COSC 1P03 Lab 5
Mar. 24-28, 2014

Exercise 1 (and only)
A Trivial ADT (Sorted Arrays)

Estimated Time: The Lab

For this lab, we're going to be using a very simple data structure: a *sorted array*.
Rather than having multiple independent exercises, this lab instead deals with it as a single exercise, with a few *checkpoints* along the way.

Step 1
For convenience, we aren't even going to deal with an interface (note: strictly speaking, this is not just inappropriate for an ADT; it's outright wrong. However, the KISS principle trumps for educational purposes here). Instead, simply download the skeleton code provided on the course website.

A *sorted array* is simply a type built around the idea of an array, but *sorted* (I'm hoping you could see that coming). The ADT can be used for different types of elements, but they all have the same property in common: They must be *Keyed* (i.e. the same Keyed as in the Chapter 19 code). Because a Keyed element can give a key, and because that key is a String, and because Strings are Comparable, we can thus compare the relative ordering of any two elements.

So, why don't we just use Comparable instead of Keyed? Because bounded parametric types are already enough of a nuisance.

As mentioned above, you should have downloaded the included source files. They don't *work*, but they do *compile*. Note that there's also minimal commenting added to provide clues on what to add.

Assuming you have all of the files, and arranged appropriately in your IDE, you should be able to compile everything. You won't be able to *test* it yet, as there's no implementation (and the for-each loops won't actually work with a null iterator).

Ensure that you can compile before continuing.

Step 2
First, let's just deal with the public constructors. Technically, one is already taken care of for you (the 'default' constructor that simply chains to the 'real' one), so we just have to figure out how to create the array.

By now, you should be comfortable with creating a generic array.
If not, remember the key points:

- You are actually creating an array of Keyed type
- You need to force it into believing it's an array of type E

Ensure that you actually have an instance variable to receive the array, of course.
Next, you should have an instance variable keeping track of how much of the array is being used, which should be initialized in the constructor.

At this point, try compiling. It should compile just fine, *except* it should throw a *warning*.

You can easily get around this by adding the following line before the constructor:
`@SuppressWarnings("unchecked")`

Note that this doesn't actually fix the concern (that generic arrays are often frowned upon); it simply tells the compiler to stop complaining about it.

By this point, you should have your instance variables (including the array), and your public constructor should be done. It should compile without any *warnings*.

Step 3
Next, we're going to deal with the actual addition of elements.
You'll have to figure out how to do this yourself, but here are a few hints:

- Since you have an upper limit on capacity, remember to check for overflow
- There's a special case for insertion (when it starts off empty)
- Otherwise, you're looking for the first item in the array that has an equal priority, or belongs after the item being added
  - Remember that you're comparing the *keys* (i.e. Strings); not the elements themselves!
- Once you've found where the new item should go, simply shift everything from that element to the end back by one
  - Note: by “to the end”, that means, “to the last stored item”; not, “to the end of the array”
  - The easiest way to do this loop is to start at the end, and decrement the counter towards the insertion point

While you're at it, also fill in the `getSize()` function, since it's pretty darn easy.

By this point, you should be able to add items to your array. Try compiling to preclude any obvious errors.

Step 4
We're not even close to done yet, but we have two issues:

1. Being able to compile doesn't verify that our algorithm works
2. There's no point in writing/testing the iterator before we know it won't explode the first time it tries doing a real insert!

This means we need to try running our (very simple) test harness.
It *will* crash, but it should crash at line 16, and shouldn't complain about the *add* method at all. If you've introduced a new bug, you'll want to fix that before continuing. The easiest mistake to make is to get the index wrong for one of the loops.

I'm assuming you'll have it working before you continue to the next step: writing the iterator.

There are two basic places to write the iterator:

- In a separate file
  - Probably *package-private*
  - Make sure to give it the same package name as `SortedArray`!
- As a private class within `SortedArray`
  - I don't know why you'd want to do this

Either way, the code will be the same.

- Remember to have the class implement the Iterator interface
- Be careful with your generics
- Remember to adjust the iterator() function so it creates a new instance of the iterator

- Remember that the iterator needs a reference to the actual data structure (i.e. the array itself), and needs to know how much of the array is being used

Getting the header and generics right is easily the hardest part of this task. The iterator itself is simply traversing across part of a 'variable-sized' array, after all.

By this point, you should be able to run the test program as it was provided to you, and should have verified that your code thus far works (outside the 'merge').

Step 5
There's only one part of the ADT that's still unwritten: merge
Some tips:

- You already know how many items are currently held within the data structure (you should have an instance variable for that)
- You should have an accessor function that lets you know how many items are held in the 'other' structure
- You should be able to create a temporary array that's the perfect size to hold the elements from both structures
- Your iterator should let you copy the elements from each structure (the 'this' structure, and the 'other' structure) into said array
- You should use a *private* constructor to handle transferring the contents of that array into a new SortedArray, including the sorting

Private arrays are actually pretty simple: they're good for when you want a 'special case' constructor for a specific use, but wouldn't want a client program to use it.
In this case, we're assuming that a constructor can:

- Receive a right-sized array of elements
- Allocate sufficient space for those elements, with adequate space for growth
  - Assume the SortedArray starts half-full. i.e. take the size of the provided array, and use double its length for the new one
- Use the *add* method to add each element from the provided array into the new structure

Tip: If you haven't picked up on this, you should be using 'this' to first call the constructor you've already written (to allocate space), before writing the additional lines that make this constructor special.

By this point, you should be able to use the merge function, tested in the test harness.

Step 6
If you've gotten this far, then your final task should seem incredibly simple: Currently, you've only used `KeyedChars`. Let's create a `KeyedInt` class, and uncomment the rest of the testing code to try it out!

Once you have that working, and the entire code running, show your TA. However, it's suggested that you not stop there (if you still have time).

The existing test harness is incredibly lacking. You should try expanding it. Can you break your own code?

Also, the last line of the test harness is 'not legal'. Uncommenting it should prevent the program from compiling. Can you see why?