CSC 385
Homework Assignment, Module 3
20 points

Solution and Grading Rubric

Solution

Below is one solution to the assignment — other solutions are possible.

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

    return low;
}
```

As it turns out, the algorithm already returns the correct insertion point if the item being inserted is already in the array. It only returns an incorrect value (-1) when the item is not in the array. The way the algorithm works, when the item is not in the array, the correct insertion point will always be the value of `low`. An easy way to determine this would have been to place a statement in the code that displays the values of `low`, `mid`, and `high` before each of the `return` statements, to see what values those variables have at the end of the algorithm. This is commonly called "instrumentation code", and is often a very helpful problem-solving and debugging tool.

The greatest challenge, from a design aspect, was how to handle the case of the empty array. How you handle this depends on how you define what constitutes an "empty" array. Is an empty array an array that contains cells, each of which is empty (like an empty egg carton); or is an empty array an array that contains no cells? Arguments can be made for either case. Since I did not adequately define the what an empty array means in the assignment, I gave you credit for either definition. When an array is instantiated in Java using the `new` keyword (e.g., `Comparable[] theArray = new Comparable[10];`), and a dimension is provided, the array will always contain cells, and these cells will always be automatically populated by a set of default values (e.g., 0 for `int` arrays, `null` references for arrays of references to objects). The array therefore contains cells, but it contains no data items, where "data items" refers to items that have been explicitly added to the array programmatically. Thus, if you query for the length of an initialized array that you have not yet populated with actual data, the return value is the number of cells in the array. An array in Java does not have a convenient way to return whether or not it contains any actual data items. This leaves one of two options for determining whether or not an array created with the `new` keyword is empty:

1. *(Quicker, but buggy, option)* Assume the contents of one array cell, or a small subpopulations of array cells, indicate the state of the array. For arrays of type reference-to-object, which applies to this assignment, since the arrays you used should all have been of type reference-to-`Comparable`, an empty array will contain a `null` reference in every cell. In this case, you would look at the value at index 0, and if it is a `null` reference, assume the array is empty. This is a O(1) operation so long as the number of array cells examined is very

small compared to the total number of cells. The obvious pitfall to this approach is the possibility that one or more of the other array cells contain actual data items.

2. *(Slower, but more accurate, option)* Traverse the entire array, and if all the cells contain the default "empty array" value, assume the array is empty. This is a O(N) operation, so it is much slower, but it avoids the pitfall of the first option. It is not foolproof, however, since there is a possibility that the default "empty array" value may also happen to represent actual data. This is easier to see with primitive `int` arrays, since it would not be surprising to have a 0 as a data item. For arrays that contain references to objects, it usually isn't as likely for a `null` value to represent actual data, but it is possible, depending on the program design.

Ideally, the array would be able to distinguish whether or not it contains any items, but since arrays are rather primitive data structures, we're stuck dealing with them as they are. Later in the course we will talk about how to avoid this problem with more advanced data structures.

If, on the other hand, an array is created using the empty initializer (e.g., `Comparable theArray = {};` or `Comparable theArray = new Comparable[] {};`), the value returned by `theArray.length` will be 0, which is consistent with an empty array. This makes it very easy to determine emptiness in O(1) time; however, such an array is also 0-dimensional, which isn't really very useful. It's true that the array contains no data, but it also has no way of storing any data since it contains no cells. It's sort of like an empty box without the box.

The test arrays and search items I used are summarized in the following table:

Test Number	Condition Tested	Input	Expected Output
	Array contains an even number of items. Initial array: **`[ 3 ][ 5 ][ 7 ][ 9 ][ 11 ][ 13 ][ 15 ][ 17 ][ 19 ][ 21 ]`**		
1	non-existing item that should be added immediately before the first item	1	0
2	non-existing item that should be added immediately after the last item	30	10
3	non-existing item that should be added between the first and last items	12	5
4	existing item that should be added immediately before first item	3	0 or 1
5	existing item that should be added immediately after the last item	21	9 or 10
6	existing item that should be added between the first and last items	11	4 or 5
	Array contains an odd number of items. Initial array: **`[ 3 ][ 5 ][ 7 ][ 9 ][ 11 ][ 13 ][ 15 ][ 17 ][ 19 ][ 21 ][ 23 ]`**		
7	non-existing item that should be added immediately before the first item	1	0
8	non-existing item that should be added immediately after the last item	30	10
9	non-existing item that should be added	12	5

	between the first and last items		
10	existing item that should be added immediately before first item	3	0 or 1
11	existing item that should be added immediately after the last item	21	9 or 10
12	existing item that should be added between the first and last items	11	4 or 5
13	**Array is initially empty (actually, each index contains a null reference)**	11	0

[1]For those of you who altered the `binarySearch` method to accept primitive `int` values only, I altered my testing code to pass the array as an array of `int` values. For those of you who altered the method to accept `String` values only, I altered my testing code to pass the array as an array of `String` values (single digit values were given leading zeroes to ensure proper ordering).

Grading Rubric

The grading rubric I used is outlined in the table below. If you have any questions about your grade, please send me an e-mail.

	Points
Correctly passed test case #1	1
Correctly passed test case #2	1
Correctly passed test case #3	1
Correctly passed test case #4	1
Correctly passed test case #5	1
Correctly passed test case #6	1
Correctly passed test case #7	1
Correctly passed test case #8	1
Correctly passed test case #9	1
Correctly passed test case #10	1
Correctly passed test case #11	1
Correctly passed test case	1

#12	
Correctly passed test case #13	1
Submission of your own test cases	7
Total:	**20**