Analysis document: BinarySearchTree

● Graded

Student

HARRY KIM

Total Points

31 / 35 pts

Question 1

Question 1

1 / 1 pt

✓ **– 0 pts** Correct

Question 2

Question 2

4 / 5 pts

✓ **– 1 pt** Did not correctly state that the `add()`, `contains()`, `remove()` methods are $O(N)$ in a left- or right-heavy tree

Question 3

Question 3

8 / 10 pts

✓ **– 1 pt** X- or Y-axis is unlabeled, missing units/ description of problem size, or otherwise incorrect

✓ **– 1 pt** Plot is difficult to interpret without reading description

Question 4

Question 4

9 / 10 pts

✓ **– 1 pt** X- or Y-axis is unlabeled, missing units/ description of problem size, or otherwise incorrect

Question 5

Question 5

4 / 4 pts

✓ **– 0 pts** Correct

Question 6

Question 6

5 / 5 pts

✓ **– 0 pts** Correct

Question 7

Late Penalty

0 / 0 pts

✓ **– 0 pts** submitted before deadline

Harry Kim
U1226472
10/21/2020
CS 2420

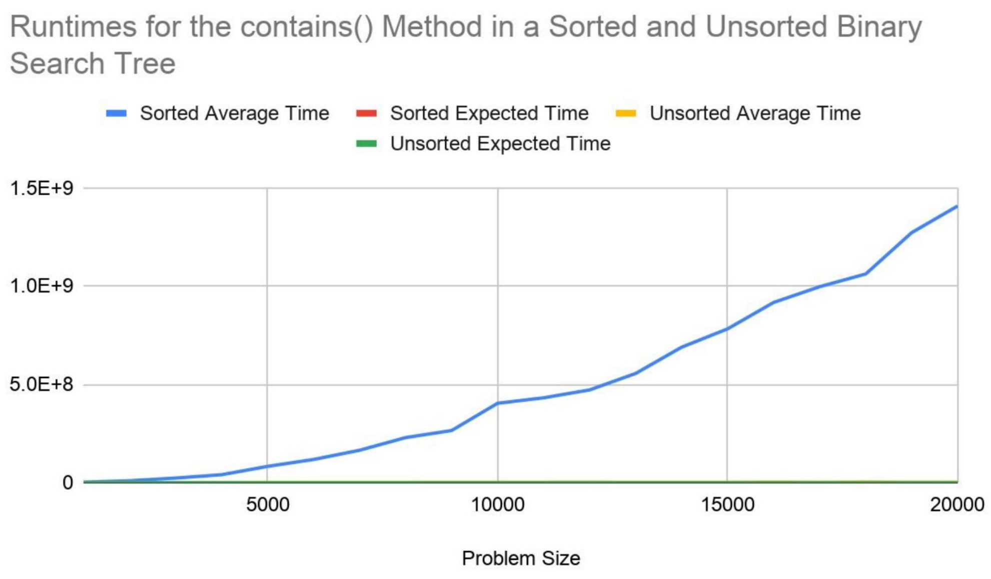
Analysis Document — Assignment 6

1. I did complete the programming portion of this assignment with a partner. My programming partner is Braden Morfin, and I am the one who submitted the assignment to Gradescope.

2. The order that items are inserted into a BST affects the construction of the tree because the first item inserted is always the root node and subsequent nodes added are either placed to the left or to the right of the root depending on the size of the inserted node. This means that the tree is not always guaranteed to be balanced. The construction affects the running time of subsequent calls to the add, contains, and remove methods as our BST is not guaranteed to be balanced, and so we can assume that the complexity of the add, contains, and remove methods will be the height of the tree. If those same methods were to be used on a balanced BST, the running times would display a logarithmic behavior. This is because when we do comparisons on a balanced tree, half of the tree will be discarded every single comparison, meaning the complexity will be O(logN) even in its worst case, which is hardly the case for an unbalanced BST with the number of nodes in a tree being its worst case.

Question assigned to the following page: 3

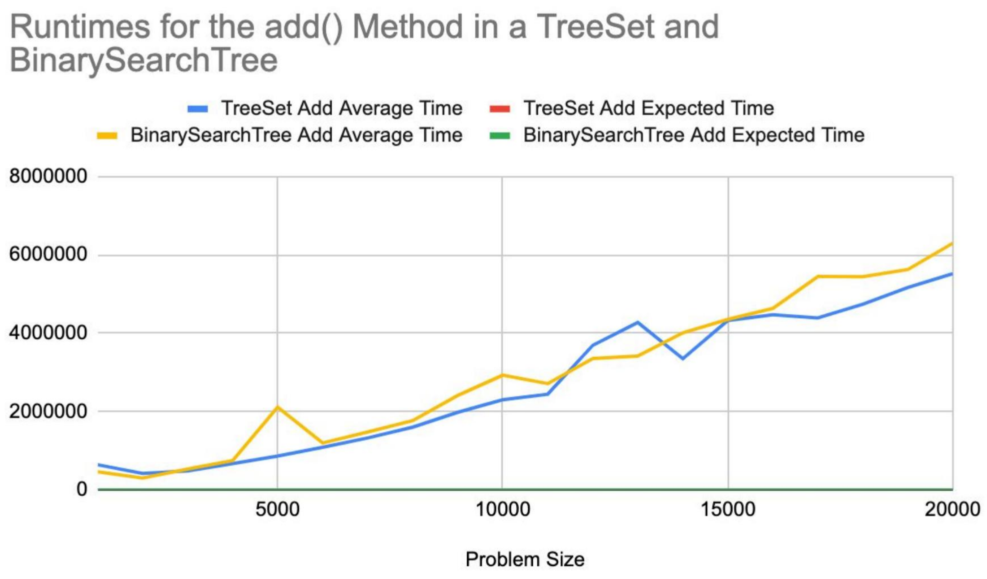
3. In order to illustrate the effect of building an N-item BST by inserting the N items in sorted order versus inserting the N items in a random order, we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 1,000 and went up to no greater than 20,000. We then created an ArrayList of type Integer with every number from 0 to one less than the problem size (int i = 0; i < probSize; i++). After adding every number from said ArrayList into our BST, we then recorded the average time and the expected time required to invoke our *contains* method for each item in the sorted BST. We then ran the same test for a BTS of type Integer with every number from 0 to one less than the problem size, but this time the BTS was in a permuted random order. We then recorded the average time and the expected time, using the "CheckAnalysis" technique, required to invoke our *contains* method for each item in the unsorted BST.



As demonstrated by the graph, the *contains* method shows quadratic growth (O(N^2)) for the sorted, in order, BST while the unsorted BST displays linearithmic (O(NlogN)) growth. The runtimes of the sorted BST displays quadratic growth because the *contains* method must loop through every node in ascending order to find the correct item it is looking for, and it must do this for every node (N * N). The runtimes of the unsorted BST displays linearithmic growth because the *contains* method loops through half of the nodes on average in order to find the correct item it is looking for, and it must do this for every node (logN * N). These two runtimes are expected since the “CheckAnalysis” technique shows us that the expected time converges to a positive number, meaning this method behaves the way we expect it to with the given BTSs. It would seem that the unsorted BTS is more efficient, which is expected since it displays linearithmic time which is faster than quadratic time.

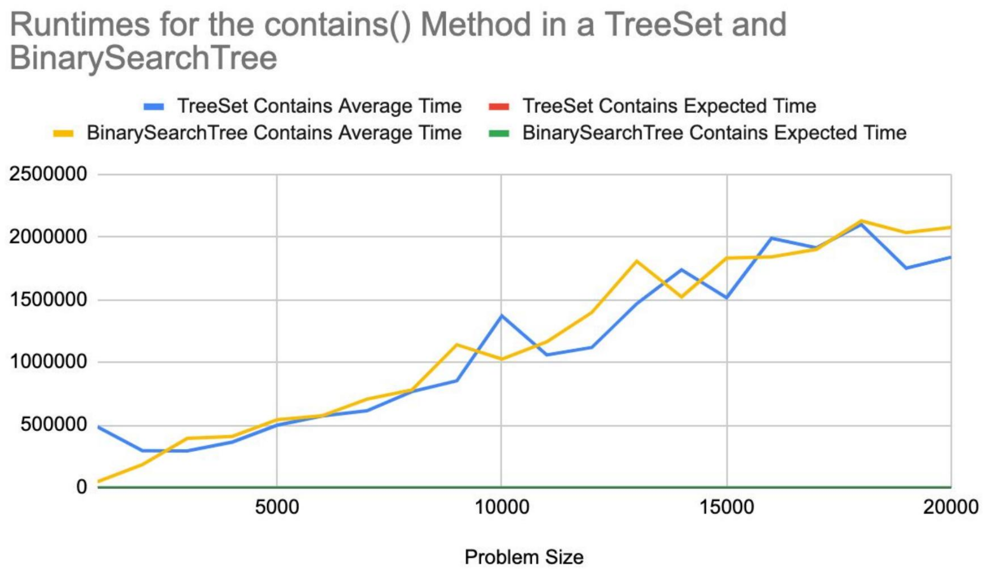
Question assigned to the following page: 4

4. In order to illustrate the differing performance in a BST with a balance requirement and a BST that is allowed to be unbalanced, we first ran the test through a for loop which incremented the problem size to be used in each loop by sizes of 1,000 and went up to no greater than 20,000. We then created an ArrayList of type Integer with every number from 0 to one less than the problem size (int i = 0; i < probSize; i++) and then shuffled its contents so that the list of integers was in permuted order. We then recorded the average time and the expected time required to invoke our *add* method for each item in the list into both Java's *TreeSet* and our *BinarySearchTree*. We then recorded the average time and the expected time, using the "CheckAnalysis" technique, required to invoke our *add* method for each item from the ArrayList into the *TreeSet* and the *BinarySearchTree*.



As demonstrated by the graph, both the *TreeSet* and the *BinarySearchTree* method show linearithmic (O(NlogN)) growth because the *add* method loops through half of the nodes on average in order to find the correct item it is looking for in both trees, and it must do this for every node (logN * N). These two runtimes are expected since the "CheckAnalysis" technique shows us that the expected time converges to a positive number, meaning this method behaves the way we expect it to with the given BTSs. It would seem that the *add* method for Java's *TreeSet* is slightly more efficient, which tells us that the self balancing nature of Java's *TreeSet* makes the code run more efficiently than an our unbalanced *BinarySearchTree*.

Questions assigned to the following page: 4, 5, and 6



As demonstrated by the graph, both the *TreeSet* and the *BinarySearchTree* method show linearithmic (O(NlogN)) growth because the *contains* method, like the *add* method, loops through half of the nodes on average in order to find the correct item it is looking for in both trees, and it must do this for every node (logN * N). These two runtimes are expected since the “CheckAnalysis” technique shows us that the expected time converges to a positive number, meaning this method behaves the way we expect it to with the given BTSs. It would seem that the *contains* method for Java’s *TreeSet* is also slightly more efficient, which tells us that the self balancing nature of Java’s *TreeSet* makes the code run more efficiently than an our unbalanced *BinarySearchTree*.

5. I think that a BST is a good data structure for representing a dictionary only if it is balanced. Because searching for a word within the dictionary, which would be a BST and would only loop through half of the nodes on average in order to find the correct item it is looking for, would be much more efficient than trying to go through every single word in the dictionary in order to find the correct word, even longer for a list of words, you are looking for.

6. The problem that would occur for a dictionary BST if it is constructed by inserting words in alphabetical order would be that the dictionary would be a very unbalanced, right heavy tree. The worst case to find a word would be the total number of words within the dictionary itself, which takes way too long and is very inefficient and cumbersome. The best solution would be to add a self-balancing nature to the BST like Java’s *TreeSet* which would keep the tree balanced and have any case of finding a word be logarithmic as opposed to linear.