Andy Dao | uID: u0692334 | CS 2420

ANALYSIS DOCUMENT ASSIGNMENT 8

1. **Who is your programming partner? Which of you submitted the source code of your program?**
   Sam Bridge was my partner and I will be the one to submit the source code.

2. **Evaluate your programming partner. Do you plan to work with this person again?**

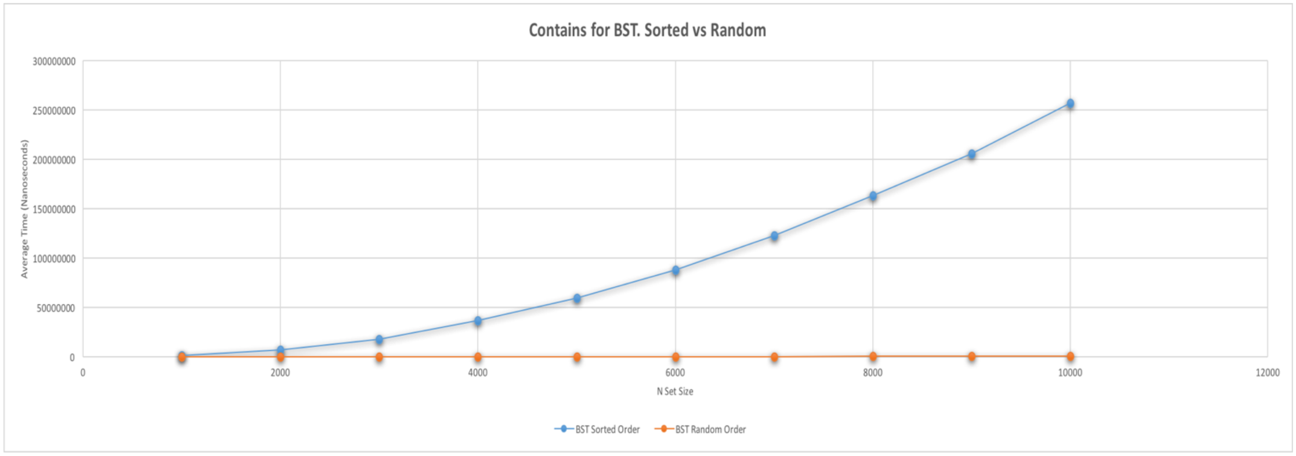
   Sam was a great partner. Very communicative and is pretty down to earth. One of the reasons why I liked working with Sam was because he asked a lot of questions about the assignment, such as how a certain method is supposed to work and its runtime, etc. This allowed me to teach him, but the benefit was actually trying to explain it to him, allowing me to learn the material better as we go. We definitely plan to work with each other for the next partner assignment.

3. **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. Submit any code required to conduct your experiment with the rest of your program and make sure that the code is well-commented. 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 is critical.**
   Sorted Order vs Random Order:

   Our experiment is simple really. For the sorted portion of our test, we have our N set sizes goes from 1000 to 10000, incrementing by 1000. We add into our BST starting from 0 to our N set size, then right after the integers are added, we start timing our contains method for each N items in our BST (so for a set size of 1000, we do the contains method call from 0 to N to hit each element) and add to the total time. We do this portion to what we set our ITER_COUNT to, which was 100. We then at the end (set size of 10000), divide the total time added for each of those N set sizes, by the iteration count that we talked about earlier to give us our final average times.

For our random order test, we did the same, but to have it inserted by a random order, what we did was make an ArrayList and add integers from 0 to N set size as normal. This gives us a sorted array list, but then we call the Collections.shuffle(list) method on the array list, to mix up the array list in random order, which we then have our BST call the addAll() method on that list so that way, we have a randomly inserted order BST. The following graph is the result of our timing test:

Figure 1:



What we see here, is that when running the contains method on each N items on the randomly inserted order BST, we got a run time of O(log N), and for our sorted order BST, we got a run time of O(n). This is to be expected, because when we insert into the BST in a sorted order fashion, it becomes a right-heavy tree, since each integer after the one that was inserted is larger than the previous. That means we have to actually touch each integer to reach the integer that is passed into contains. This is the worst case run time for a Binary Search Tree, due to it being a right-heavy tree. This essentially makes it act like a linked-list as each operation doesn't cut the tree in half.

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 (http://docs.oracle.com/javase/7/docs/api/java/util/TreeSet.html) as an example of the former and your BinarySearchTree as an example of the**

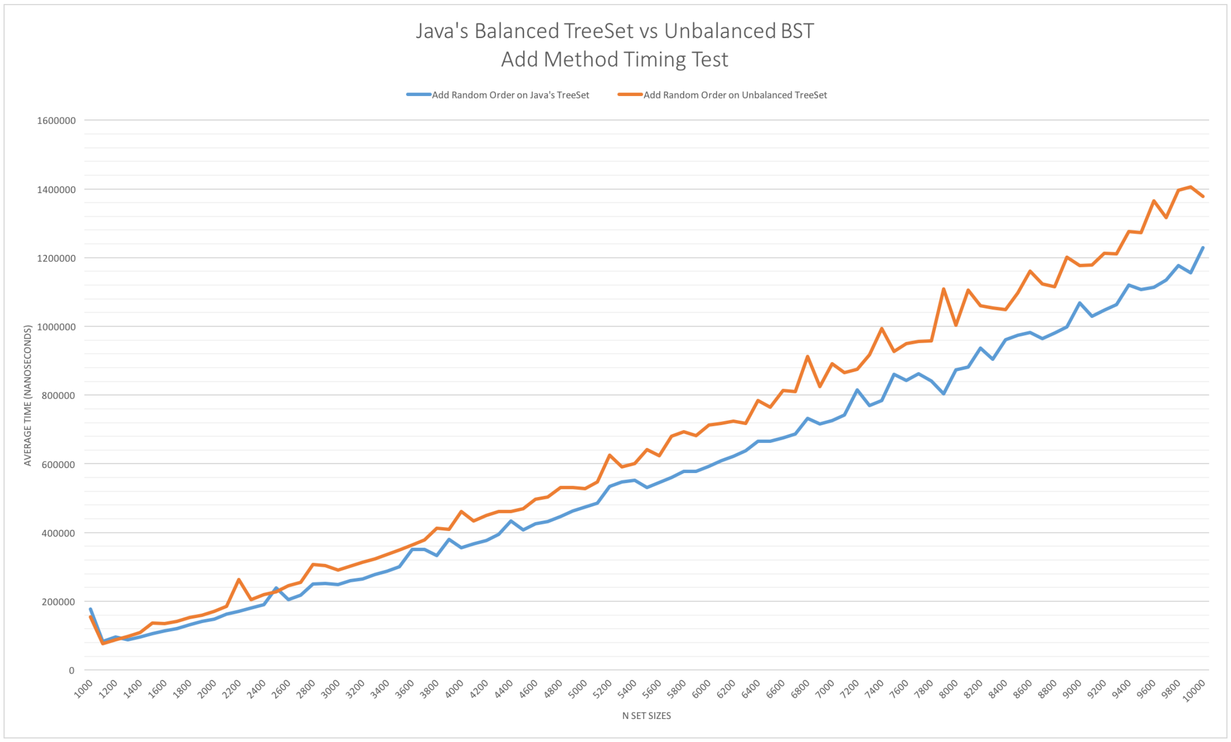
latter. Java's TreeSet is an implementation of a BST which automatically re-balances itself when necessary. Your BinarSearchTree class is not required to do this. Carefully describe your experiment, so that anyone reading this document could replicate your results. Submit any code required to conduct your experiment with the rest of your program and make sure that the code is well-commented. 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 is critical.

To compare the efficiency between Java's Tree Set and our implementation of a Binary Search Tree without the requirements of being balanced, we first made an array list from 1000 to N set size, which was max at 10000 items, incrementing by 100 for each loop. Then we randomize it to get a random sorted array list. We had two different methods, one for adding and one for contains, but we made sure to run either add or contains on the same array list for the specific method so that way the data won't be skewed due to non-consistent list insertion or searching. We will talk about the add method first.

Add Method | Java Tree Set vs Unbalanced BST – Inserted Random Order:

So here is our graph output:

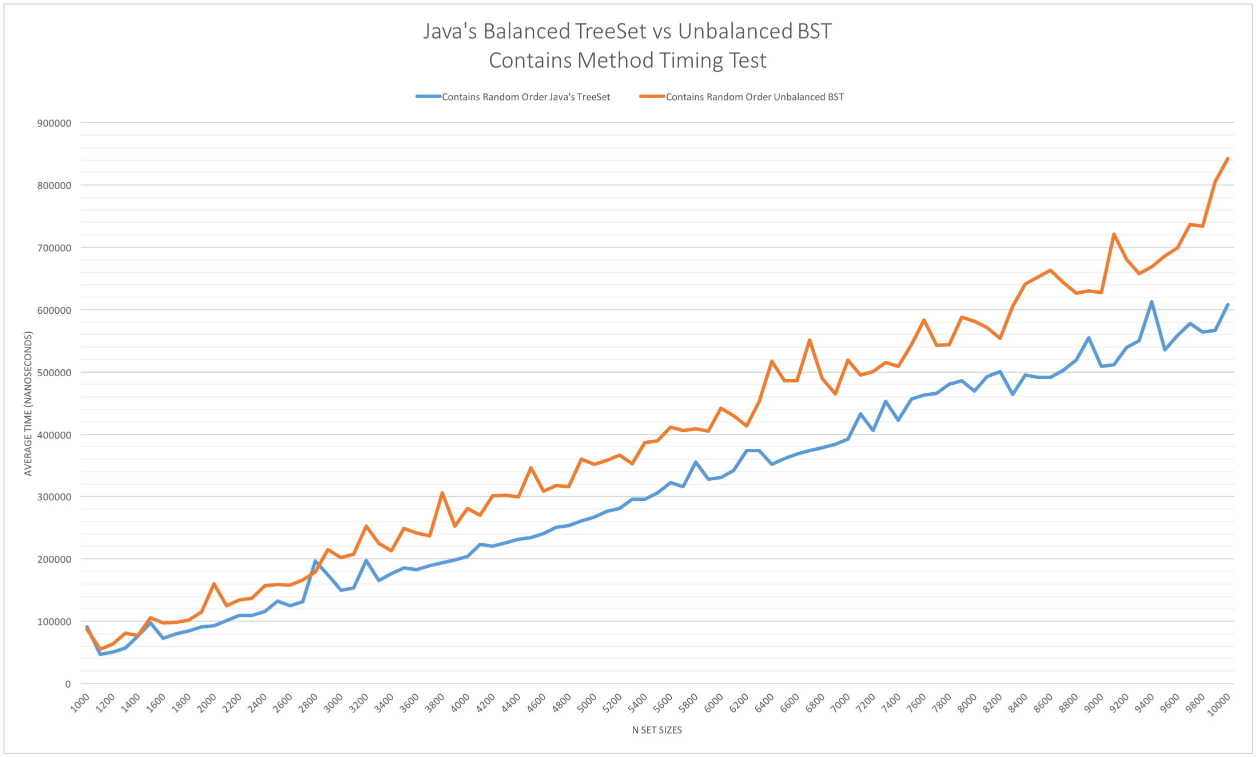
Figure 2:



What we can see here is that Java's Add method ran a bit quicker compared to our BST implementation, though they do both run at O(log n) run time, adding the array list that was in random order. If we had to run the add method with a sorted array list, we can't perform O(log n) and have to go through each item to get to the insertion point (this is specific to the unbalanced BST that we implemented since we don't have an requirement for a balanced tree). With Java's tree set, it is a AVL tree, which is a self-balancing tree. This gives us guarantee run times of O(log n) for searching, insertion, and deletion, in both average and worst case.

Contains Method | Java Tree Set vs Unbalanced BST – Inserted Random Order:

Figure 3:



We can see here again that Java's Tree Set on contains runs quicker than our implementation of the Binary Search Tree, though again, they both run at O(log n). We believe that this is because Java's Tree Set rebalances along the way as it is being inserted, and is actually balanced unlike our unbalanced BST, which is relying on how random our random order insertion is (though still runs O(log n) average time). So contains can run at a much "truer" balanced list giving us a better O(log n).

5. **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? Explain what you could do to fix the problem.**

   Since it's inserted in alphabetical order into our Binary Search Tree that doesn't have a requirement to be balanced, the problem that we will create is that it will become a right-heavy tree. One of the ways we can combat this problem is by starting from the middle of our dictionary(s) and insert that word, then we can add the middle word that's in between the middle and the last word, and continue doing so. If we want to be very messy but not care how we insert the words, we can just insert the words randomly.

6. **How many hours did you spend on this assignment?**
   We spent about 15 hours on this assignment.