COSC 2P03 – Assignment 3 – In for a heap of trouble!

Objective:

Didn't y'all just *love* how the first assignment had multiple things to work on independently?

Yes?

Oh, I'd actually assumed you wouldn't have, but if you did, then I guess we can do that again!
This time, we're going to have fun with heaps!

Specifically, we'll focus on two tools, *and* three separate tasks!

- Your submission needs two different implementations of an Indexed Heap, both of which adhere *perfectly* to its precise specification/interface
    - Actually, both the interface and one of the implementations is already provided to you!
    - The one you'll be writing (`IndexedHeapTree`) is as an actual *linked tree*, complete with Nodes (with only left/right/item)
    - An `IndexedHeapTree` is simply one that can return a requested element by **tree index**
        - e.g. if you request for index 1, you get the root's value; 3 is the root's right child, etc.
        - Note that these do not modify the tree; they're effectively just *peek*s
    - The only parameter your tree-based heap should need is its `Comparator`
- The first task is a basic test program
    - I've provided *most* of what you need for this. It's not actually worth anything in the marking scheme so you *could* skip this (well, technically you could just submit what I provided for you), but the reason it's mentioned is that you need to ensure your implementation *perfectly* matches the behaviour of the one that's provided (other than the tree not being able to overflow, obviously)
- The second task is to assist with sorting
    - Write a program that can use the heaps to assist with sorting integers or Strings:
        - Randomly-selected integers (the user chooses how many to generate)
        - User-entered Strings (accept a single line of text, then use .split() to break it up into separate tokens)
    - Use Scanner on System.in for input
    - Let the user decide whether 'to try again' or quit
    - For both numbers and Strings, let the user choose which heap to use
    - The mechanism by which they are sorted is as follows:
        - Write a method that accepts an array of entries and a **max**-heap
        - Iterate through the array and insert its contents into the **max**-heap
        - Dump the **max**-heap back into the same array
        - I trust you can figure out what's actually required here? Yes? Good!
    - Note: you're doing this as a max-heap, or "you didn't do it"
- The third task is something neat!

Something neat?

The third task is about **authentication**. Everyone's familiar with *passwords*. And by now we've all had to deal with alternate authentication forms like SMS and/or code generation (not going to get into the specifics of RSA, etc. here). But that's not the only way to do it!

You might be familiar with One-Time Passwords (OTP): a super-secret you can use precisely *once* to prove you are who you say you are. The idea is you write it down somewhere safe, and it's a fallback for your regular password. But there are also **passcode grids**, like what the CRA uses. These act as a list of assigned

passwords, that are indexed. e.g. you might be prompted with "B6", and need to align column B and Row 6 to get a three-character string of text. We'll do something similar to that: you'll assume two parties each have access to the same specialized indexed data structure, and can request its contents based on some single index. When both parties can confirm they 'know' the same token, they can trust each other!

To clarify: you're not actually handling both parties; the fact that there would be two is simply why it matters that the structure *always* be consistent and predictable: if not, then you're a stinky impostor and nobody should trust you.

In this case, the data structure will simply be two of your **Indexed Heaps** encapsulated together!

- Your class is to be called `Authenticator`, and will contain two IndexedHeaps
  - Both will hold Strings
  - One will be the min- version, and the second will be the max- version
- The authenticator has a single function, `getCode`, that accepts an index value, and returns the *concatenation* of the result of requesting for that index from the Indexed MinHeap and the result for the same index from the Indexed MaxHeap
- Note: nothing outside of the Authenticator ever *directly* interacts with the heaps
- Use the IndexedHeapTree you implemented in the other task for this (for both)
- The contents of the tree are initialized in the Authenticator's constructor:
  - It receives a single argument: an array of Strings
  - It simply inserts those Strings, in the appropriate sequence, into both heaps

Of course, you'll again need a basic program to drive this. As before, use Scanner on System.in.

Let the user enter as many codewords as desired, for initializing/populating the Authenticator; followed by allowing the user to provide as many numbers as desired (displaying the confirmation tokens each time).

Additional requirements and tips:

- Both because your heap is generic, and because you'll want to be able to use the same implementation as a MinHeap or a MaxHeap, you'll also be using Comparators
- Adhere to specifications. In this case, it should be abundantly clear why this is mandatory
- Once again, there are both multiple packages, and multiple main classes
  - By this point, you're expected to know how to work out 'what goes where'
- To confirm: beyond the sample code I've provided, **you're implementing everything yourself**
  - If you're unsure of something: ask
  - My total imports are Scanner, Iterator, Comparator, and obviously my own packages written for these tasks. If you're using something else, you better have checked with your instructor
    - Obviously this doesn't mean you can bypass `import` statements by including paths

Submission:

This course (and by extension, this assignment) uses IntelliJ. Do not use any other IDE.

- As a clarification, you aren't writing in something else then importing into IntelliJ. You aren't expecting the marker to fix your project for you. You're including a run *configuration* that explicitly includes the main class
- Don't try clumsily importing a solution into an IntelliJ project, resulting in multiple nested folders. It's confusing and won't be graded
- (If you have any questions, don't forget you have, like, an instructor you can ask?)

You are expected to adhere to certain minimal standards *before* true evaluation of your work even starts:

- Comment your code
  - Comment what it *does*; not what it *is*!

    - e.g. indicating that a ‘for loop’ is ‘a loop’ is worse than nothing. Don’t do that
- Document your code
    - No, this is not the same thing
- Follow basic standards
    - Avoiding hardcoded literals
    - Obviously nothing ‘static’ other than the main method of the main class
        - Ask first before attempting an exception, or expect a zero
    - (You get the idea!)
- Reminder: beyond what’s allowed above, no `import`s. At all. Seriously.
- Useful variable names. Organized. Indented correctly.
    - i.e. your submission will be readable, or you did not submit

Include a sample output, along with your entire IntelliJ project and summary.
Bundle/compress everything into a single .zip file.

- Not a .rar
- Not a .7z
- Not a .tar.gz
- Please don’t decide to earn a zero from something so trivial

Submit your .zip in Sakai