Assignment 7

Question 1

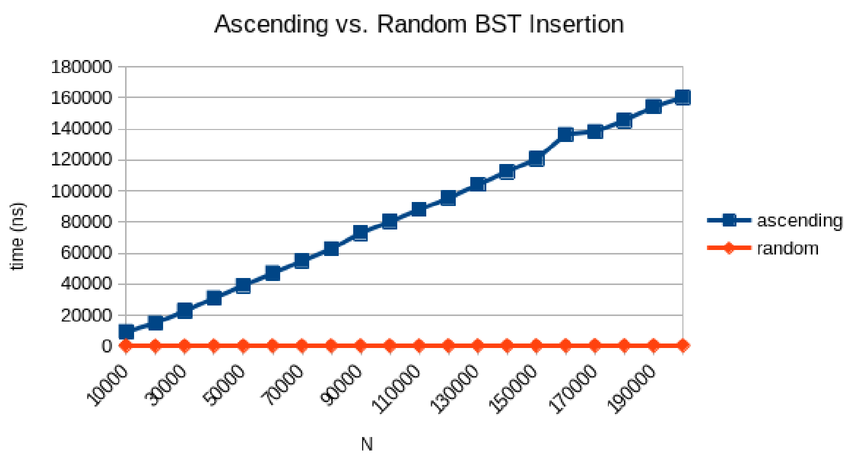
I (Chase Yates) and Liam Astin completed this assignment together. I submitted the files to Gradescope.

Question 2

If items are inserted into a BST(Binary Search Tree) in a sorted (ascending or descending) order, then the tree will entirely be one-sided, and lead, in general, to O(N) times rather than the ideal O(logN) times for all `add`, `contains` and `remove` methods.

Question 3

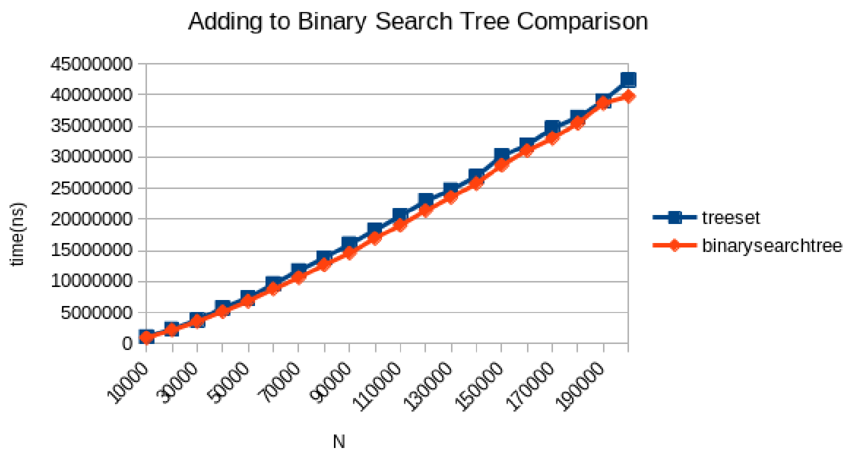
In this experiment I created two smaller experiments and compared their times. In the first experiment, I ran various tests with N sized from 10,000 all the way to 200,000 in steps of 10,000. I created sorted Integer ArrayLists of size n from 0 to n-1. I then used the BST `addAll` method to add my collection to the BST in order. I then ran a timing test with 10,000 loops to get an average time to run the BST `contains` method, checking contains on a random integer within the tree. The second experiment was exactly the same as the first, however I ran `Collections.shuffle` to randomize the order of the integer ArrayList before adding it to the BST. I then compared the times together and got this result:

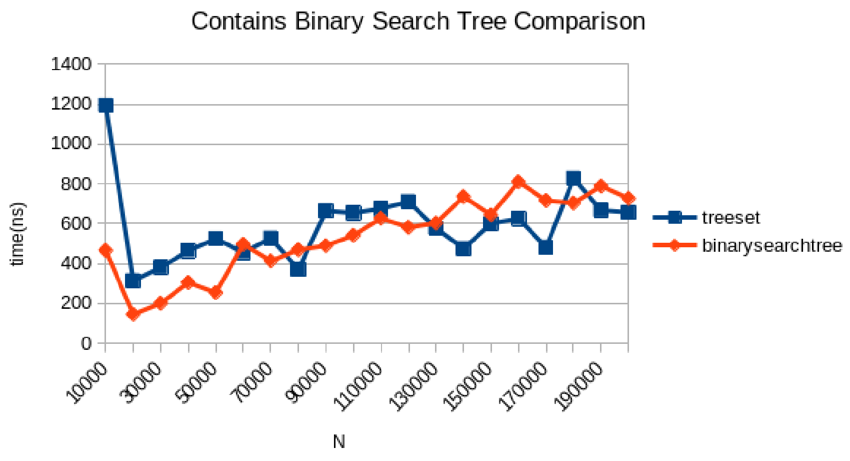


From these results, you can see the clear O(N) time on the ascending array BST for the `contains` method, and the random array BST being so small in comparison that it doesn't even register on the graph. Looking at the times for the random BST, you can infer and see times of O(logN) as the BST willbe much more balanced.

Question 4

For this experiment I ran 4 tests. The first test was to see the time it took to add a random list of integers of size N to a java `TreeSet` which implements tree balancing automatically. I took values of N from 10,000 to 200,000 in increments of 10,000, and repeated the timing tests 1,000 times. Secondly, I did this exact same thing using my `BinarySearchTree` instead of java's `TreeSet`, which does not implement auto-balancing. For the next to tests, I used the same numbers, timing how long it took to run the contains method on each of these data structures for random integers within the trees, and compared their times as well. Below are the results of these tests:





As you can see, adding a list of N integers to a tree is clearly O(N) time for both trees as shown by the first graph, however the leaner `BinarySearchTree` class barely eeks out the competition. I believe the natural balancing that random integers provides reduces the impact that the balancing `TreeSet` can produce.

The results for the second graph are a bit stranger and all over the place, however we can make some sense of it. For both the `TreeSet` and `BinarySearchTree` I think we can safely say that the graphs slowly increase over time, perhaps in O(logN) time. Which class is better at running the `contains` method seems to be changing back and forth over time, indicating that their performance is rather close to each other.

Question 5

BST's *can* be good data structures for dictionaries, as they can lead to fast, O(logN) query times. However, a sorted list using a traditional binary search would run just as fast, and would occupy a contiguous space in memory (or on disk). However, when it comes to insertion, Arrays or ArrayLists will find where the item needs to go in O(logN) time, but have to move all subsequent items over, leading to overall O(N) time. Insertion for something like a dictionary only happens once, however for all the elements, and most dictionaries are already sorted when they are created. Due to this, the creation speed for a new dictionary based on an existing one would likely be O(N) time for its entirety, not just one element. The same goes for deletion as deletion almost never happens for dictionaries. They are created and queried often, but how often do we really need to remove something? Not that often at all. Due to these factors, I think something like a sorted ArrayList would likely be your best bet for Dictionaries, but BST's aren't far off.

Question 6

If you construct a dictionary using a BST, and your dictionary is in alphabetical order, then your BST will be one sided. To fix this issue, two options come to mind. The first is to randomize the order of the source dictionary so that your BST will be fairly balanced just by chance. The other, and perhaps better option is to use something like an AVL tree, which balances itself automatically and precisely.