Samantha Pope

Analysis Document A3

1. **If you had backed the sorted set with a Java List instead of a basic array, summarize the main points in which your implementation would have differed. Do you expect that using a Java List would have more or less efficient and why? (Consider efficiency both in running time and in program development time.)**

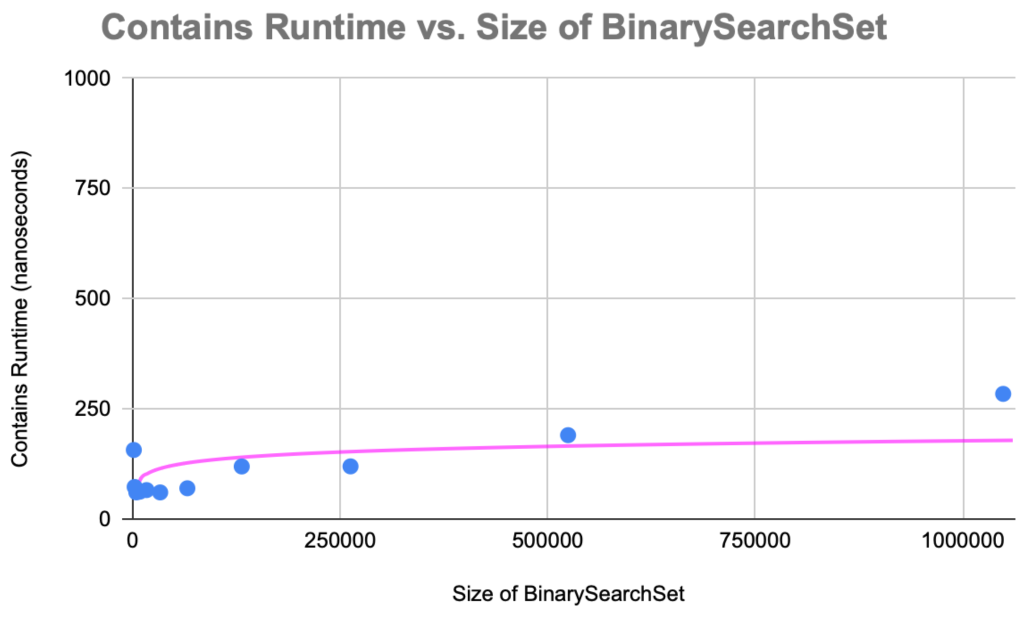
If I had backed it with a Java List, then a lot of the methods would have already existed so we wouldn't have had to write them. However, I was able to have more control over what happened with the array and what different functions meant in my code. I think that it could have been more efficient with writing the program, because I am familiar with using the already written methods.

2. **What do you expect the Big-O behavior of BinarySearchSet's contains method to be and why?**

I expect the runtime of the BinarySearchSet contains method to be similar to my binarySearch. My binarySearch method I expect to have a runtime of O(log(N)). My contains method calls my binarySearch method, and then it uses whichever comparator is not null(dependending on which constructor is used) and compares the element found at that index with the array as a check. I do not think the comparing of those two elements takes enough time for the big O to be affected. I bet it adds time (ex: O(logN + compareTime) but it is negligible in terms of bigO.

3. **Plot the running time of BinarySearchSet's contains method, using the timing techniques demonstrated in previous labs. Be sure to use a decent iteration count to get a reasonable average of running times. Include your plot in your analysis document. Does the growth rate of these running times match the Big-oh behavior you predicted in question 2?**

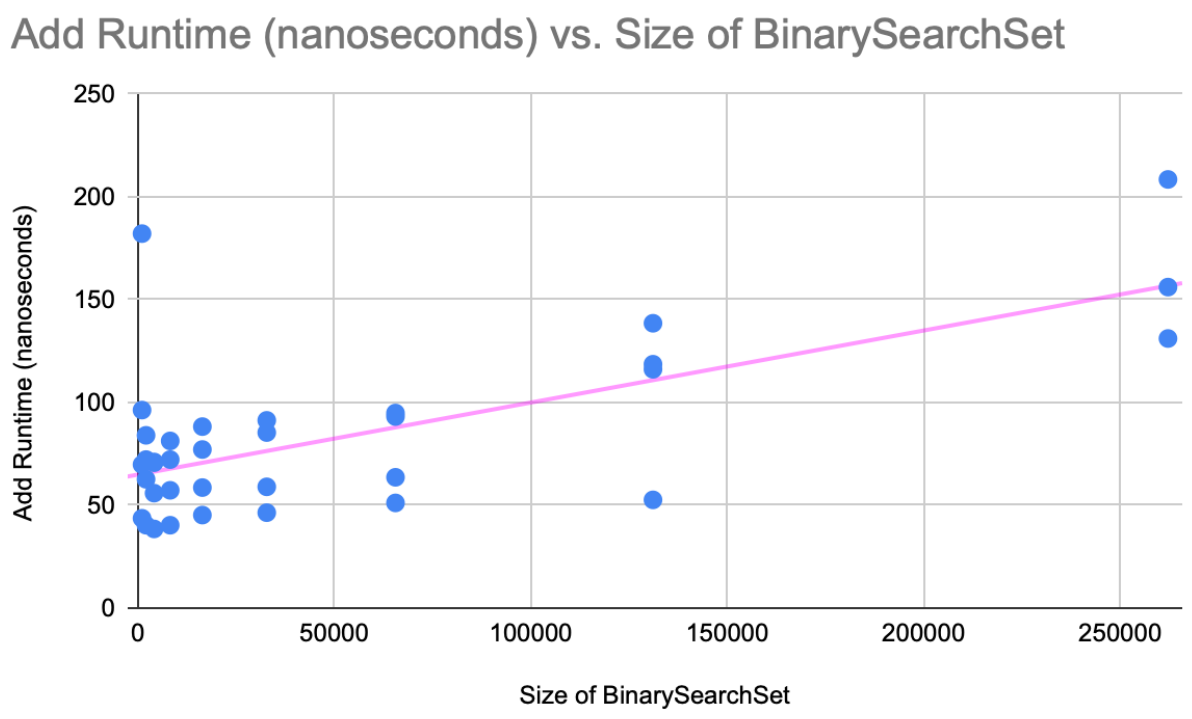
   I ran my timing experiment for 5,000 iterations and produced the results shown in the following graph. The code is also available on my github repository. This does match the behavior of what I predicted in question 2. The function grows extremely slowly and follows a logarithmic trend, indicating a big O of log(N). Note also the amount that the x-axis grows compared to the y-axis.



4. **Consider your add method. For an element not already contained in the set, how long does it take to locate the correct position at which to insert the element? Create a plot of running times. Pay close attention to the problem size for which you are collecting running times. Beware that if you simply add N items, the size of the sorted set is always changing. A good strategy is to fill a sorted set with N items and time how long it takes to add one additional item. To do this repeatedly (i.e., iteration count), remove the item and add it again, being careful not to include the time required to call remove() in your total. In the worst-case, how much time does it take to locate the position to add an element (give your answer using Big-oh)?**

For my first analysis, I tested how long it took to add one element to an array of differing sizes. I added the element inside of the time capture, and removed it outside of the time capture. This allowed me to focus only on the time it takes to add one element to an varying size of array. I had predicted this has a Big O runtime of O(N) because all of the elements are copied into a new array. This would be time consuming, especially at large sizes of the array. However, I received similar results to the contains method. This is illogical because the contains() method is called in the add method. I tested this many times for different iteration sizes. I added random elements (after removing them from the array) using random.nextDouble. I also added in elements that I thought would be “worst case scenario” (small elements in a large array with all of the elements listed after it.

Here is the graph from my adds method:


Add Runtime (nanoseconds) vs. Size of BinarySearchSet
Add Runtime (nanoseconds)
Size of BinarySearchSet