Analysis

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 affects the construction of the tree in such that in a BST all operations may require traversing from top to bottom of tree; such that the time complexity is O (height). The order that items are added changes the balance of the tree and its height hence affecting its runtime for add, contains and remove methods.

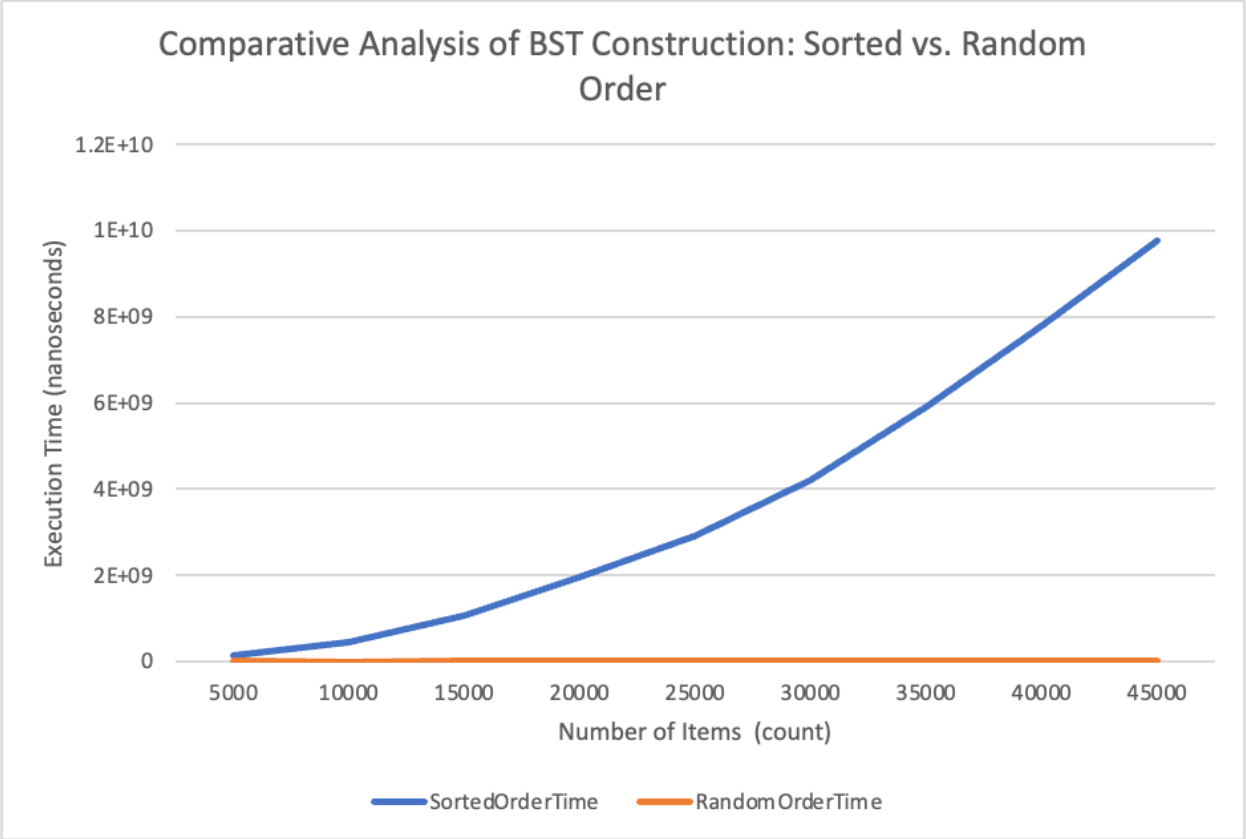
On the other hand, in a skewed tree; the running time is affected such that the add, contains and remove methods; in the worst case; would have a O (n) time complexity (where n is the number of nodes in the tree).

Furthermore, in a balanced tree, the running time of its operations is more efficient where its height has a logarithmic relationship with the number of nodes in the tree which leads to time complexity of O (log (n)) for its methods. A balanced tree structure allows for a more efficient computation of its insertion, add and remove methods.

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.**

The experiment that has been conducted is based on comparing insertions into the data structure and its run time when adding sorted elements versus unsorted elements into the data structure. The size, the number of elements added into the data structure, is varied by the number of elements (N) from 5,000 and increments of 5000(5k, 10k, 15k and so on) up to 45000 and the operations evaluate the time taken for these operations. In the sorted case; the elements are added to the BinarySearchTree from 1 to N and the experiment measures the time taken to have its method complete. On the case of the random insertion, the case generates a random number ranging from 1 to 1000 to be added into the data structure and the program computes the time that it took to run it’s completion from size 1 up to the target size. The output of the experiment displays and writes a file that contains the size of each iteration and its run time; then another python program reads from the data file and displays a graph showing visually the difference between the two behaviors.

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.**

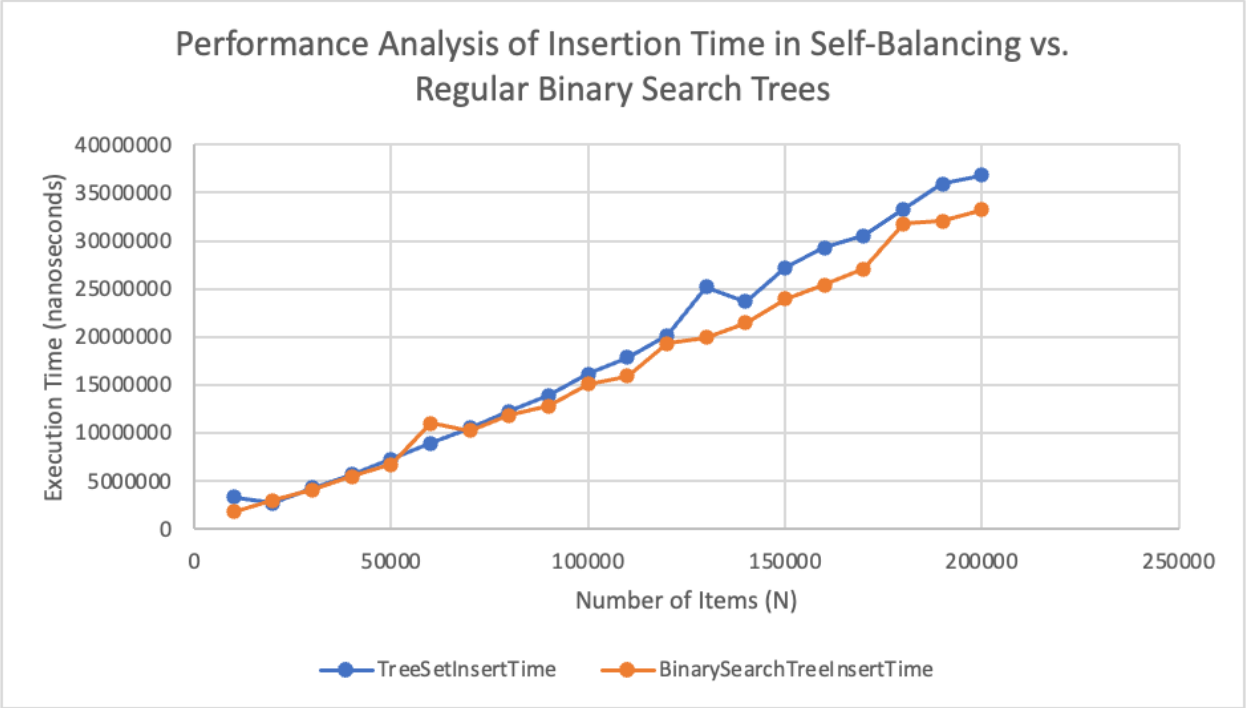


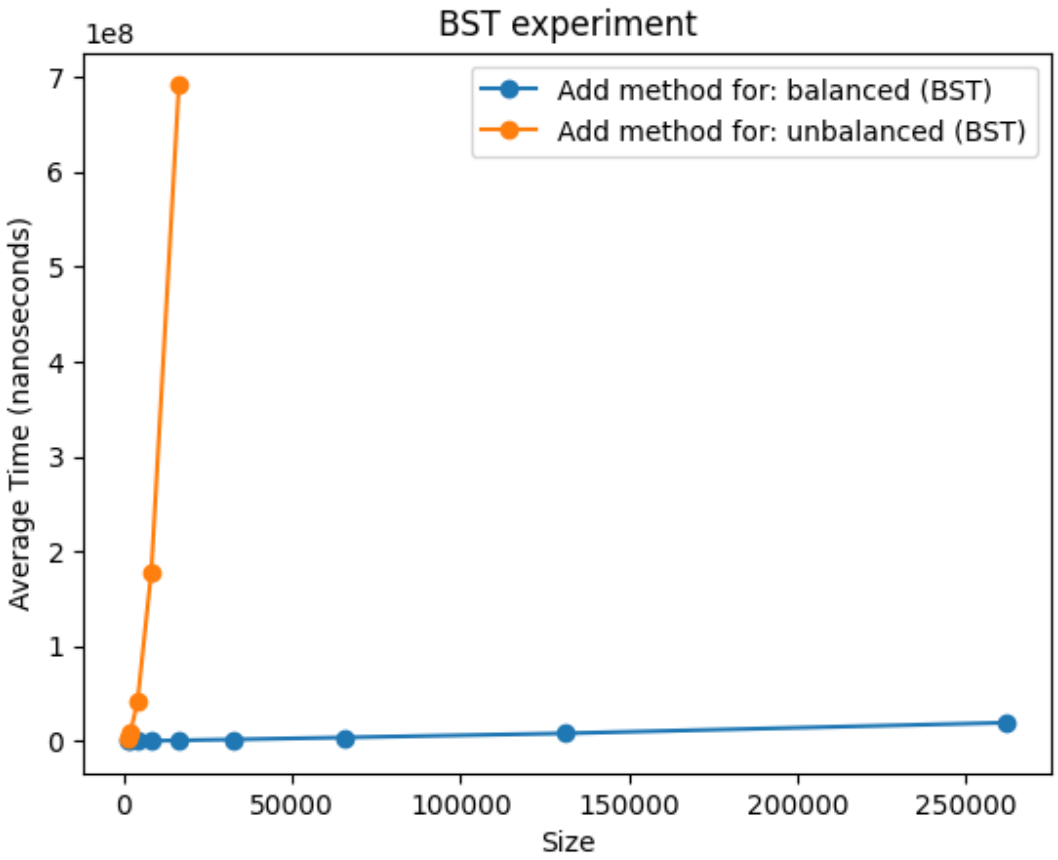
When comparing the sorted versus random order when adding elements into the data structure; the BinarySearchTree(BST) have a longer run time when compared to the unsorted BST. One of the biggest reasons on why that happens is because when adding elements from smaller to larger in a sorted faction, the new nodes in the tree are all inserted in it's right which leads to a heavily right skewed tree which leads to a worst case for a BST and the most unbalanced sort of binary tree. In this case the time complexity of sorted BST is O (n).

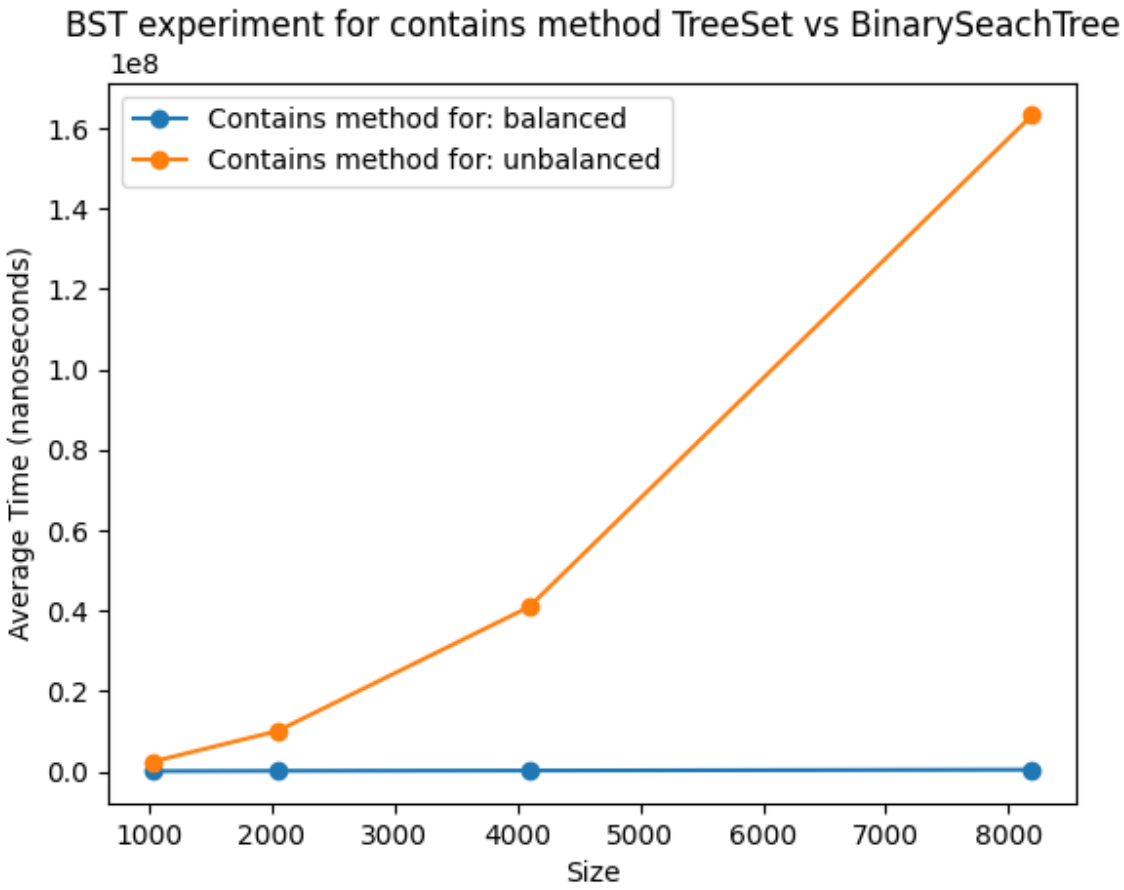
On the other hand, for the random insertion the items are added in a random order; the resulting tree tends to be more balanced which leads to the depth of the tree to have a logarithmic relationship with the number of nodes in the data structure which leads to a time complexity of O (log n).

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.




BST experiment
1e8
Add method for: balanced (BST)
Add method for: unbalanced (BST)
Average Time (nanoseconds)
Size
0
1
2
3
4
5
6
7
0
50000
100000
150000
200000
250000



The experiment is represented by the having the add method for a balanced BST (using a *treeSet*) versus a not self balancing BST (**BinarySearchTree**) data structure by running the worst case for both methods where the inserted data to the data set is already sorted. The run time differences illustrated in the graph indicates that the class that has been written, where it does not self-balance it's height have a time complexity of **O (n)** whereas the time complexity for a self-balancing binary tree is *O (log n)*.

Instructions in doing the experiment to replicate my results; use i = 1; i < size; i++ → bst.add(i); before the loop starts is where you time it and immediately after loop stop measuring the time (using System.nanoTime()). Use size variable inside of inner loop from exp 10 - 20; and size to be int of pow(10, 2) to reach powers of 2 from 10 to 20.

For the contains method, the time complexity for the treeSet was still O (log n) while the BST exhibited a O (n) time complexity but that was expected since it was used the worst case for the BSN (using a sorted set case). When the experiments were run comparing both data sets using average case; the output was very similar for both without much noticeable difference which inferred that they both keep the same time complexity of O (log n).

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.)**

A BST can possibly be a good data structure that represents a dictionary specially if the BST is a self-balancing tree like AVL or Red-Black tree. These data sets usually compute its operations in O (log n) time complexity which means if you are looking for a word in the dictionary it is pretty fast to do so. An alternative for a data structure that can be really useful for representing a dictionary would be Hash Table due to always taking input “key” and producing fixed-sized ch string which is very useful for using a key associated with a word when looking words or finding words in the dictionary. However using Hash Table comes with the trade off that it requires more memory overhead when compared to using a BST. Also a Hash Table does not preserve the order of entries; though its time complexity is O(1) (faster than log n).

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.**

If a BST is constructed by inserting words in alphabetical order; then it’s time complexity for adding, removing, looking into words would increase by the difference of log(n) to O(n) instead; which has represented by the previous graphs here if the BST is not self balancing like a AVL or Red-Black tree. The best solution can be arguable to be doing rotations when necessary. AVL’s have a balance factor to maintain each node the difference of height between left and right subtrees of no more than -1 or 1. Whenever a new insertion is made that tosses that balance off; then rotations happen. Rotations can be a bit lengthy to explain but it basically either rotates left or right whereas rotating right you use when the right subtree is too deep and the same logic for a left rotation; rotations can also be single or double depending on the requirement to maintain the balance of the tree.