Stephen Hogan

U0813633

Analysis 08

1. **Who is your programming partner? Which of you submitted the source code of your program?**

Chris Grayston (u0906710). I submitted the source code for the assignment.

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

Chris is an awesome partner. I feel like we worked well together and that we organized our work well. We set goals and plans to finish at certain times and dates and worked hard to do that and it kept us on track with where we were supposed to be. He is also proactive in working on the assignment. I do plan on working with him again after I switch partners.

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.**

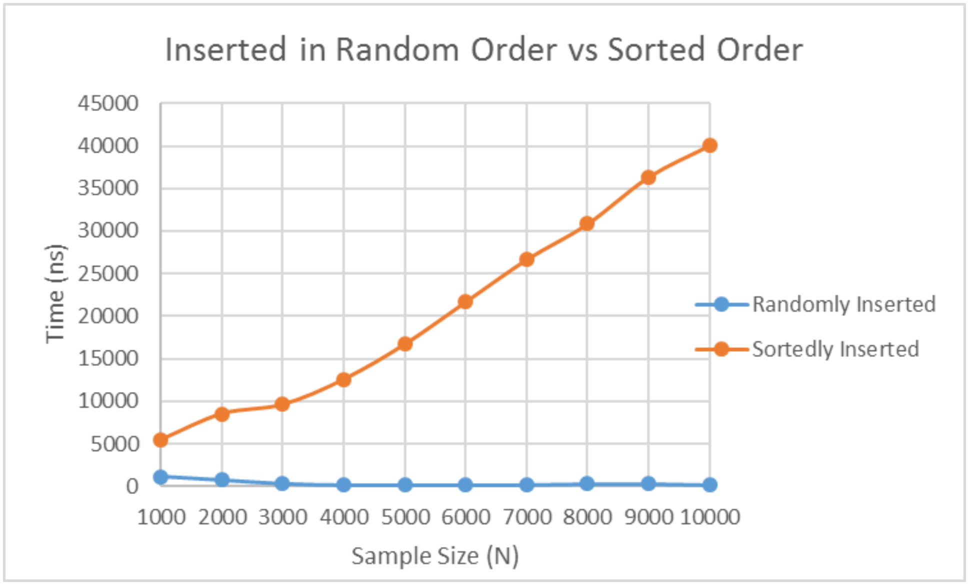
One suggestion for your experiments is:

   a. **Add N items to a BST in sorted order, then record the time required to invoke the contains method for each item in the BST.**
   b. **Add the same N items to a new BST in a random order, then record the time required to invoke the contains method for each item in the new BST. (Due to the randomness of this step, you may want to perform it several times and record the average running time required.)**
   c. **Let one line of the plot be the running times found in #1 for each N in the range [1000, 10000] stepping by 100. (Feel free to change the range, as needed, to complement your machine.) Let the other line of the plot be the running times found in #2 for each N in the same range.**

To test the effect of having inserted items in sorted order into the BST or in random order, first initialize a BST. Then you want to generate a random list of numbers of size 1000 (varies based on the incrementation of 1000) that has all values from 0 to the size in it. Since it is random order, we will add each of the elements of random numbers list into the BST through addAll. Once that has been done, we should then start a timer and call the contains method on the BST with the argument being the greatest possible value in the random numbers list (or even any number higher than that because it will traverse the entire BST either way). After the contains method finishes, we should end the timer and then add that into a totalTime variable. We should run that portion of the test about 100 times to and average the totalTime by 100 so that we get a good representation of what is actually happening and it reduces the effects of outliers. After we have done that we should print the results to the console or to a filewriter and plot the numbers for the randomly added BST. We should then increment the size of the list and repeat the entire process for say items N = 2000 until we get to N = 10000.

For the sorted BST, we should do the exact same experiment as before except instead of adding in the numbers in a random order, we want to add them in a sorted way. There are two ways to do this, you can either sort the random number list that you had from part a and addAll those elements into this new BST or you can just use a for loop and add in numbers 0 – size (1000 or more depending on the incrementation). This way, we will get a right-heavy BST which will essentially turn it into a LinkedList. This makes the complexity worse for the contains method.

After both the sorted and random BSTs have been timed, we should then plot the numbers into an easy-to-read graph similar to the one below:



The graph above shows the behavior of the sorted BST and the random BST and their contains method's runtime. As seen, the randomly inserted BST has a far better complexity than the sorted BST. This makes sense because a normal (roughly balanced) BST should have O(log N) complexity for contains because as it gets to each sub tree it gets rid of "half" of the elements in the tree (half is in quotations because it is roughly half if it is roughly balanced). This is shown in the graph in the blue line. The sorted BST is the same thing as a LinkedList which means that its complexity should actually be O(N) which is also shown above in the orange line. This makes sense because as you get to each subtree while traversing the list, you are ONLY getting rid of one element each time. This is the same as if you were iterating through a LinkedList because you have to look at each element and each time you move to the next element you are only removing one element from your potentially correct answers.

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 BinarySearchTree 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.

One suggestion for your experiments is:

a. **Add N items to a TreeSet in a random order and record the time required to do this.**
b. **Record the time required to invoke the contains method for each item in the TreeSet.**
c. **Add the same N items (in the same random order) as in #1 to a BinarySearchTree and record the time required to do this.**
d. **Record the time required to invoke the contains method for each item in the BinarySearchTree.**
e. **Let one line of the plot be the running times found in #1 for each N in the range [1000, 10000] stepping by 100. (Feel free to change the range, as needed, to complement your machine.) Let the other line of the plot be the running times found in the #3 for each N in the same range as above.**
f. **Let one line of a new plot be the running times found in #2 for each N in the same range as above. Let the other line of plot be the running times found in #4 for each N in the same range. (You can combine the plots in the last two steps, if the y axes are similar.)**

I hypothesize that the TreeSet should be slower in adding than our BST because it has to readjust the tree at certain times.

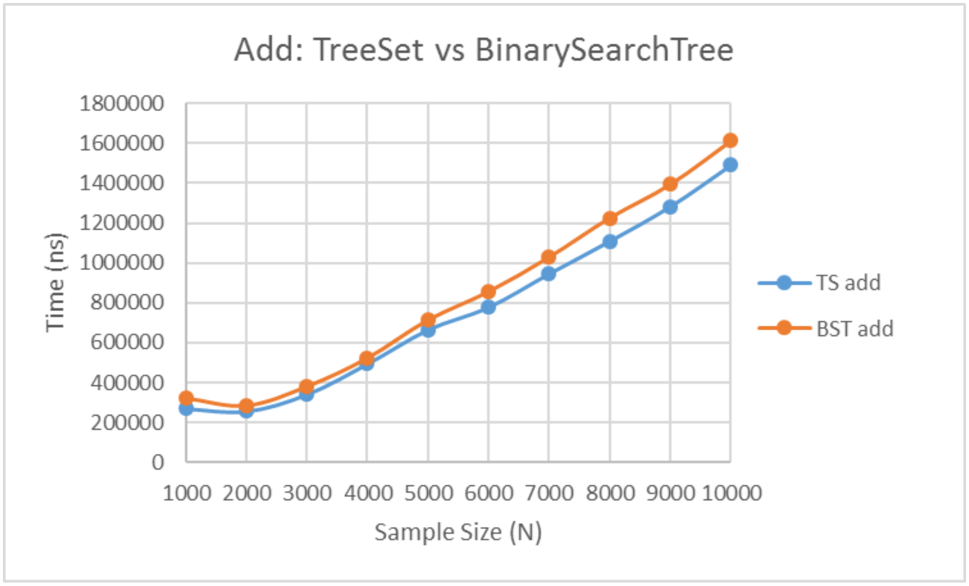
To test the difference between our BST and Java's TreeSet (automatically rebalances itself when necessary) we should begin by initializing a BST and a TreeSet. Then, we should make a random list of numbers of size [1000, 10000] (depending on the incrementation). Once we have a list of random numbers [0, 1000] we can then record the amount of time it takes for the TreeSet to addAll of these elements into it. To do this we should start a timer, then call the addAll method on the TreeSet and our random numbers list, and then end the timer. This way we get an accurate representation of how long it takes to add each element into the TreeSet. We should add this time to a totalTime variable and run this section of the timing over again say 100 times to get an accurate representation of the actual time it takes. We should continue to add to the totalTime and then divide it by 100 when we are done to get the average time it took to complete the addAll method. We should then display our results to the console and repeat this same test with a larger sample size (increment by 1000 until you reach 10000).

To test the addAll for the BST, you do the same thing except when you start the timer, you call addAll on the BST and not the TreeSet.

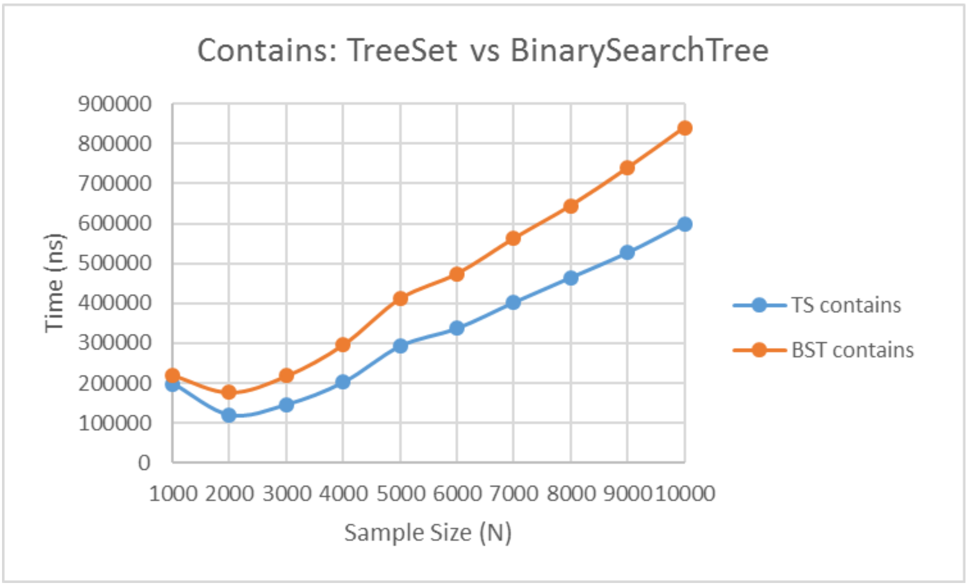
Now, to test the contains method for both, we can make a new method and set it up in a similar way (create the TreeSet and BST, make an iteration and incrementation, generate a random numbers list etc.). Once we have that set up, we can addAll the random numbers into the TreeSet and then start a timer, call contains on TreeSet with either the largest element in the random numbers list or anything bigger than that (it will search through the entire tree either way), and then end the timer. We should go about the same process as before: add to total time, repeat 100 times, and then report results to the console and increment the sample size and do it again.

To test the contains for the BST, you do the same thing except when you start the timer, you call contains on the BST and not the TreeSet.

After all the data has been collected and organized, it should be plotted into graphs to analyze the data as seen below:



Above is the graph of the addAll method being tested for a TreeSet and a BST. Surprisingly, the TreeSet took less time than the BST. I figured it would take longer because it had to reorganize the tree every so often. It could have come out this way due to the fact that Java are better programmers than I or that there is a clever way of inserting the item that doesn't really change the complexity. As of now, the complexity of both is O(log N) it appears which is what is expected. I was surprised at the results but it is comforting to know that the plot of our BST is very similar to the plot of Java's TreeSet.



Above is the plot of the contains method for our BST and Java's TreeSet. I hypothesized that they would be the same because there shouldn't be any added work for Java's TreeSet because it shouldn't have to reorganize the tree at any point. Actually, a better hypothesis would have been that the TreeSet would have been faster because since it is perfectly balanced, it really does cut in half the tree EVERY time whereas our BST is not necessarily half but only close. The graph confirms that second hypothesis because the complexities are VERY similar and I would say they are both O(log N) however, Java's TreeSet is faster than our BST. This could be because of the halving principle, also I think Java is better at implementing data structures than I.

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.**

In our case, our BST is not required to be balanced and therefore when items are added in a sorted order, it creates a left- or right- heavy tree. This causes problems because then our binary search algorithm actually has O(N) and not O(log N) because when it moves down the tree it doesn't remove "half" of the tree it only removes 1 element. This makes the running time of a spell checker or contains method much, much slower and it defeats the purpose of using a tree. At this point we might as well just use a linked list.

To fix this, you could put the words in a list and then access a randomly generated index until all items have been added to the BST. Another way to fix this is to put the words in a list and then access the middle item using the middle index and then get the item at the middle index of the left half and so on until you have accessed all elements. This would result in a balanced BST which would be ideal for any operation using the binary search method.

6. **How many hours did you spend on this assignment?**

I spent probably around 15 hours on this assignment total with my partner.