CSC 385

Module 3 Homework: Using Binary Search to Find Insertion Points in an Array

20 points

Introduction:

Suppose that you want to add items to an array such that the items are always ordered in ascending order; e.g., [1, 2, 2, 4, 5, 8, 9, 14, 32], and any duplicate values are adjacent to each other. We haven't talked about sorting algorithms yet, so assume you want to be able to keep the items in the array in order without having to sort them. So for example, suppose you want to add the integer value 7 to the array [1, 2, 2, 4, 5, 8, 9, 14, 32]. If you aren't allowed to sort the array, you need to have a way to search for the correct position, or insertion point, for the value 7 such that the items in the array will remain in ascending order. In this case, the value 7 would need to be inserted at position 5, and the resulting array would be: [1, 2, 2, 4, 5, 7, 8, 9, 14, 32]. Of course, to perform the actual insertion all the items at positions 5 to the end of the array would need to be shifted to the right to make room for the new item, and the array would need to be resized, but for this assignment we will just focus on how to determine the correct insertion point.

Finding the correct insertion point involves searching the array. A sequential search would work for this. If we begin a sequential search from the left side of the array, we would simply traverse the array towards the right until we either reach an item whose value is greater than or equal to the value of the item we are inserting, or we reach the end of the array. We could similarly begin the search from the right side of the array and traverse towards the left side. The best case scenario for this, as discussed in the Module 3 lecture, would be O(4), and corresponds to the case where the item we are inserting is less than or equal to the first item in the array. The worst case scenario would be O(4N + 3), and corresponds to the case where the item we are inserting is greater than the last item in the array. The average case would be on the order of O(2N), with some slight variation possible, depending on whether there is an even number or an odd number of items in the array.

O(N) running time isn't bad, but if we have 1 million items in the array, we will need to execute around 4 million statements in the worst case, and around 2 million statements in the average case. Using the binary search would be far more efficient, giving us a running time on the order of O(log2N). For a 1 million item array, this translates to around only 20 statement executions in the worst case, and somewhat less than that on average—drastically more efficient!

Figure 1 shows the code for the standard binary search algorithm from the lecture for Module 3. The algorithm returns the index of the search item if the search item is in the array. If the item is not in the array, it returns -1. The -1 return value is certainly useful as an indicator that the search item isn't in the array, but it tells us nothing about where an item that is not in the array should go. The goal of this assignment is to modify the binary search algorithm to return the correct insertion point for an item, whether or not the item is present in the array.

```
public int binarySearch(Comparable[] objArray, Comparable searchObj)
{
    int low = 0;

    int high = objArray.length - 1;

    int mid = 0;

    while (low <= high)
    {
        mid = (low + high) / 2;

        if (objArray[mid].compareTo(searchObj) < 0)
        {
            low = mid + 1;
        }
        else if (objArray[mid].compareTo(searchObj) > 0)
        {
            high = mid - 1;
        }
        else
        {
            return mid;
        }
    }

    return -1;
}
```

Figure 1. A straightforward, iterative, binary search algorithm.

Modifying the Binary Search Algorithm

The first part of this assignment is to modify the binary search algorithm such that the following conditions are met:

1. The algorithm returns an index where a specified item should be inserted such that the ordering of all items will be preserved after the insertion has occurred. Note that we are not concerned here with performing the actual insertion, only with returning the insertion point.

2. The algorithm should always return a **non-negative** integer regardless of whether or not the item to be inserted is already in the array. Again, since we are not concerned with performing the actual insertion, it is acceptable for the algorithm to return an index that is greater than the current length of the array. It will be assumed that some other method will handle any array resizing and item shifting. In other words, assume someone else will be writing an insert method that is responsible for actually inserting items into an array. This method would call your modified binarySearch algorithm to get the correct insertion index, and then insert the item.

3. Your algorithm **must** use `Comparable` objects. Do **not** write the algorithm so that it only works with primitive data values such as `int` or `double`. If you want to use integers or primitive data, use the built-in wrapper classes (`Integer`, `Double`, `Long`, `Float`, `Character`, `Byte`, etc.). These wrapper classes already implement the `Comparable` interface, so they are `Comparable` objects. Java's autoboxing feature will automatically convert a primitive data value to the appropriate wrapper type, eliminating the hassle of manually instantiating wrapper objects. For example, you can create an array of `Integer` objects by typing: `Integer[] intArray = {1, 2, 3, 4, 5, 6};`.

   (One cautionary note: keep in mind that characters and character strings are compared

using ASCII codes, which means, for example, that an upper case 'Z' is considered to be less than a lower case 'a'.)

4. I will test your code by pasting your modified method in my own test harness. Be sure to remove all debugging code from your method, and make sure your method does not need to call other methods or reference any global variables in order to work.

Testing Your Implementation

By now, I'm sure everyone is aware of the importance of testing code, but because this particular assignment lends itself well to being an exercise in test case generation—since the test cases for this problem can be easily elucidated—we will incorporate test case creation as part of the assignment. There are 3 conditions that need to be tested that are related to the positioning of the item being inserted:

1. the lower boundary of the array
2. the upper boundary of the array
3. between the upper and lower boundaries of the array

For each of the three conditions there are two possibilities:

1. the item already exists in the array
2. the item does not exist in the array

Furthermore, the array may contain:

1. an odd number of items, or
2. an even number of items.

The combinations of the above factors yield a total of 3 * 2 * 2, or 12 test categories, assuming the array is not empty. The empty array yields a 13th test category, which should also be tested.

To test your code, do the following:

1. Create a set of 13 test cases, one test case for each of the 13 test categories described above. Recall that every test case should consist of 1) inputs and 2) expected output. Each test case should consist of an array of Comparable objects and an object of the same type to be inserted into the array, as inputs, and the correct insertion point for the item to be inserted, as the expected output. You can create your tests using jUnit or some other testing library, or you can create your own test harness. A test harness is simply a program that passes the inputs of test cases to the code to be tested, and checks the actual output against the expected output. The results are displayed on the screen or written to a testing log file.

2. With your test cases in hand, test your modified `binarySearch` method.

What You Need to Turn In

1. the .java source code for your modified binarySearch method

2. the set of test cases you used to test your method (**this is important—if you don't turn in your test cases you won't get full points**). For this part you need to turn in a table listing the following information: each of the 13 test categories

    a. each of the 13 test categories
    b. the array of items you used for each test
    c. the item you tested for insertion for each test
    d. the index you expected the algorithm to return
    e. the index the algorithm actually returned

Rubric:

	Points
Correctly passes all 13 test categories	13 (1 pt. per test category)
Submission of your own test cases	7
Total:	**20 points**