Assignment 6 analysis document:

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.

Best Case: Balanced Tree. A balanced BST has the smallest possible height for a given number of nodes. The height of a balanced BST is proportional to log(n).

Worst Case: Unbalanced Tree. If items are inserted into the BST in sorted order (either ascending or descending), the tree will become unbalanced, resembling a linked list. In this case, the tree's height will be n

The shape of the BST, which is determined by the order of insertions, greatly impacts the running time of operations.

add Method:

> Balanced Tree: Inserting an item will take O(log n) time since you can traverse the tree from root to leaf, visiting at most log(n) nodes.
>
> Unbalanced Tree: Inserting an item will take O(n) time, because in the worst case, you need to traverse all n nodes.

contains Method:

> Balanced Tree: The search will take O(log n) time, as you traverse the tree at most log(n) levels.
>
> Unbalanced Tree: The search will take O(n) time, because you might need to check every node if the tree is unbalanced.

remove Method:

> Balanced Tree: Removal requires finding the node (which takes O(log n)) and possibly rearranging the tree, which is done in O(log n) time.
>
> Unbalanced Tree: Removal takes O(n) time in the worst case, because you may need to traverse the entire height of the tree, which is n for a degenerate tree.

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 aims to measure and compare the performance of a Binary Search Tree (BST) when built with elements inserted in **sorted order** versus **random order**. Specifically, the experiment records the time required to invoke the contains method for each item in the tree after all items have been inserted.

Experimental Setup:

Problem Sizes: The experiment is conducted for problem sizes ranging from 10,000 to 200,000 elements, in increments of 10,000.

Data Structure: A custom implementation of a Binary Search Tree (BinarySearchTree<T>) is used, which supports insertion (add), searching (contains), and retrieval of elements in sorted order.

Test Case 1: Inserting elements in sorted order (e.g., 1, 2, 3,..., N).

Test Case 2: Inserting the same elements in random order using the Collections.shuffle method to randomize the order.

Measurement: For each test case (sorted and random order), we measure the time taken to check for the presence of all N elements using the contains method. The time is recorded in nanoseconds.

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.

Based on the experiment results, I observe the following trends:

Sorted Order BST:

> When you insert elements in sorted order, the BST becomes unbalanced. This results in a skewed tree(resembling a linked list) where the height of the tree is equal to N. This makes the contains operation take linear time (O(N)) for each lookup.
>
> As N increases, the time for each contains operation increases significantly, leading to quadratic growth in total time.

Random Order BST:

> When elements are inserted in random order, the BST remains more balanced on average, resulting in a logarithmic height (approximately log2(N)). The time for the contains operation will therefore be logarithmic (O(log N)) for each lookup.
>
> As N increases, the time for each contains operation grows much more slowly compared to the sorted order case. The overall time will increase, but at a much slower rate compared to the sorted order.

Expected Graph Behavior:

The Sorted Order line should show an approximately quadratic growth in time with respect to N.

The Random Order line should show a much slower logarithmic growth in time, with the time per operation increasing slowly even for large values of 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.

Create and populate the TreeSet and BinarySearchTree with random items.

> Generate a collection of N random numbers (e.g., from 1 to N).
>
> Insert these numbers into both a TreeSet and a BinarySearchTree.

Measure the time to insert items into each tree.

> Use System.nanoTime() before and after the insertion process to measure the time taken for insertion in both trees.

Measure the time to invoke the contains method on each tree.

> For each tree, iterate over the same N random numbers and invoke contains on each one.

Again, use System.nanoTime() to measure the time for the contains operations.

Repeat the experiment for varying values of N (10,000, 20,000, ..., 200,000) to observe how the performance scales with tree size.

Plot the results:

Plot the insertion times for both the TreeSet and BinarySearchTree on one graph.

Plot the contains operation times for both the TreeSet and BinarySearchTree on another graph.

TreeSet Insert Time Analysis:

TreeSet is implemented as a self-balancing binary search tree.

Insertion into a self-balancing Tree takes O(log N) time because the tree remains balanced after every insertion. So as N increases, the time to insert grows logarithmically.

From the table, we can observe that the insertion times for TreeSet seem to be increasing logarithmically, as the difference between consecutive insert times gradually increases at a slower rate.

BST Insert Time Analysis:

Your BinarySearchTree implementation is unbalanced. In the worst case, if the items are inserted in sorted order (or nearly sorted), the tree will degenerate into a linked list, and the insertion time will become O(N).

Even if the items are inserted randomly, if the tree is not perfectly balanced, the insertion time will still approach O(log N) at best, but it could degrade towards O(N) if the tree gets unbalanced (due to uneven distributions of inserted nodes).

From the results, we can see that the insert time for the unbalanced BinarySearchTree increases faster than for the TreeSet. Specifically, at higher values of N (e.g., N = 100,000 and N = 200,000), the BST Insert Time becomes larger than the TreeSet Insert Time, suggesting that the tree may not be maintaining optimal balance as N grows.

TreeSet Contains Time Analysis:

The contains operation in a self-balancing tree also takes O(log N) time because of the tree's self-balancing property.

As N increases, the TreeSet Contains Time grows as expected for logarithmic time complexity, and it scales well, but it increases at a slower rate than the BinarySearchTree due to its balance.

In this experiment, we observe that as N increases, the TreeSet Contains Time grows slowly but steadily in line with the logarithmic growth expected for O(log N) operations.

BST Contains Time Analysis:

The BinarySearchTree performs O(N) in the worst case if the tree is degenerated into a linked list, which can happen when the tree is unbalanced. Even with random insertions, there could still be unbalanced cases, making the search time closer to O(N) instead of O(log N).

As we can see from the results, the BST Contains Time is roughly similar to the TreeSet Contains Time for small N (like 10,000 and 20,000), but as N grows, the BST Contains Time grows more quickly. For larger values of N (like N = 100,000 and 200,000), the BST Contains Time shows signs of becoming closer to O(N) rather than 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.)

Insertions and deletions in a typical dictionary are infrequent, while searches are more common. A balanced BST is a good choice.

In a balanced BST, the time complexity for search operations is O(log N), where N is the number of elements (words) in the tree. This means that for looking up a word in the dictionary, the search operation is quite fast, especially when compared to linear search in an unsorted list or array, which would take O(N).

However, this assumes that the tree is balanced, which is crucial. If the BST becomes unbalanced, the search time can degrade to O(N), as the tree can become skewed, degenerating into a linked list. This would make searching inefficient.

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 inserting words in alphabetical order, each new word will be lexicographically greater than all the previous words. As a result, each new word will always be inserted as the right child of the last inserted word. The BST will degenerate into a linked list.

The best solution is to use a self-balancing BST, which ensures that the tree remains balanced during insertions and deletions. These trees automatically perform rotations to maintain balance and avoid degenerating into a linked list.