Assignment #3 - Heap, Priority Queues and Adaptable Priority Queue

This assignment is not easy. It requires you understand the related data structures, the algorithms, and it requires strong OO design. You should read through the assignment to get a basic idea of the requirements and then start working through the assignment in the recommended order.

Objectives:

The main objectives of this assignment are:

1. Implement Priority Queue with heap, which is then implemented using ArrayList.
2. Implement Adaptable Priority Queue with location aware design
3. Annotate the running time complexity with `@TimeComplexity` annotation and `@TimeComplexityAmortized` annotation. See section "Annotation".
4. Write justification comments. See section "TCJ".

Your PQ implementation will be used in 3 future assignments. So please make sure you implement all the methods correctly.

The Assignment:

You are to implement Adaptable Priority Queues with heap data structure.

1. Implement the heap using the **`ArrayList`** data structure, which is the one that you did in assignment1.
2. There are four constructors in `HeapAPQ.java`. When no comparator is specified, you should instantiate and use the default comparator **`DefaultComparator`**, which is a class that has been provided to you as an embedded class.
3. When you implement Entry<K,V>, you need to make it **location aware**, which means you need to keep the index in the entry object data. Without the location aware design, the method `replacekey()` and `remove`() will **not** be efficient.

Getting Started:

1. Download and import the zip file.

    1) Open Eclipse
    2) Choose "File → Import"
    3) Select the "General" category
    4) Pick "Existing Projects into Workspace"

5) Click on "Next"
6) Select the button for "Select Archive File" and browse to/select the zip file.
7) Click on "Finish"

A new project should be added to the workspace: `HeapAPQ`, which provides `HeapAPQ.java`, and sample but incomplete JUnit test files `HeapAPQTest.java`. The `net.datastructures` package contains all of the interfaces needed for this assignment (including `Entry`, `PriorityQueue`).

2. **Copy** the `ArrayList` implementation and any additional required files you used for Assignment 1 into your cs2321 directory for the project and use it to build the Heap.
3. A special method Method data(). This method returns the internal array, so we can easily examine if the data has been stored correctly in the right order.

```
/*
 * Return the array in arraylist that is used to store entries
 * This method is purely for testing purpose of auto-grader
 */
public Object[] data() {
        //TODO: replace the line below to return the actual array in arraylist
        //TODO: You may want to add a method in your arrayList implementation
        //TODO: to allow the access to the array.

        return  null;

}
```

4. Some test cases has been provided to you under tests folder. Study them and add more if you see some cases are not being tested.

5. Implement HeapAPQ.java.
   - Please write comments at the same time when you write the code.
   - Please implement your method in an order where unit test can be performed immediately.

Annotate the running time Complexity and Write time complexity Justification. See section " Running Time Complexity Annotation" below for details.

Submission:

First, look at the following checklist:

1. Do you ever `import` from `java.util`? If so, be sure you only import allowed components (like `Iterator`, Comparator, Exceptions, etc.). Unless the assignment specifically mentions it is permissible, you should never include any of java's native data structures.

2. Did you annotate all the methods with `@TimeComplexity` (and `@TimeComplexityAmortized` if it applies) ?

   Important methods:

   insert(): worst case and amortized
   removeMin(): worst case
   remove(): worst case
   replaceKey(): worst case

3. Do you provide adequate `TCJ` comments to prove your time complexity?
4. Is the indentation easily readable? You can have Eclipse correct indentation by highlighting all code and select "Source → Correct Indentation".
5. Are comments well organized and concise?

If you have reviewed your code and all the criteria on the checklist are acceptable, follow the submission procedure export to prog3.zip and upload to canvas.

Grading Criteria:

- Correctly implement all methods and interfaces: 80
- Correct documentation of time complexity: 10
- Clear concise code with good commenting: 6
- Running Time complexity justification: 4
- Location aware implementation: 5

Running Time Complexity Annotation

In this assignment, we ask you to use Java' annotation to specify the running time complexity. We implemented several annotation interfaces. Each annotation interface takes a string as it's argument.

- `TimeComplexity`
- `TimeComplexityAmmortized`
- `TimeComplexityExpected`

Here is the java code for TimeComplexity Annotatation.

```
package cs2321;

import java.lang.annotation.*;

/**
 * Claim a method's worst case time requirements.
 *
 * Valid values are:
```

```
 *     O(1), O(n), O(n^2), O(n^3), O(lg n), O(n lg n), O(n^2 lg n), O(2^n), O(?)

 *     O(?) is used for methods whose complexity isn't relevant.
 *          (Such as "main" methods used for testing purposes)
 *
 * Example syntax:
 *    @TimeComplexity("O(n)")
 *    METHOD_DECLARATION
 */
@Retention(RetentionPolicy.RUNTIME)
public @interface TimeComplexity {
    String value()  default "[unassigned]";
}
```

To use the annotations, simply add the correct annotation above the method. For example, if the worst case running time for method is O(n), then you will add a line above your method declaration. Like this:

```
@TimeComplexity("O(n)")
PUBLIC VOID METHOD_A (…) {

}
```

For some methods, it is worthy to specify the amortized or expected upper bound if they are different with the worst case upper bound, then you can have multiple annotations like this.

```
@TimeComplexity("O(n)")
@TimeComplexityAmmorized("O(1)")
PUBLIC VOID METHOD_A (…) {

}
```

Be sure that each of your annotations is **valid string.** Here are some examples: "O(1)", "O(n)", "O(n^2)", "O(n^3)", "O(lg n)", "O(n lg n)", "O(n^2 lg n)", "O(2^n)" and "O(n+m)".

Each of your method should have at least one of the Time Complexity annotations for each method. You will need to specify variable represent in the time justification comments.

TCJ (Time Complexity Justification) comments

Each non-trivial (with loops and function calls) method should include special comment blocks that begin with the letters TCJ" that justify the time cost (TCJ=Time Cost Justification)

```
/**
 * Removes and returns the element at the given index, shifting all subsequent
 * elements in the list one position closer to the front.
 * @param  i   the index of the element to be removed
 * @return the element that had be stored at the given index
 * @throws IndexOutOfBoundsException if the index is negative or greater than size()
 */
    @TimeComplexity("O(n)")
    public E remove(int i) throws IndexOutOfBoundsException {
           /* TCJ
            * All the elements starting from index i+1 to the last one [size-1]
            * have to shift to its left. The number of shifting is size-i.
            * At worst case when i=0, there are size shifts.
            * The function call checkIndex() is O(1)
            * The worst case of remove method is O(n) where n is the data size
            */

           checkIndex(i,size-1);

           E old = data[i];
           //move element from [i+1 .. size-1] to its left spot
           for (int k = i; k <= size-2; k++) {
                  data[k]=data[k+1];
           }
           size --;
           return old;
    }
```

FAQ

Q: How to compare two generic type objects E1 and E2?
A: Use comparator.compare(E1,E2). See the class file DefaultComparator.java

Q: When to throw IllegalArgumentExeption for the insert(K,V)?
A: When the comparator can not compare the keys.

Q: If we need to compare between keys, should we change the class from HeapAPQ< K,V> to be HeapAPQ<K extends Comparable<K>, V> ?
A: No. Because we want to define the way that we will be comparing data in different situation, not limited to the compareTo method associated with the key type.