Assignment 6 Analysis

Aiden Pratt

1. Explain how the order that items are inserted into a BST affects the construction of the tree, and how this construction affects the running time of subsequent calls to the add, contains, and remove methods.

   The order that items are inserted into a BST is very important for the construction of the tree and the run times of the add, contains, and remove methods. When elements are inserted in a sorted order, the tree has a linear structure and will be left or right skewed. This results in the tree's height approaching N (the number of elements in the tree). This is the worst case for these methods as they each essentially will need to perform a linear search type algorithm to complete.

   Inserting the elements in a random order can lead to a more balanced tree on average that likely will not be heavily skewed left or right. In this case, the height of the tree is closer to log N, so the time complexity of these methods is O(log N).

2. Design and conduct an experiment to illustrate the effect of building an N-item BST by inserting the N items in sorted order versus inserting the N items in a random order. Carefully describe your experiment, so that anyone reading this document could replicate your results.

   In my experiment, I time how long it takes to call the contains method on large BST's in which the elements are inserted in sorted order and random order.

   I first utilized a for loop which iterates from N(# of elements) = 100 to 20,000, incrementing by 2000 each time. In this loop, I create a new BST for a sorted test and a new BST for a random test. This creates a total of 10 BST's of each type.

SORTED TEST:

I begin by looping through and adding numerically sorted elements from 1 up to N into the BST. I then begin to time how long it takes for the contains method to run on the N sorted BST with System.nanoTime();. This provides me with the time in nanoseconds it takes for the contains method to be run on each BST of a different size N. This results in a time complexity of N^2 because contains is being run on N elements.

RANDOM TEST:

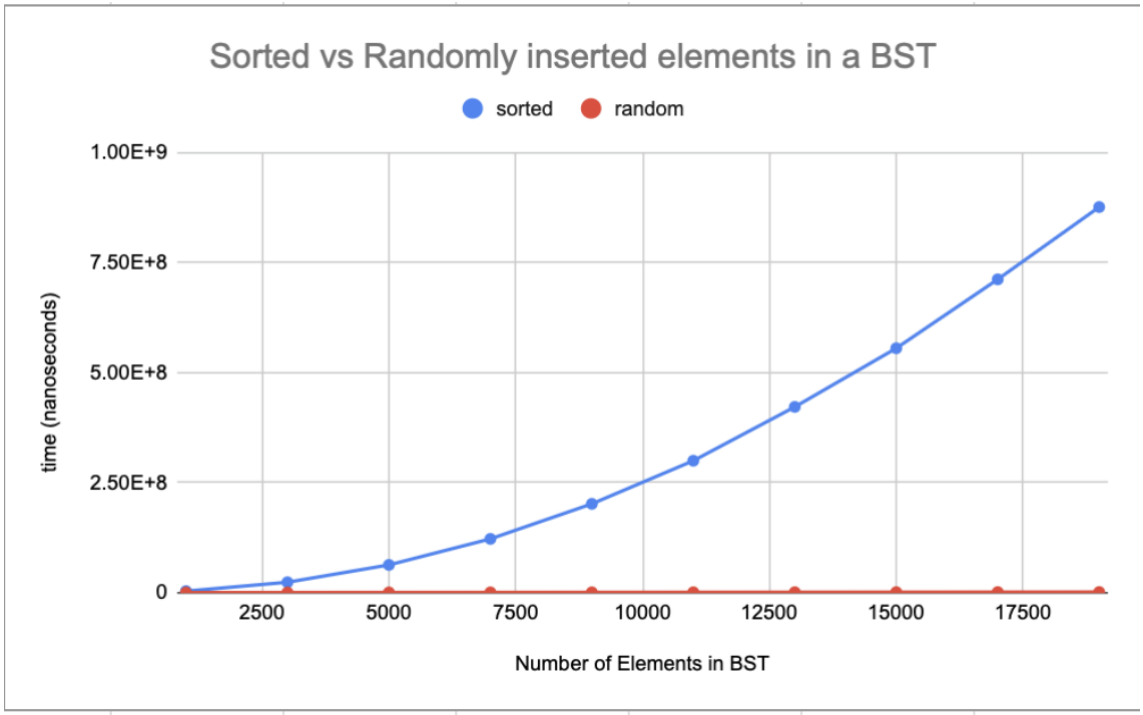
This test is designed to run 100 times per N size to determine an average of the randomly entered elements and the contains method. A random order of numbers is generated and added into a new BST set to contain random elements. Similar to the sorted test, System.nanoTime(); is used to time the contains method on each size N of the randomly inserted elements in the BST. The average of the 100 runs is then found. This will result in a time complexity of O(N log N) because contains is being called N times on each element to find the element.

3. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plots(s), as well as your interpretation of the plots, are critical.

This plot of my data aligns well with what I anticipated for the sorted vs randomly inserted elements. The sorted list takes much more time to find the contained element than the random list. This is because the sorted search is O(N^2) and the random search is O(N log N) because I am calling the contains method on every element in the BST. The random list still increases in time, but at a much less significant rate.

Data for inserting a sorted list vs a random list into a BST

N	sorted	random
1000	3298291	64983
3000	23418958	162413
5000	62776458	288382
7000	121905250	401205
9000	201510375	531668
11000	299489708	647542
13000	421768125	781926
15000	555033416	911450
17000	711508375	1051698
19000	875726500	1184474



4. Design and conduct an experiment to illustrate the differing performance in a BST with a balance requirement and a BST that is allowed to be unbalanced. Use Java's TreeSet as an example of the former and your BinarySearchTree as an example of the latter. Carefully describe your experiment, so that anyone reading this document could replicate your results. Plot the results of your experiment. Since the organization of your plot(s) is not specified here, the labels and titles of your plots(s), as well as your interpretation of the plots, are critical.

My experiment first generates a list of numbers in a random order. That same list is added to the TreeSet and to the BST set. The time for the program to loop through each item in the list of size N and added to TreeSet and BST is timed and recorded for the data plotting. This experiment is run 100 times and the average run time of each size of N is calculated and used for the data analysis. Then, the program loops through the items in the randomly ordered list and times how long it takes for the TreeSet's contain method vs the BST's contain method to find the element within the set. This experiment is also run 100 times and the average run time of each size of N is found and used for the data analysis.

Data for add method testing with a random list:

N	TreeSet Contains	BST Contains
10000	1157841	3317427
20000	2359648	13857595
30000	4000178	33180859
40000	5662072	63379120
50000	7296258	101458711
60000	8798916	146608308
70000	10401408	198191937
80000	12880968	269713430
90000	14499608	347179918
100000	15805017	428080392

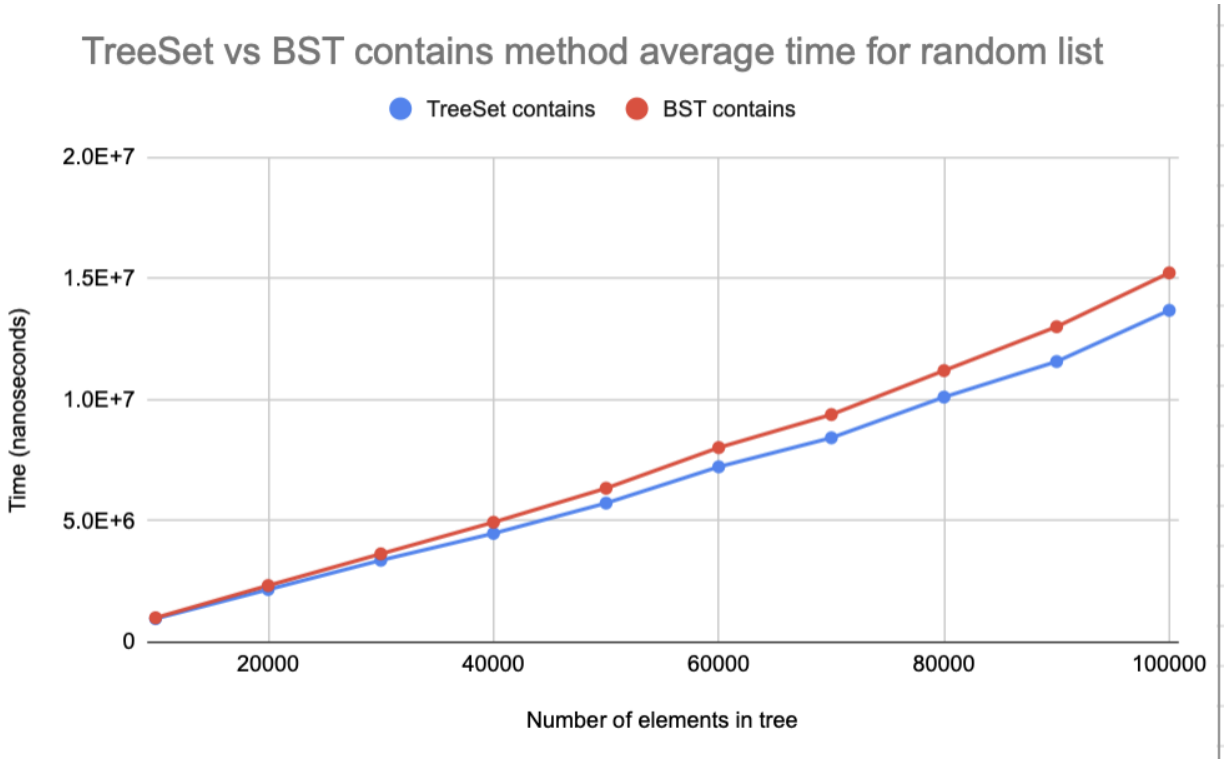
This experiment displays the data from the add method in the TreeSet vs the add method in the BST. The BST is shown to take significantly longer as the size(N) of the elements being added grows. This discrepancy is primarily due to the optimization differences between my custom BST implementation and the Java TreeSet.

This could also be the case because the BST is not self balancing, and as more elements are added, the tree can become less balanced and have a less efficient addition.

The difference in these performances reflect in their respective time complexities. The TreeSet and BST expect to display an average time complexity of O(log N) for insertion in a balanced scenario. However, it seems in this case my BST is likely toward the worst case which is O(N) due to an imbalance.

Data for contains method testing on a randomly inserted list:

N	TreeSet contains	BST contains
10000	960499	997685
20000	2162366	2328012
30000	3369772	3633814
40000	4477017	4936856
50000	5726690	6340357
60000	7224684	8017009
70000	8421316	9379376
80000	10099605	11192027
90000	11565766	13002222
100000	13677337	15223372



In this experiment, data was randomly inserted into both a TreeSet and a custom Binary Search Tree (BST), likely resulting in a balanced BST due to the random insertion order. The TreeSet, being a specialized Java data structure optimized for balanced trees, demonstrates slightly higher efficiency compared to the custom BST.

The contains method is expected to run in O(log N) time on average in a well-balanced tree. However, due to the need to traverse the entire tree in this experiment, the overall time complexity of the contains method in this experiment approaches O(N log N) for both the TreeSet and BST. This complexity arises from the necessity to loop through the entire tree N times to check for the element.

5. Discuss whether a BST is a good data structure for representing a dictionary. If you think that it is, explain why. If you think that it is not, discuss other data structure(s) that you think would be better. (Keep in mind that for a typical dictionary, insertions and deletions of words are infrequent compared to word searches.)

There are a few reasons for why a BST could make for a good data structure that represents a dictionary. A BST allows for efficient searching due to how it is sorted. Finding a word in a dictionary by looking it up could thus be fast due to the time complexity of O(log N). BSTs also sometimes require less memory overhead compared to other data structures.

However, if frequent insertions and removals occur in the dictionary BST, it could lead to an unbalanced tree which could harm the faster search performance and result in a worst case O(N) complexity. Another alternative structure that could be useful are AVL trees or hash tables. AVL trees are good data structures for ensuring balance. Hash tables can offer much more efficient insertion and removal. These traits will be greatly important for a dictionary data structure. These could lead to more consistent performances if there are going to be frequent modifications.

The BST also may not always maintain a balance depending on how the words are stored within the BST. If many words are short and only a few letters, it may lead

to an unbalanced tree. Depending on the size of the dictionary and the types of words being stored, the resulting tree may be unbalanced if the words are inserted or organized in a poor performing way.

6. Many dictionaries are in alphabetical order. What problem will it create for a dictionary BST if it is constructed by inserting words in alphabetical order? What can you do to fix the problem?

The problem with creating a dictionary with words in alphabetical order is that the list will become heavily left or right skewed and the search method will become effectively like a linear list. This is a time complexity of O(N).

This could be fixed if there are self balancing mechanisms that occur when an item is added to the BST. We could also randomize the order in which items are inserted into the BST, so that there is a better chance of a more efficient BST structure.