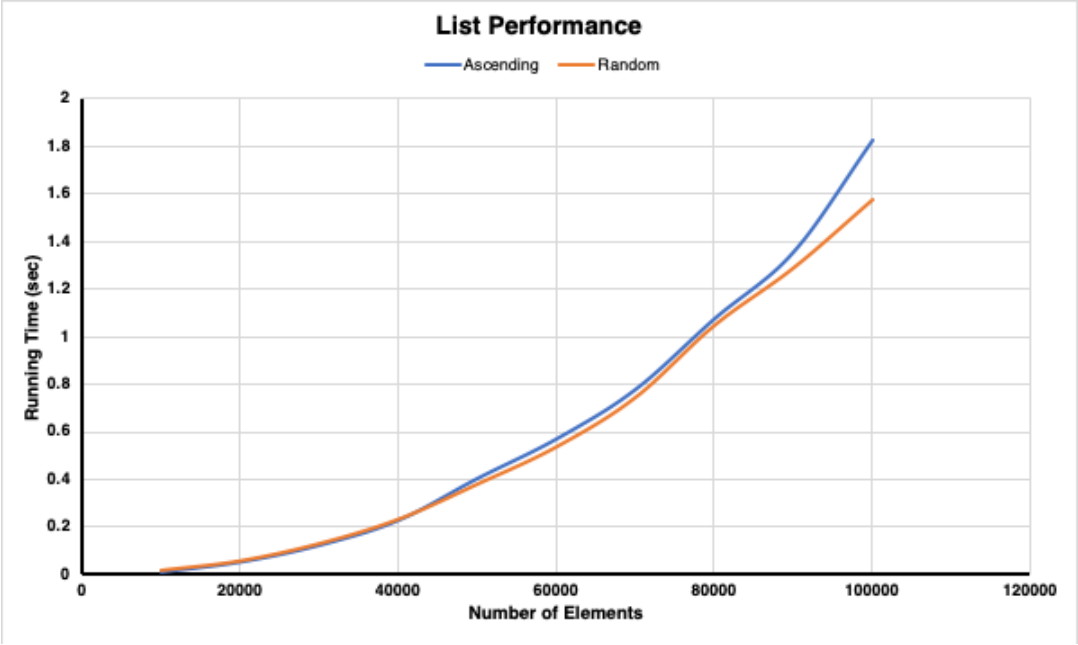
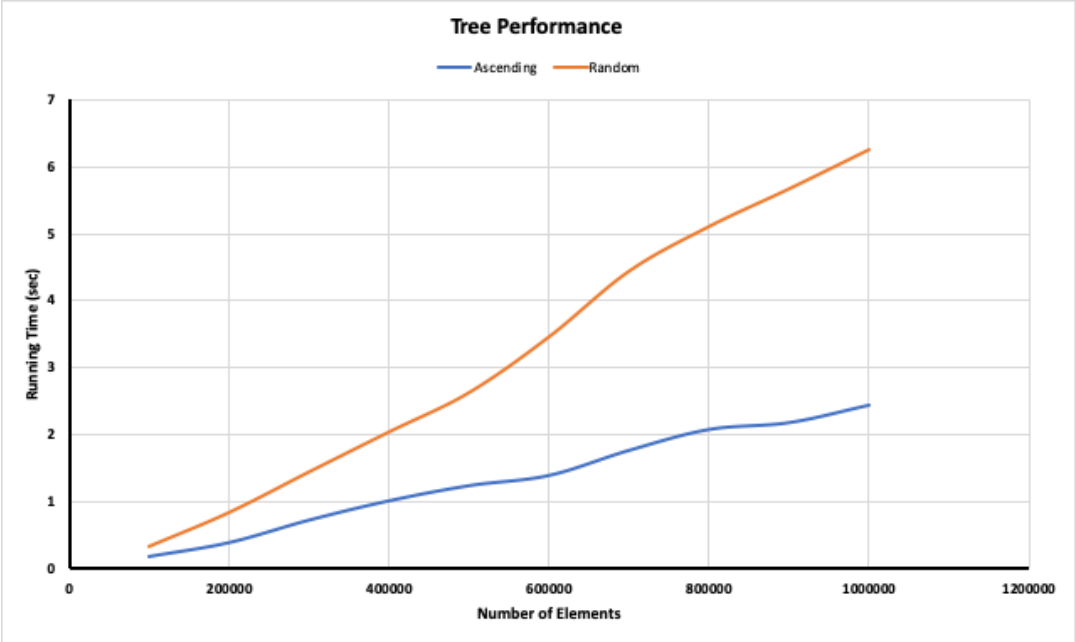


Inserting an element into a dictionarySet takes $O(n^2)$ time, shown by our quadratic graph. This is because every insert of n nodes perform O(n) operations. An O(n) operation means that it must, in a worst case scenario, traverse the entirety of a dictionarySet to perform its operation. In the case of insert, this is the List function remove_assoc. This is called before appending the new key value pair onto the dictionarySet in order to remove any preexisting pair with the given key. It traverses the whole association list to find if any key value pair uses the given key—that takes O(n). The next step is appending the new key value pair onto the potentially trimmed dictionarySet; we do this using the cons operator, which is O(1). The combination of this operations results in a runtime of O(n) for each insertion performed, of which there are n.

Ascending lists and random lists perform very similarly because both require the same amount of traversal down the list to remove any existing bindings and add the new key value pair.



For inserting a node into a tree, it takes O(n log n) time, as evidenced by our linearithmic graph (which is a linear graph that's curved ever so slightly upward). This means that, in the worst case scenario, inserting n nodes will take log n operations each time. Based on our recursive implementation of insert, this make sense. Either the new node value is the same as the head of the current subtree we're looking at, it's less than the head node's value, or it's greater than the head node's value. For the first case, that node can't be inserted, because the RI of a BST prevents duplicate values, so the original tree is returned. However, for the last two cases, that means that the left or right subtrees (respectively) need to be stepped through the see at which level of the subtree the node can be placed. Stepping through a subtree means traversing down either the path headed by the left child or right child of a node—the maximum number of steps this can take is defined by the height of the tree, which, in a tree of n nodes, is log n. Worst case scenario is that all n nodes have to bubble down each level of the tree to be added to the bottommost level, which would result in an overall timing of n log n.

Ascending (i.e. sorted) lists of nodes perform better than randomly generated ones because there is less of a need to balance the tree between each insert. Balancing the tree involves looking at each newly inserted node and changing the colors for each so that the RI is followed (no red node has red children and all branches have the same number of black nodes). A list of integers in ascending order will always have to insert the node at the bottommost level of the tree, but will not have to perform as many balance operations. On the other hand, a randomly composed list of integers will have to place nodes at different levels of the tree but will have to balance more often to prevent the ripple effect of the changing colors of surrounding nodes due to each insert.