Assignment 06 Analysis Doc

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.**

   In a BST all operations may require traversing from the top to the bottom of the tree, so all operations are O(tree height)--meaning the height of the tree has a direct effect on runtime.

   The order that items are added changes the balancing of the tree (height), thus affecting the running time of the add, contains, and remove methods.

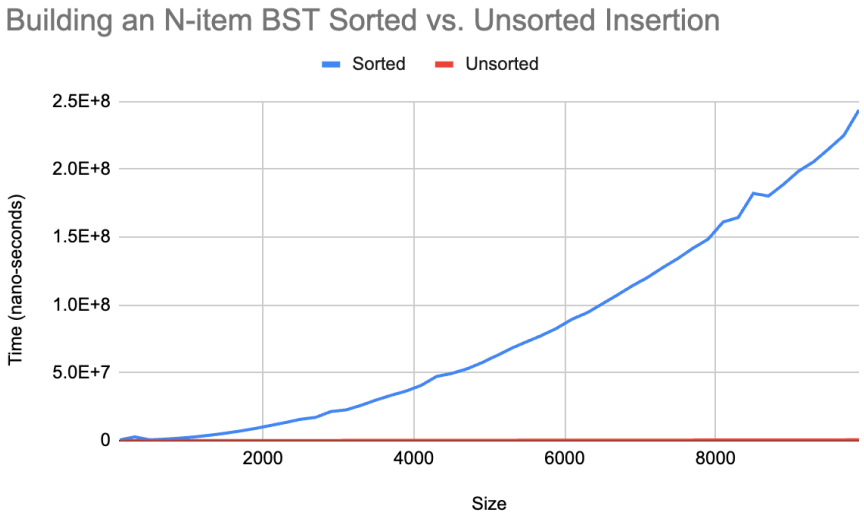
   In a skewed tree, the running time of these operations becomes inefficient, the add and contain methods would have a worst-case time complexity of O(n), where n is the number of nodes in the tree. This is because each operation would require traversing through all the nodes in the tree.

   In contrast, in a balanced tree, the running time of these operations is more efficient. In a balanced tree, the height of the tree is logarithmic in the number of nodes, resulting in a time complexity of O(log n) for the add, contains, and remove methods. This is because the tree's balanced structure allows for a more efficient search, insertion, and removal of nodes.

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.**

   This experiment delves into the performance comparison between sorted and randomly ordered insertions and subsequent contains checks within my custom BinarySearchTree. The size is varied by the number of elements (N) from 100 to 10000, incrementing by 200, and evaluates the time taken for these operations. For the sorted insertion, it adds elements incrementally from 1 to N into the BinarySearchTree and measures the time taken for contains checks. On the other hand, for random insertion, it generates a randomly shuffled order of integers from 1 to N, adds these elements into the tree, and records the average time taken for contains checks across multiple runs (100 runs). The output displays the time taken for each operation with sorted insertions and the average time taken for random insertions across the varying sizes of N.

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.**



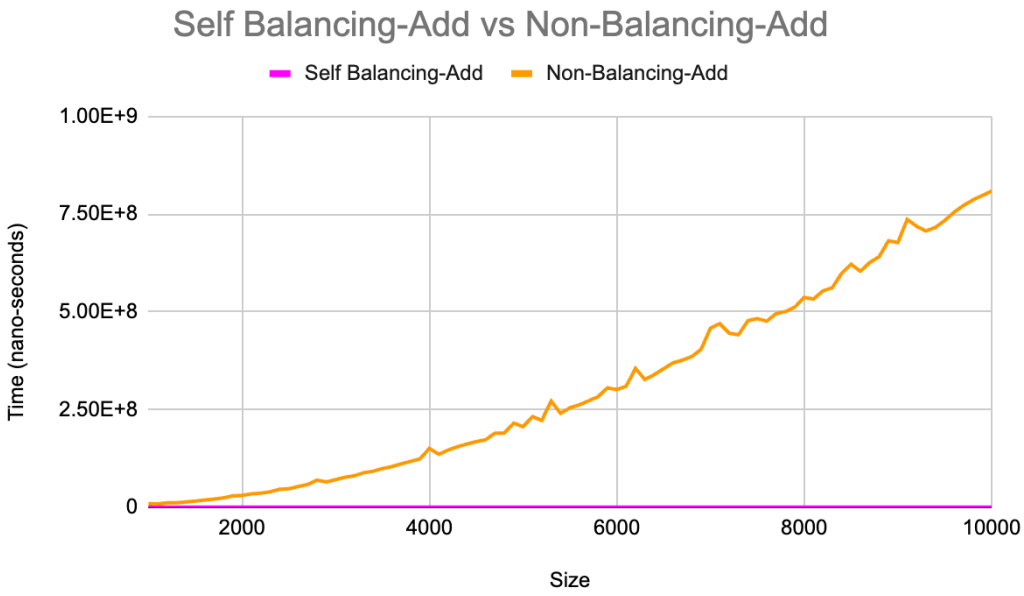
Looking at this data, the sorted BST has a longer run time than the unsorted BST. This is because when the sorted tree is being constructed (calls to insertion) it is adding the only to the right, creating a heavily right skewed tree. This is the worst case for BST and is the most unbalanced tree possible. Thus the runtime for a sorted BST is O(N) because each operation requires traversing through all the nodes in the tree.

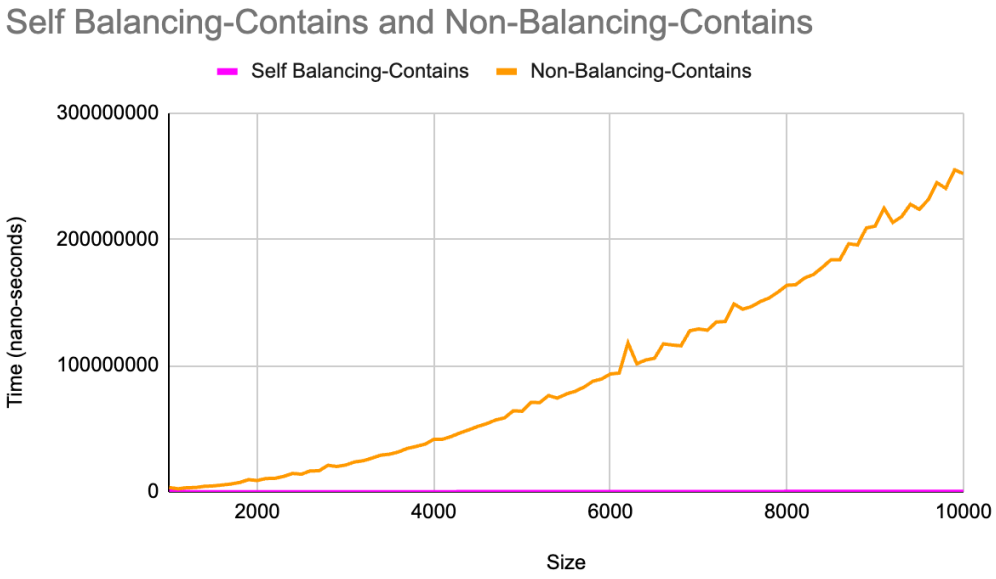
The unsorted BST is more likely to be balanced and a balanced tree the height of the tree is logarithmic in the number of nodes, resulting in a time complexity of O(log n). Shorter height means less traversing through all the nodes in the tree.

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.**

   This experiment evaluates the performance disparity between a TreeSet (a self-balancing binary search tree) and my custom BinarySearchTree (non-self-balancing). The experiment varies the number of sorted elements (N) from 1000 to 10000 in steps of 100 and measures the time taken for insertion and contains checks. For each N value, it generates a sorted ArrayList and

records the execution times for adding elements and checking contains in both tree structures.





The results, presented in graphical format, demonstrate how the Java TreeSet (self-balancing) consistently maintains efficient insertion and contains checks regardless of the number of sorted elements, while the BinaryTreeSet (non-balancing) exhibits varying performance based on the data size due to its lack of self-balancing capability. The non-balancing BST has a much slower runtime compared to self-balancing BST.

In Java TreeSet, insertion involves additional rotations and rebalancing to maintain the tree's balance, ensuring a height of O(log N). This guarantees that the depth of the tree remains logarithmic with respect to the number of nodes, resulting in an average insertion time complexity of O(log N).

In my BinaryTreeSet the runtime is O(N). This is because the elements are inserted in sorted order, causing the tree to degenerate into a linear structure, which is the worst-case scenario because the tree's depth becomes equal to the number of nodes (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.)**

   BST serves as a decent structure for a dictionary, particularly for frequent searches (O(log N)), which makes it efficient for finding words and definitions based on keys. But BST's are sensitive to the input order during insertion, which can lead to skewed trees with poor runtime (O(N)).Self-balancing BST could mitigate BST's shortcomings as they address the balance problem but while they offer better insertion and deletion performance at the cost of slightly slower searches.

   Alternatively, Hash Tables might be preferable due to their consistent performance regardless of input order and faster average-case time complexities for insertions, deletions, and lookups O(1). A trade off to a HT is they might not maintain a sorted order unless additional data structures are used in conjunction.

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?**

When words are inserted in alphabetical order into a BST, each subsequent word is placed to the right of the previous word. This results in a tree that resembles a linked list, with nodes arranged in a single linear path, causing the tree to lose its advantage of logarithmic search time complexity.
One approach to prevent the formation of skewed trees in a dictionary BST is to tweak the BST to be a self-Balancing Tree. Using self-balancing BSTs like AVL or Red-Black

trees guarantees balanced structures despite the input order. These trees autonomously manage balance during insertions and deletions, ensuring consistent logarithmic time complexity for search operations.