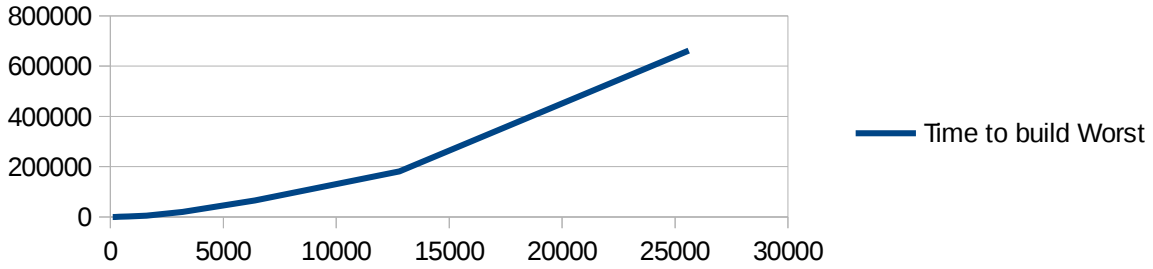
Josiah Bailey and Shawn Bailey
Prog 2 Report

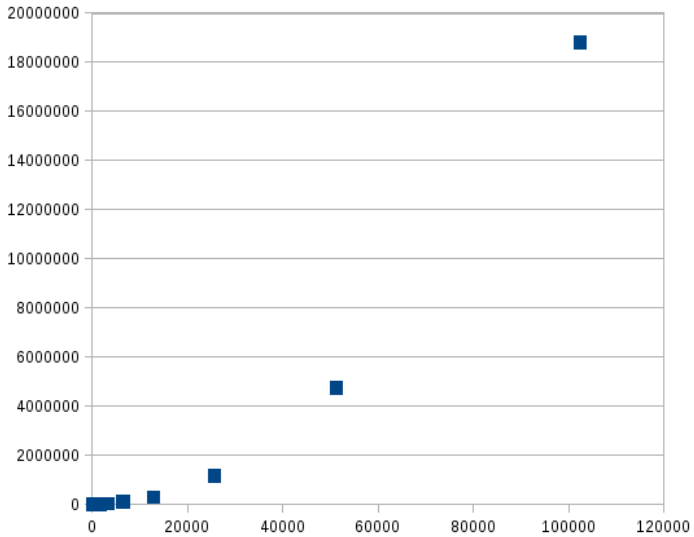
We set out to test the time complexity of our k-ary tree methods with a worst case and best case analysis. To time each case, we used #include <sys/time.h> and calculated time values.

Worst Case Insert

(Insert Method) We created an unbalanced k-ary tree, starting with a root of zero and adding 1 to j in ascending order, where j started at 100 and doubled until it reached 25,600. We time how long it took for the insert method to build the k-ary tree 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.

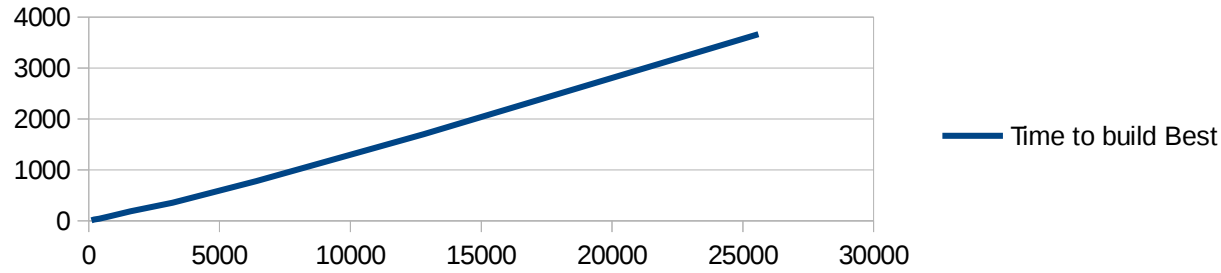


The above graph compares the k-ary tree worst case insert with the BST worst case insert. They both have the same runtime complexity.

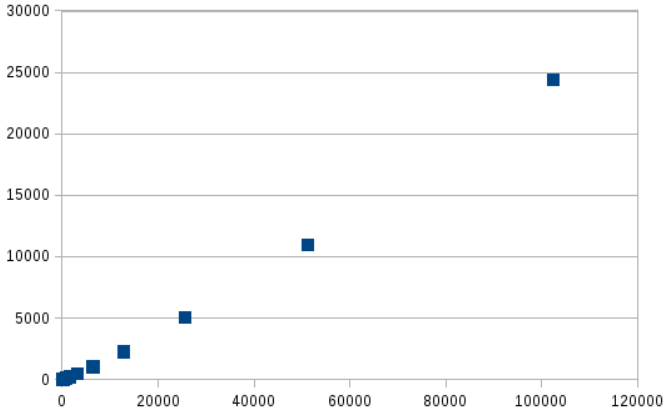
Best Case Insert

We decided the average case of adding random values into the k-ary tree would provide us with a close estimate to the best case of a balanced k-ary tree.

(Insert Method) As mentioned, we simply inserted random values into a randomly generated root, but did it in a for loop n times as we did with the worst case building. Here are our results:



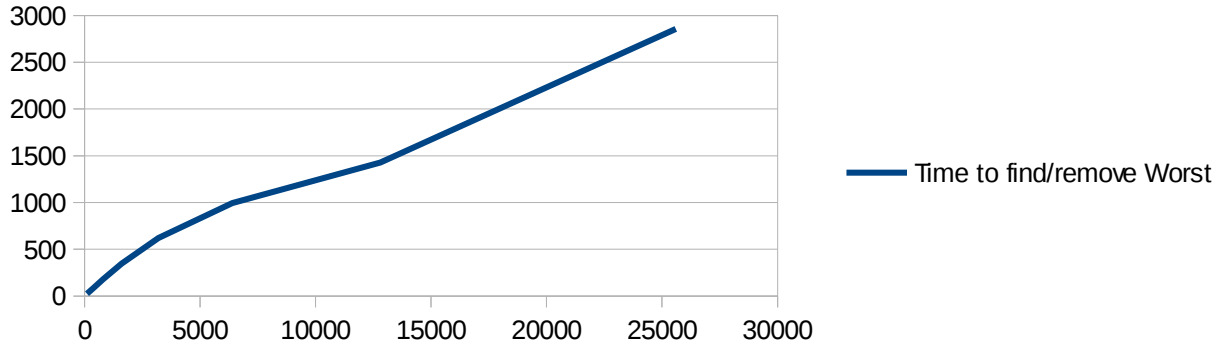
The graph here is O(NlogN) time complexity, which implies that our insert function has a best case time complexity of O(logN).



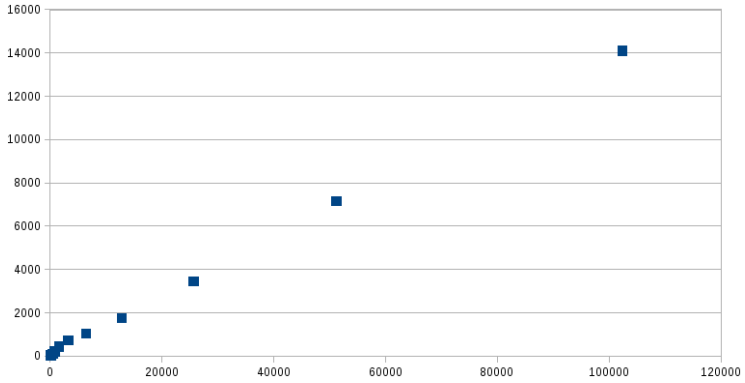
The above graph is a comparison of our best case k-ary tree insert to our best case BST insert. Both have a O(logN) runtime complexity.

Worst Case Find/Remove

(Find and Remove) For each unbalanced tree we created we reasoned the worst case would occur if it had to search the entire 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 25,600 times, because the operations are actually fast, and we wanted to get some times that would make a relevant graph. With that in mind here are the results:



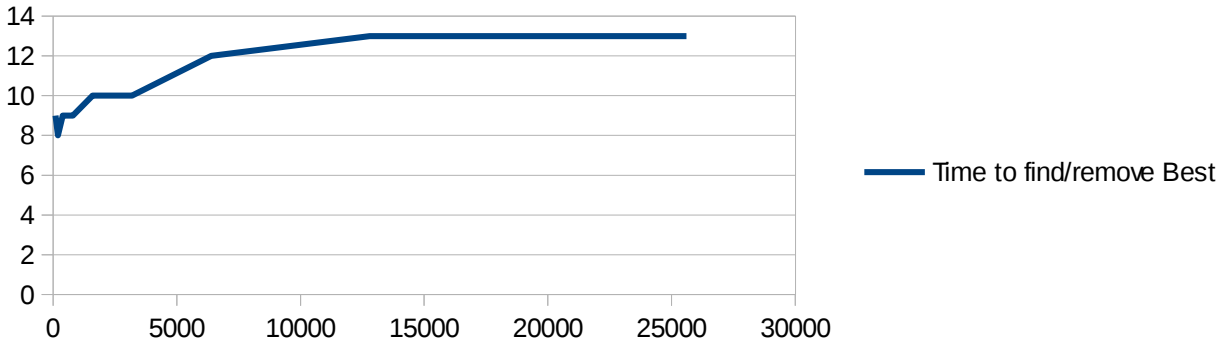
Note that in this case the fact that we put the remove and find functions in a for loop does not effect our time complexity analysis, because we used a constant coefficient of 20 for our loop, so it will give us O(20N) time complexity, which means in the worst case, our find and remove methods are O(N).



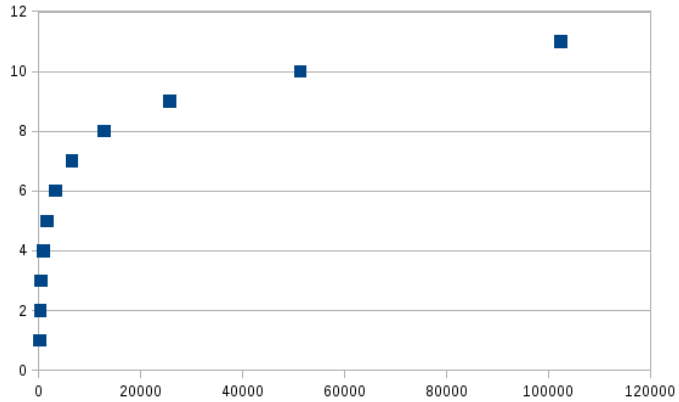
The above graph compares the worst case find/remove of our k-ary tree with the worst case find/remove of our BST. Both have O(N) runtime complexity.

Best Case Find/Remove

(Find and Remove) Here, generated a tree with randomly inserted values to simulate as close to a balanced tree as we can. We then ran find and remove 20 times. The results are below:



Here the graph is showing O(logN) time complexity.



The above graph compares our best case find/remove k-ary tree with our best case find/remove BST. Both are O(logN) runtime complexity.

Conclusion:

All our results are within the expected time complexities for a k-ary tree in the worst and best case. Note that both BST and k-ary tree's best case are O(logN), and the base of 3 or 2 does not effect it. The reason for this is because log base b (a) = ln(a)/ln(b) and for b = 2 and b = 3, these constants are disregarded.