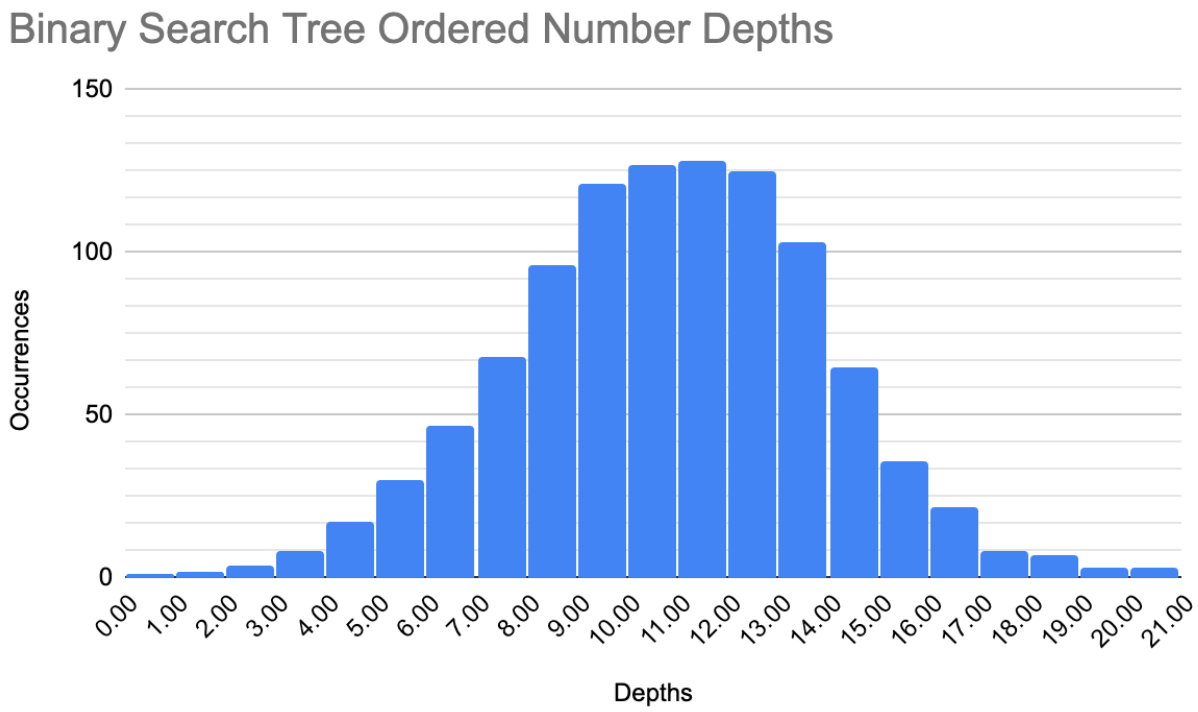
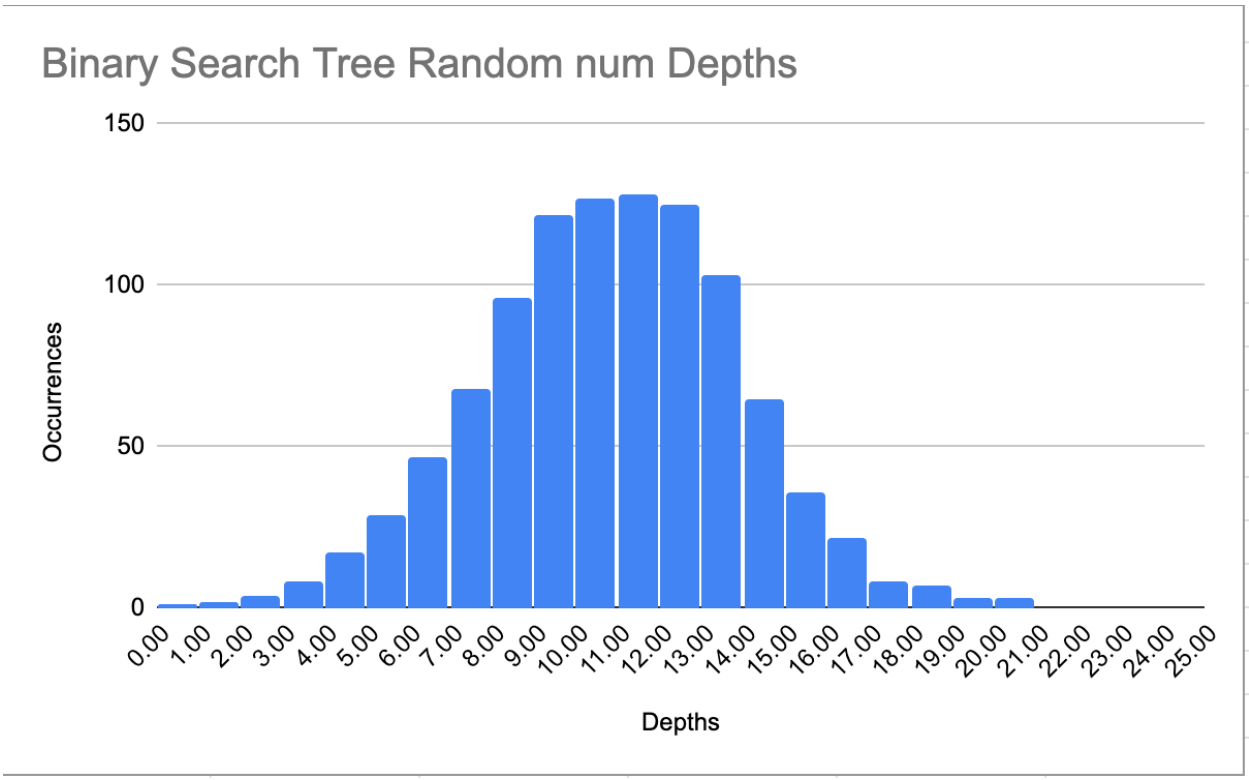
Lab 3 Graphs

Jimmy Mee

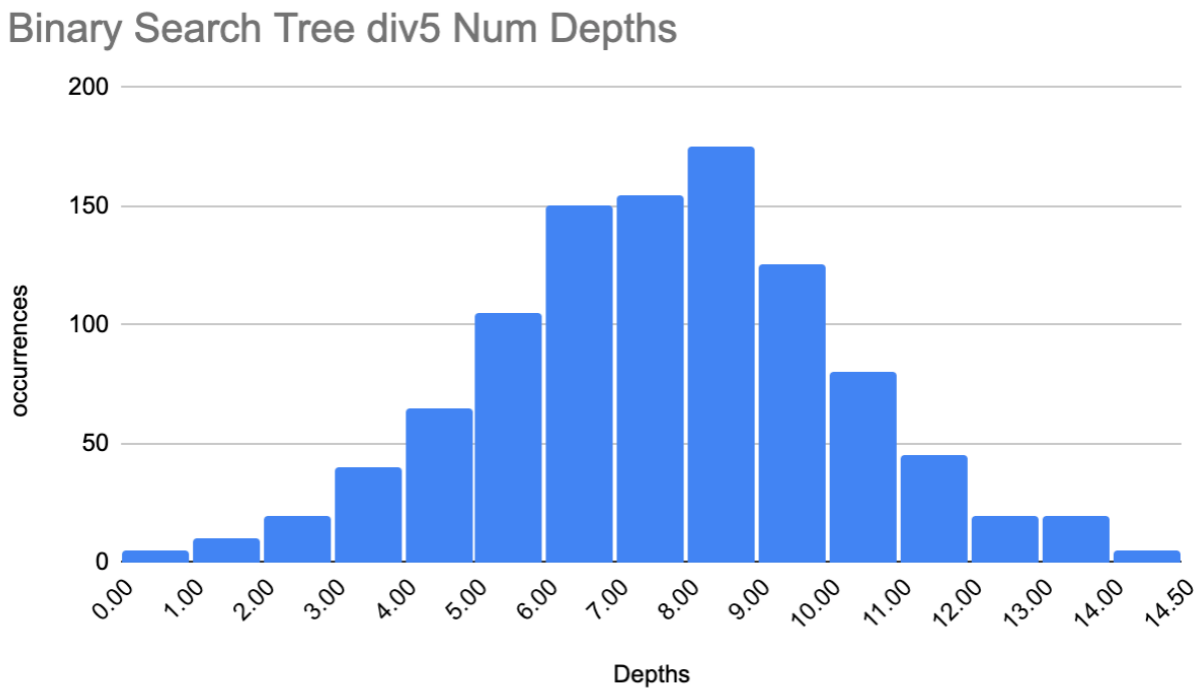
All tables are formatted the same. The X axis is the depths, and the Y axis is the number of times that a specific depth occurred



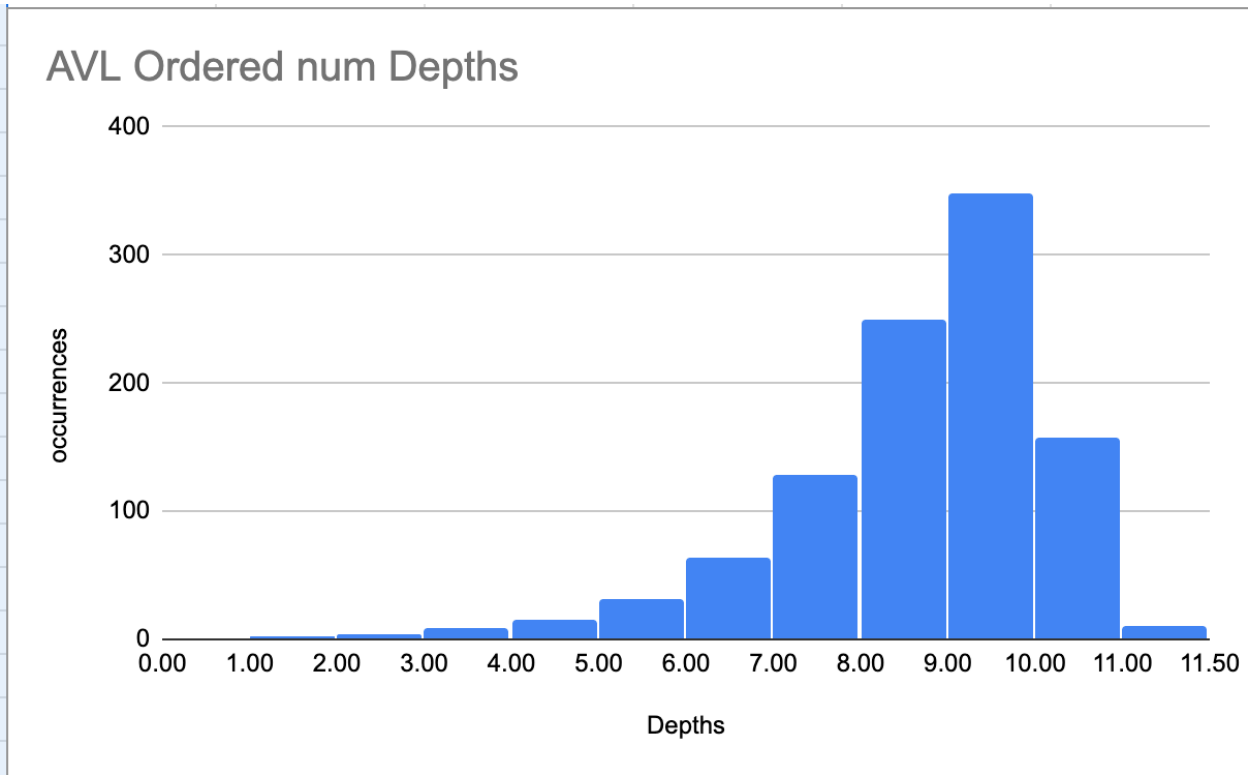
This data makes sense for an ordered index binary search tree. As more elements are added to the tree, they must be stored at deeper generations. In my case, I am comparing the ASCII value of two strings. For a binary search tree, the worst case would be O(N) number of nodes. The best case for a binary search tree is when it is perfectly balanced. The maximum tree depth would be O(logN) = 10.69. My tree is not perfectly balanced though, instead of the maximum depth being around 11 is up at 21. The minimum value is zero with the average being 12.



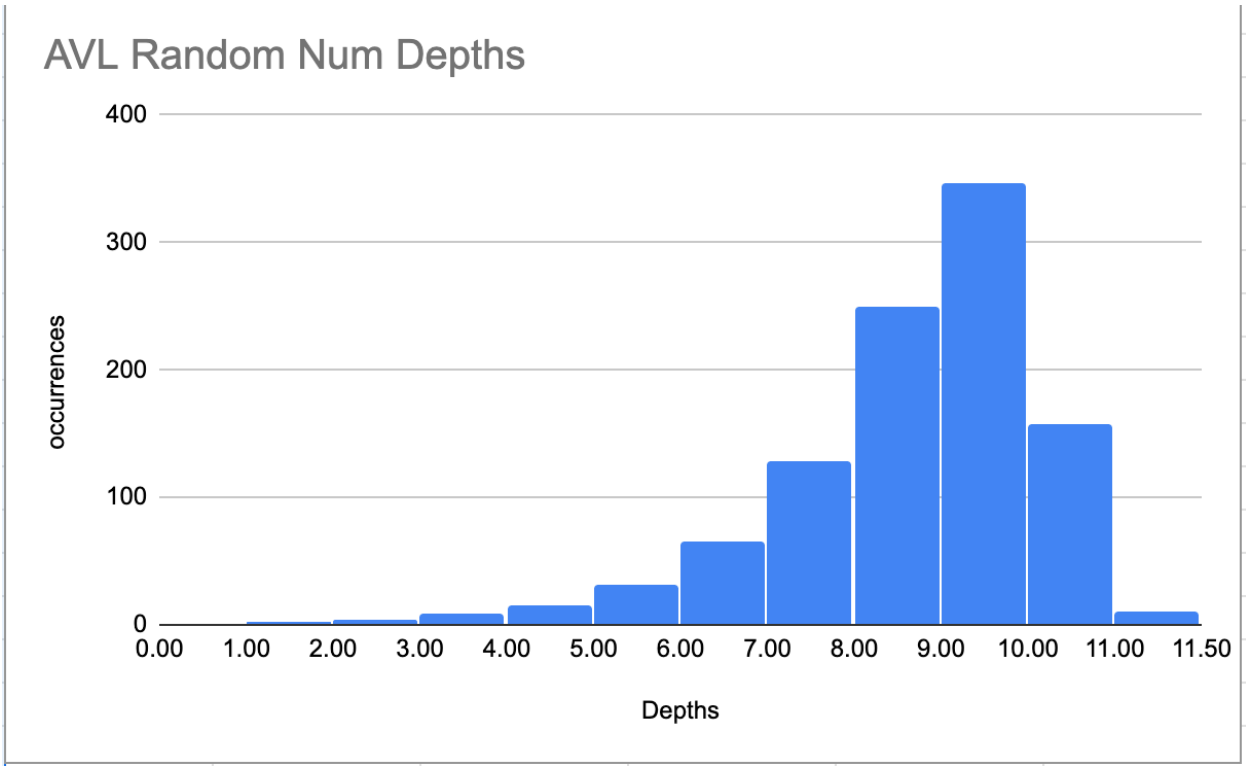
This graph is displaying the depths of the nodes in a Binary Search Tree, when values from my vector were inserted in a random order. As you can see, it only differs slightly from the first graph, and that was to be expected. As previously stated the worst case scenario, if nodes were inserted in descending order, would be O(N). Once again, the best case for a binary search tree is when it is perfectly balanced. If this were the case it would have a max depth of around 11, but since that's not that case, the max depth is 21 with a minimum of 0 and average of 12.



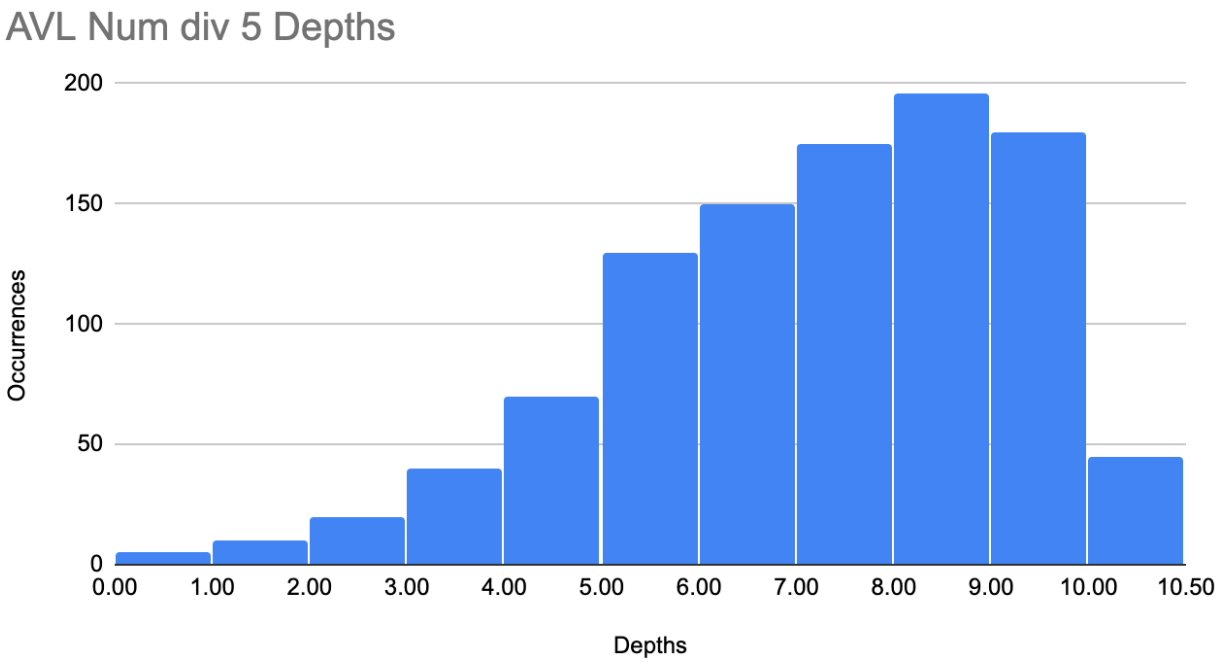
This graph is only 1/5 of the data set within a Binary Search Tree and it shows. The maximum median, and mode of all decreased. The Best and worst scenarios have not differed from the first two descriptions, because it is still a binary search tree. But, in the case the maximum depth is 14.5 with an average depth of 8. The minimum value remains 0.



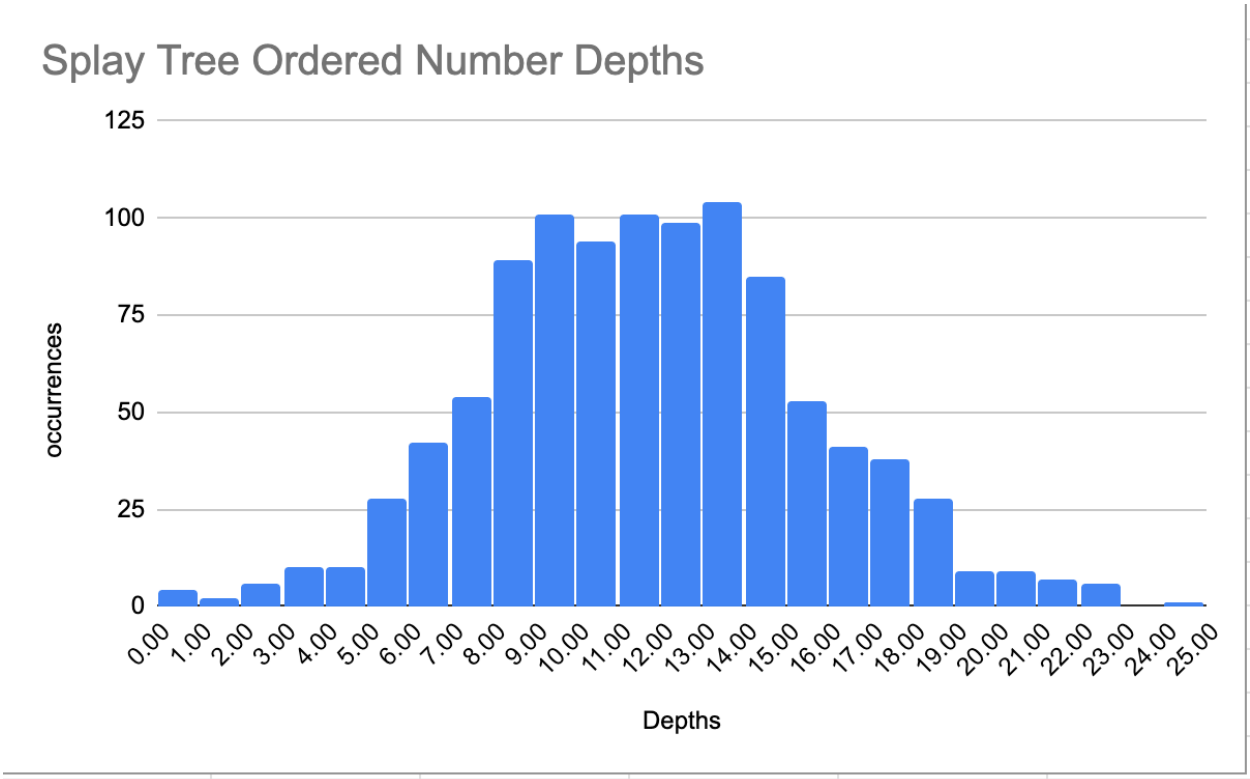
This graph depicts the depth of an AVL tree searching for my object that were inserted in numerical order. This graph seems very accurate for the current circumstances. The AVL tree has as self-balancing feature where the height of two subtrees of a node cannot differ by more than one. This ordered AVL tree has a maximum depth of 11.5 and an average depth of 9. It makes sense that these values are much smaller, because due to AVL's unique property, the tree tends to be much wider than it is tall. This Tree is more efficient than the Binary Search tree due to the lower max and lower average depth values.



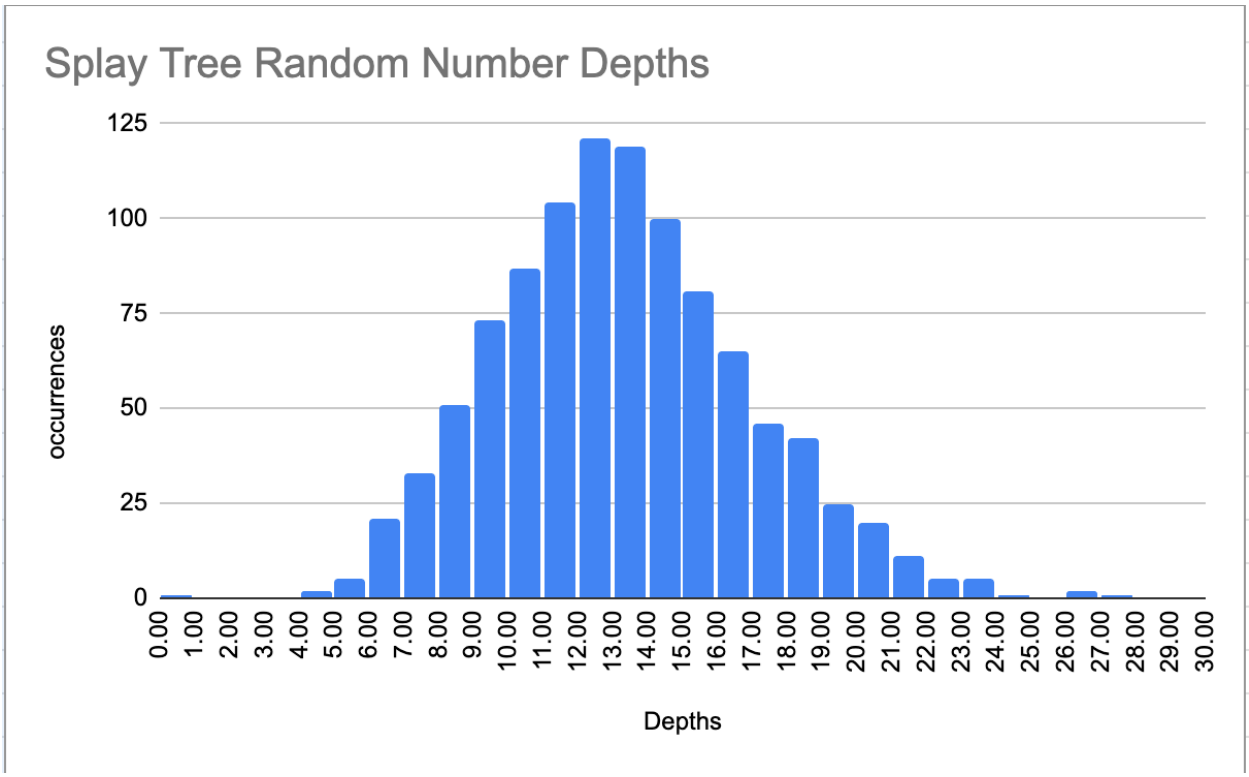
This graph is very similar to the on above, and that is to be expected. The min, max, and mode depth values all remained the same. Therefore, this tree is still more efficient that the binary search trees previously examined. AVL trees improve the efficiency of deleting, inserting and searching with a worst-case complexity of O(logN).



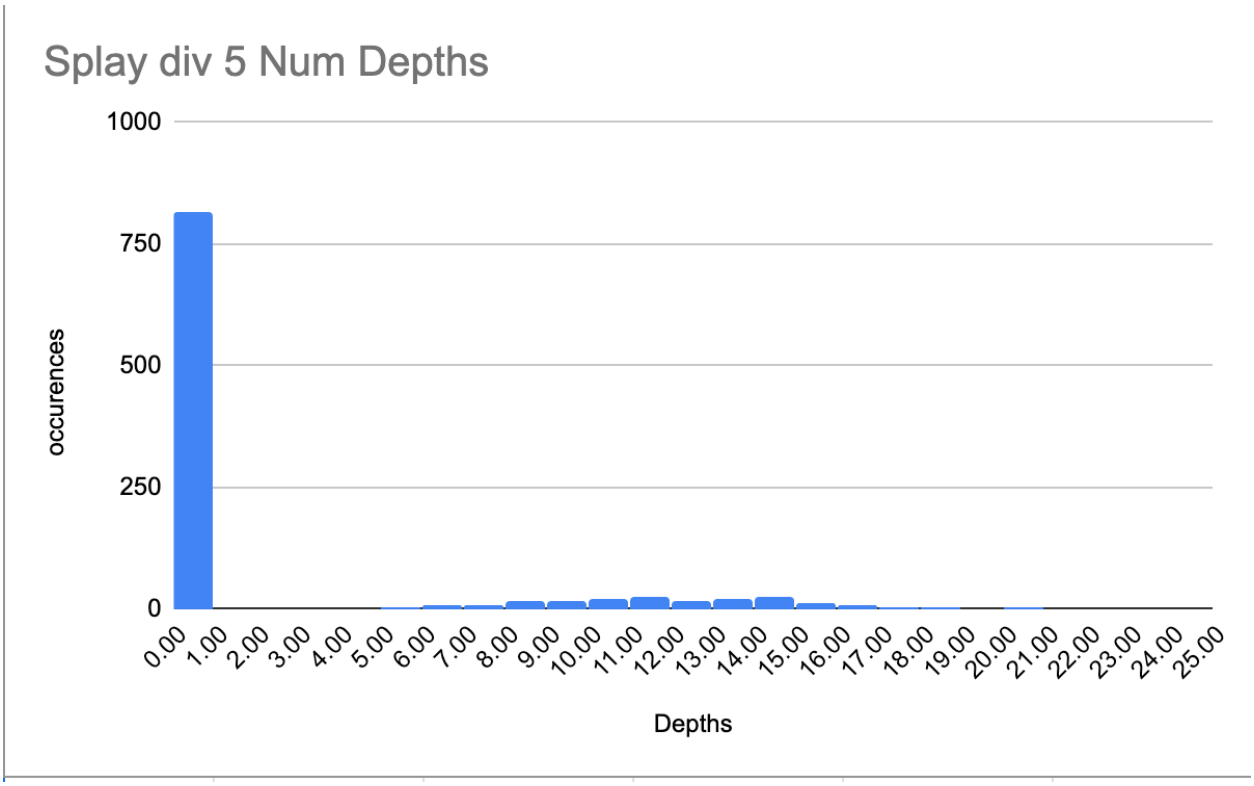
Once again, this tree appears how I expected it to. The tree is searching for the same value 5 times over and over again, which would return the same depth 5 times. Here the max depth value is 10.5 once again, that is more efficient than the binary search tree. This is mostly due to the self-balancing property AVL trees contain. These trees tend to be much wider than they are tall



These next three graphs are representative of the data received when searching through my splay tree. When the find method is called, the node is splayed, which then makes it the root node. This means that if the nodes are being searched for in the same order they were added to the Splay Tree, then the next node would be a direct child of the root node which was just splayed. This graph shows a maximum depth value of 24 and an average of about 12. This graph is rather symmetric which was certainly expected. The worst-case complexity for this tree is O(logN).



This graph depicts the results of the splay tree when the indexes of the vector were randomized. This graph differs from the previous, which was expected, because when dealing with splay trees, order does indeed matter. Here the occurrences have increased quite significantly, and the max depth has increases to 26. The order of which the nodes were inserted changed and the graph accurately represents that. The worst-case complexity remains O(logN).



This graph caught me off guard at first, but after thinking about it, it looks just the way it should. Each node is inserted five times for this particular test. The first time an index is searched for, it is found, and splayed to the root. This means, that for the next four times that index is searched for, the value will still be at the root, so it should return a depth of zero. This why there an overwhelming majority of depths occurring at 0. The worst-case time complexity will be much better than the other splay searches. We are only searching for 1/5 of elements previously searched for. We can visually see that this search is better than the first two done within this tree. The worst-case complexity is O(logN).

When searching for an object that is not in the tree, the find function will return zero, which corresponds to false. If you were to find the depth of these searches they would be at the halfway point in your tree. When searching for an object that was not in my Binary Search Tree, it returned a depth of 13. When searching for an object that was not in my AVL tree, the depth was 12. Lastly, when searching for an object that was not in my splay tree, it returned a depth of 13 as well. See lines 53-64 in main.cpp to see tests.