When you are satisfied that your program is correct, write a brief analysis document. Ensure that your analysis document addresses the following.

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.)

   1) The first and foremost difference between Java List and array is that array must Have fixed size at compile time, whereas List is dynamically sized and can be declared at run time. So we wouldn't need to declare a temporary array capacity in the constructor had we used Java List.
   2) List itself is backed with basic array, and the API contains most of the methods we need to implement, so it will make some of our BinarySearchSet methods implementation a lot easier by simply calling List's built-in methods. For example, methods like toArray(), remove(), contains(), we can simply call them from List library→ more efficient in program development time
   3) I would expect using ArrayList to be less efficient at run time. The reason is that for most of the methods implementations of BinarySearchSet backed by ArrayList, we have to call ArrayList's built-in methods to achieve it. So essentially we are calling ArrayList's method on our call stack, which allocate memory on the heap. Whereas if we simply use array to back the BinarySearchSet, we don't have as many methods calls on the call stack. This will decrease the space complexity of our data structure.

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

Answer:

It should be LogN. This is because I called binearySearch method to determinate whether The element is contained in the set or not. The Big-O behavior for bineary search is logN.

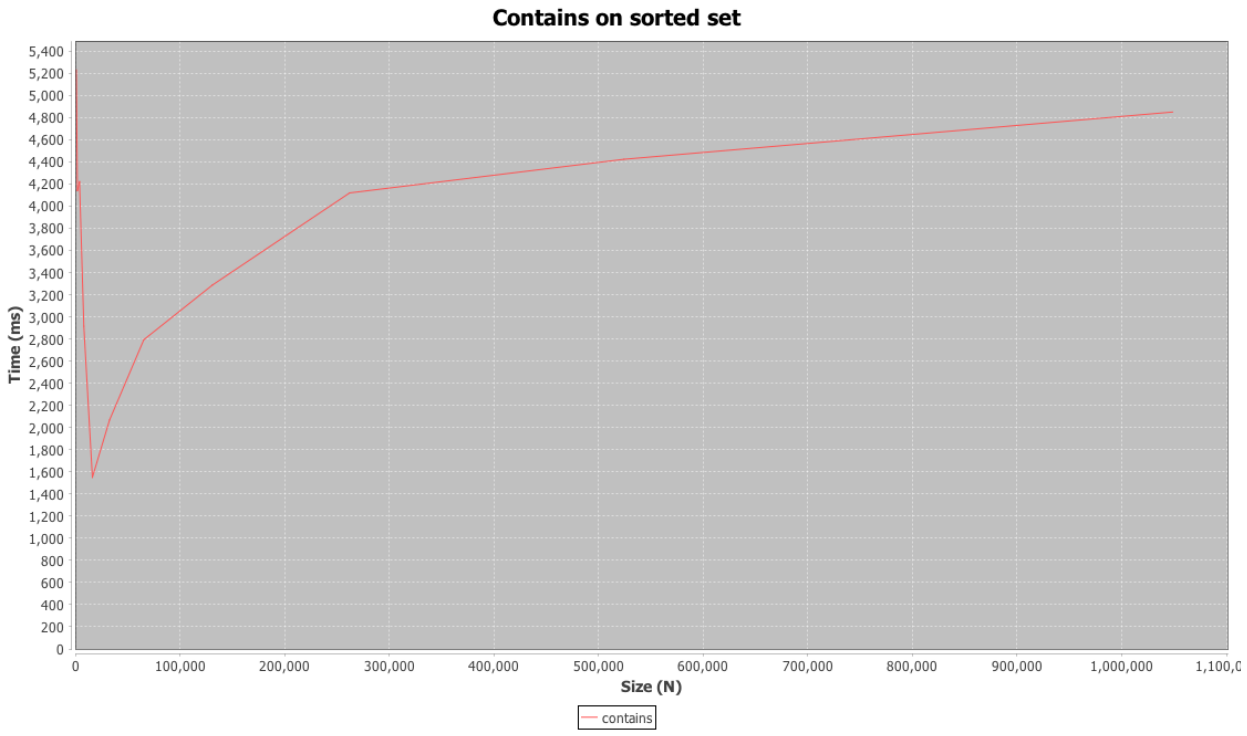
3. Plot the running time of BinarySearchSet's contains method, using the timing techniques demonstrated in previous labs. Be sure to use a decent iteration count to get a reasonable average of running times. Include your plot in your analysis document. Does the growth rate of these running times match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

Answer:

The growth rate of this running time matches the growth rate of a logN function**.**

Data collection:

I collected 10 data points in total with each BinarySearchSet size is twice as big as the previos data point. For getting the average running time of each binarysearch set size, I have an iteration of 100, and average out the total time. I then call contains() of a randomly generated integer on every single binarysearch set I have created. I then plotted these 10 data points and obtained the result as below.



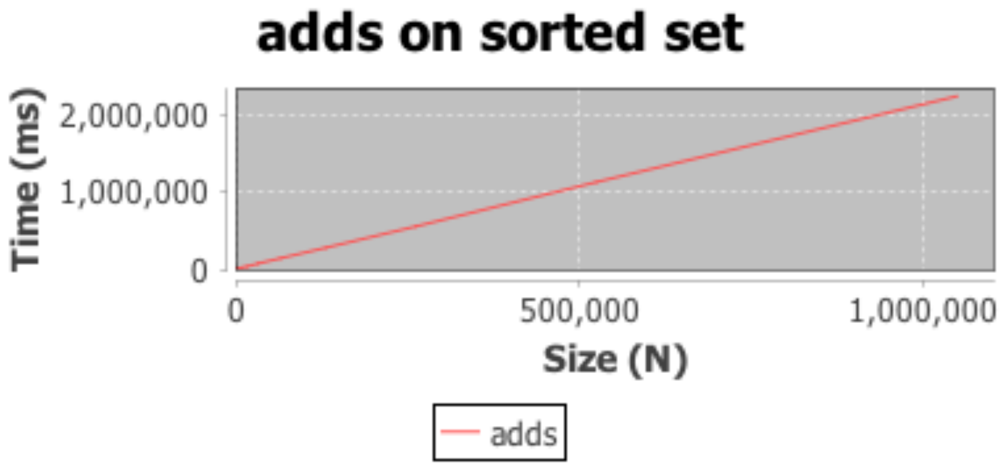
4. Consider your add method. **For an element not already contained in the set, how long does it take to locate the correct position at which to insert the element?** Create a plot of running times. Pay close attention to the problem size for which you are collecting running times. Beware that if you 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)**?

Answer:

The time complexity of worst-case to locate the insert position is O(logN). In worst-case, either the target is not in the set, or the very first or last element of the set is the target. We half the set iteration to compare the middle element with the target. So the time complexity is O(logN).

Data Collection:
I collected 10 data points. For each set size, I iterate 100 times to average out the total time of calling the add() method. I add -1 (which is guaranteed not presented in the set) to the set and stop the nano Timer right after the all of add(-1) to ensure that I don't time the remove(-1) call.

Eventually, these 10 data points give me a linear graph which is what I expected it to be. Since in my add function, I firstly call binary search to find out the correct index to insert the element, the time complexity is O(logN). For the actual insertion operation, it is linear complexity. Based on asymptotic behavior principle, the overall time complexity is O(N).


adds on sorted set
Time (ms)
Size (N)
adds