N here, lg N is sufficiently small compared to N, so we can view it as a constant coefficient in comparison to N, so O(lg N) is approximately O(1) here. The upshot of all this, the time complexity of the find and remove methods will be O(lg N) for our balanced trees.

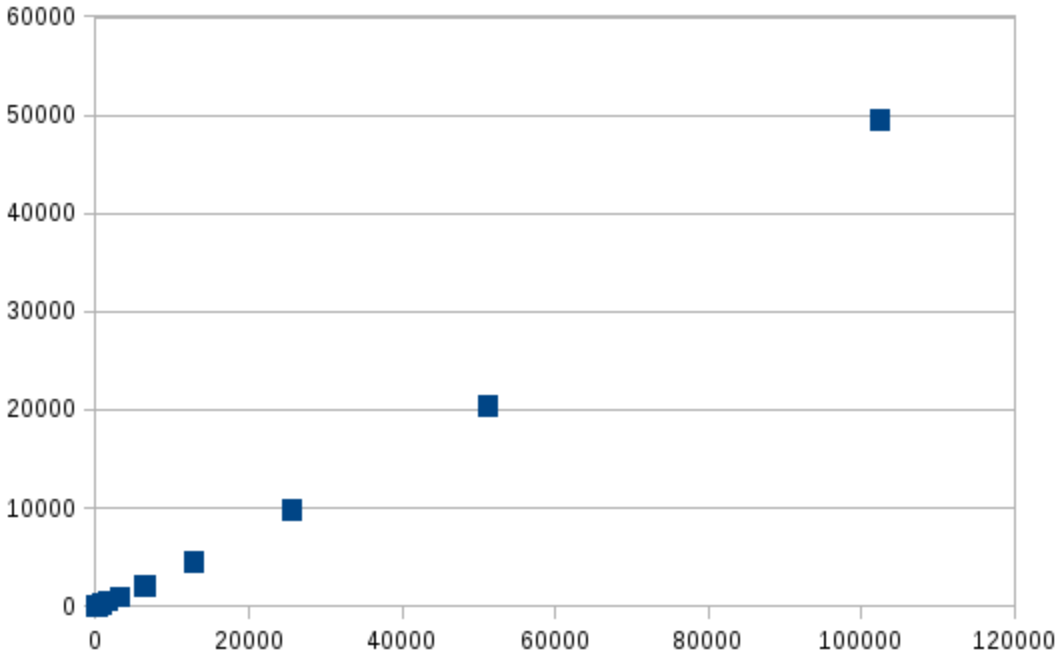
Average Case We figured that a good way to gauge the average time complexities for each method would be to create completely random BST's, giving us a more average idea of how long each function would take in general.

(Insertion Method) As mentioned, we simply inserted random values into a randomly generated root, but did it in a N for loop as we did with the Worst Case building. Here are our results:



At this point the details are similar to the find and remove analysis in the best case we did above. The graph here is O(N lg N) time complexity, which implies that our insert function has an average time complexity of O(lg N).

(Find and Remove) Here, we ran remove and find in a j for loop again, BUT we also reinsert the random node we removed. Why? Because for smaller j, it is more likely that the tree will become empty after we remove that many nodes, so we want to maintain the same tree size to not skew results for smaller trees. Also the random values are modulo'd with j to get reasonable values for each respective BST. The results are show below:



Here the graph is again showing O(N lg N) time complexity, which implies on average our find and

remove methods are O(lg N).

Conclusion:

All our results are within the expected time complexities for a BST in the worst case and average case. In hindsight, our “best case” analysis is the same as an average case, but we have a better idea of how long BST methods take for balanced versus unbalanced trees.