CSE 143, Winter 2012
Programming Assignment #2: SortedIntList (40 points)
Due Thursday, January 19, 2012, 11:30 PM

This program focuses on implementing a collection. Turn in two files named `SortedIntList.java` and `SortedIntListTest.java` from the Homework section of the course web site.

For this assignment you are to **write a class called `SortedIntList`** that is a variation of the `ArrayIntList` class written in lecture. Your class has two primary differences from the original list:

- A `SortedIntList` must maintain its list of integers in **sorted order** (non-decreasing).
- A `SortedIntList` has an option to specify that its elements should be unique (**no duplicates**).

The new class should have the same public methods as `ArrayIntList` except for the two-parameter `add` method that adds a value at a particular index. Because we want to keep the list in sorted order, we don't want to give clients the ability to decide where to insert a value, so this method should not be a public method of the new class. Some of the common methods between the two classes will need to be implemented differently in `SortedIntList`: `add` and `indexOf`. You will also add two constructors and the `getUnique`, `setUnique`, `max`, `min` and `count` methods.

`SortedIntList` should have the following public interface:

`public SortedIntList()`	constructs an empty list of a default capacity, allowing duplicates
`public SortedIntList(boolean unique)`	constructs empty list of default capacity and given "unique" setting
`public SortedIntList(int capacity)`	constructs an empty list with given capacity, allowing duplicates
`public SortedIntList( boolean unique, int capacity)`	constructs an empty list with given capacity and "unique" setting
`public void add(int value)`	possibly adds given value to list, maintaining sorted order
`public void remove(int index)`	removes value at given index, shifting subsequent values left
`public int get(int index)`	returns the element at the given index
`public int size()`	returns the number of elements in the list
`public boolean isEmpty()`	returns `true` if the list doesn't contain any elements
`public int indexOf(int value)`	same behavior as before, but optimized; see next page
`public int max()`	returns the maximum integer value stored in the list (throws a `NoSuchElementException` if the list is empty)
`public int min()`	returns the minimum integer value stored in the list (throws a `NoSuchElementException` if the list is empty)
`public int count(int value)`	returns the number of times the given value is in the list
`public boolean getUnique()`	returns whether only unique values are allowed in the list
`public void setUnique(boolean unique)`	sets whether only unique values are allowed in the list; if set to `true`, immediately removes any existing duplicates from the list
`public String toString()`	returns a string version of the list, such as `"[4, 5, 17]"`

Your class will have a field to keep track of whether or not it is limited to unique values. The value can be set by your second constructor or by calling `setUnique`. Think of it as an on/off switch for duplicates that each list has.

Implementation Details:

`public SortedIntList()`
This constructor should initialize a list of default capacity (10) with uniqueness set to `false` (duplicates allowed).

`public SortedIntList(boolean unique)`
This constructor should initialize a list of default capacity (10) with uniqueness set to the given value.

`public SortedIntList(int capacity)`
This constructor should initialize a list with the given capacity and with uniqueness set to `false` (duplicates allowed). If the capacity is negative, an `IllegalArgumentException` should occur.

`public SortedIntList(boolean unique, int capacity)`
This constructor should initialize a list with the given capacity and with the given setting for whether or not to limit the list to unique values (`true` means no duplicates, `false` means duplicates are allowed). See `get/setUnique` below. If the capacity is negative, an **`IllegalArgumentException`** should occur.

`public void add(int value)`
Your single-argument `add` method should ensure a sorted list. You should no longer add at the end of the list; instead you should add the value at an appropriate place to keep the list **in sorted order**. For example, if the list is [-3, 7, 18, 42] and the user adds 27, afterward the list should contain [-3, 7, 18, **27**, 42] in that order. (See `indexOf`.)

The `add` method has to pay attention to whether the client has requested **unique values only**. If so, your method must not allow any element to be added if that value is already in the list. You should not re-sort the entire list every time an element is added. Finding the correct index and inserting the element there is more efficient than re-sorting the whole list.

`public boolean getUnique()`
This method should return the current uniqueness setting (`true` means no duplicates, `false` means duplicates allowed).

`public void setUnique(boolean unique)`
Allows client to set whether to allow duplicates in the list (`true` means no duplicates, `false` means duplicates allowed).

If the unique switch is set to off (`false`), the list allows any integer to be added, even if that integer is already found in the list (a duplicate). If the unique switch is on (`true`), any call to `add` that passes a value already in the list has no effect. In other words, when the unique switch is `true`, the `add` method should not allow any duplicates to be added to the list. For example, if you start with an empty list that has the unique switch off, adding three 42s will generate the list [42, 42, 42]. But if your list has the unique switch on, adding those same three 42s would generate the single-element list [42].

If the client sets unique to `true` when the list has duplicates, `setUnique` should **remove all duplicates** and ensure that no future duplicates can be added unless the client sets unique back to `false`. If the client changes the unique setting to `false`, the list elements do not immediately change, but it will mean that duplicates could be added in the future.

`public int max()`
`public int min()`
These methods should return the largest and smallest element values contained in the list, respectively. For example, in the list [4, 4, 17, 39, 58], the max is 58 and the min is 4. If the list is empty, it has no largest or smallest element, so you should throw a **`NoSuchElementException`**.

`public int count(int value)`
This method returns the number of times the given value is in the list. If the list contains the values [-3, 1, 2, 2, 2, 5, 6], the call `count(2)` returns 3 because there are 3 occurrences of 2. Take advantage of the fact that the list is sorted. You should not traverse the entire list looking for elements equal to the parameter. Finding an occurrence of the value using binary search and counting elements of the same value around it is more efficient (see `indexOf`).

Note that binary search may not give you the index of the first occurrence of the value. For example, if the list were to contain [-3, 1, 2, 2, 2, 5, 6], searching for the number 2 would give the index 3 which is in the middle of a group of 2s. Your method will need to count occurrences of the value to both the left and right of the index given by binary search.

If the value is not in the list, your method should return 0.

`public int indexOf(int value)`

This method should take advantage of the fact that the list is sorted and use the faster **binary search** algorithm. If the element is found in the list, return its position. The element may occur in the list multiple times; if so, return any index at which that element value appears. If the element is not found, **return -1**.

Use the built-in `Arrays.binarySearch` method for all index location searching ("Where is a value currently located?" "Where should I insert a new value?"). You can find its documentation in the Java API from the Links section of the course web site. For example, to search indexes 0-16 of an array called `data` for values 42 and 66, you could write:

```
// index        0  1  2   3   4   5   6   7   8   9  10  11  12  13  14  15   16  17  18
int[] data = {-4, 2, 7, 10, 15, 20, 22, 25, 30, 36, 42, 50, 56, 68, 85, 92, 103,  0,  0};
int index1 = Arrays.binarySearch(data, 0, 17, 42);     // index1 is 10
int index2 = Arrays.binarySearch(data, 0, 17, 66);     // index2 is -14
```

When `Arrays.binarySearch` is unable to find a value in the list, it returns a negative number one less than the index at which the value *would* have been found if it had been in the array (sometimes called the **"insertion point"**). For example, in the example above the value 66 is not found, but if it had been in the list, it would have been at index 13; therefore the search returns -14. You should take advantage of this information when adding new elements to your list.

To get access to the `Arrays` class, you should `import java.util.*;` at the beginning of your class.

Testing Program (`SortedIntListTest.java`):

Along with your `SortedIntList`, turn in a short **testing file** named `SortedIntListTest.java`. This program should test your implementation of `SortedIntList` by creating at least two lists, adding some elements to them, calling various methods, and checking the expected results. This part of the assignment is worth only a few points, but we want you to practice testing your own code. For full credit, your testing file must generate at least 5 lines of output, and you must call at least 3 of the different methods on the list(s) you create.

There are also several provided testing programs on the course web site. You may ask, "Why do I need to write my own testing program when these testing programs are already provided?" Our testing programs are large and complex. It can be good to have a smaller test that runs just a few methods that you are working on. We encourage you to add to this program as you develop your `SortedIntList`. If you want to write a more complex test, that is fine, but not required.

Development Strategy:

We suggest that you develop the program in the following three stages:

1. Write a first version that always allows duplicates and doesn't worry at all about the issue of unique values. Just keep your list in sorted order and use binary search to speed up searching. This stage involves the following:
    a. Create your class file and the overall class header. You may want to start with a copy of the `ArrayIntList` code.
    b. Write the constructors.
    c. Make sure the two-argument `add` is not a public method.
    d. Modify the single-argument `add` method so that it preserves sorted order. You will need to use binary search here if your solution involves two steps (first locate, then insert).
    e. Modify `indexOf` so that it uses the binary search method; also write `max`, `min` and `count`.
2. Modify your code so it keeps track of whether the client wants only unique values. Add any state necessary and modify constructor(s) as needed. Next modify `add` so that it doesn't add duplicates if the unique setting is `true`.
3. Write the `getUnique` and `setUnique` methods. Remember that if the client calls `setUnique` and sets the value to `true`, you must remove any duplicates currently in the list.

We will provide testing code for each of these 3 stages. For this program only, you are allowed to discuss how to write testing code with other students. Keep in mind that our tests are not guaranteed to be exhaustive and we do not guarantee full credit for passing the provided tests. Use them as examples of the kind of testing code we want you to write. You will want to do a lot of simpler testing before you try running any of these high-powered tests.

References:

Textbook Chapter 15 covers the implementation of `ArrayIntList` in detail.

Textbook sections 13.1 and 13.3 discuss the binary search algorithm in more detail. 13.3 discusses how it is implemented and its return values in detail. Seeing how it is implemented may help you understand how to use it appropriately.

Style Guidelines and Grading:

You may not use any features from Java's collection framework on this assignment, such as `ArrayList` or other pre-existing collection classes. You also may not use the `Arrays.sort` method to sort your list.

A major focus of our style grading is **redundancy**. As much as possible you should avoid redundancy and repeated logic within your code, such as by factoring out common code from `if/else` statements, creating "helper" methods to capture repeated code, or having some of your methods/constructors call others if their behaviors are related. Any additional methods you add to `SortedIntList` beyond those specified should be `private` so that outside code cannot call them. Note that even constructors can be redundant; if two or more constructors in your class contain similar or identical code, you should find a way to reduce this redundancy by making them call each other as appropriate.

Properly **encapsulate** your objects by making any data fields in your class `private`. Avoid unnecessary fields; use fields to store important data of your objects but not to store temporary values only used within a single call to one method. Fields should always be initialized inside a constructor or method, never at declaration.

You should follow good **general style** guidelines such as: appropriately using control structures like loops and `if/else` statements; avoiding redundancy using techniques such as methods, loops, and `if/else` factoring; properly using indentation, good variable names, and proper types; and not having any lines of code longer than 100 characters.

Commenting will be more of a style focus on this program (and future programs) than it was on the last assignment. You should comment your code with a heading at the top of your class with your name, section, and a description of the overall program. Also place a comment heading atop each method, and a comment on any complex sections of your code.

Not only will we look for the existence of comments on this assignment, but we also have stricter expectations about the quality of their content. Comment headings should use descriptive complete sentences and should be written in your own words, explaining each method's behavior, parameters, return values, and pre/post-conditions as appropriate. If the method potentially throws any exceptions, comment this and explain what exceptions it throws and under what conditions it will throw them. Be concise but specific in your comments. The `ArrayIntList` class, along with other programs from lecture and section this week, are good examples of appropriate commenting style for this assignment. (Your test program should have an overall descriptive comment header at the top of the class, but it does not need commenting on each testing method or otherwise throughout the code.)

The provided `ArrayIntList` file already has comments on its methods. If those comments are still correct and accurate for describing the behavior of the `SortedIntList`, you may re-use them. But several aspects of the behavior, constraints, preconditions, etc. of the `SortedIntList` are different than the behavior of the `ArrayIntList`. Any `ArrayIntList` comments that do not apply to your new class should be removed and any important aspects of the `SortedIntList` not described in the `ArrayIntList` comments should be document by new comments of your own

For reference, our `SortedIntList.java` is around **180 lines long** including comments (and has **128 "substantive" lines** according to our Indenter tool on the course web site). But you do not have to match this; it's just listed as a sanity check.