University of Puerto Rico – Mayagüez
Department of Computer Science and Engineering

CIIC 4020 / ICOM 4035 – Data Structures
Prof. Juan O. López Gerena
Spring 2019-2020
Laboratory #9 – Priority Queues and Heaps

Introduction

In this lab, we will work with the `PriorityQueue` ADT, and with different implementations that have been introduced in lectures. We start by reviewing some relevant concepts about the `PriorityQueue` ADT; at the end of this document we have included a discussion about complete binary trees and heaps.

The priority queue is a collection of elements represented by what we refer to as entries. Recall that an entry is a special type of object that associates two fields, a key field and a data value field, and it is specified as follows:

```
/** ADT specification of Entry. An entry is a pair of fields:
    one representing a key (of type K) and one representing a data value
    (of type V) that is associated with the particular key.
**/
public interface Entry<K, V> {
      /** Accesses the key of the entry.
          @return reference to the key of the entry.
      **/
      K getKey();

      /** Accesses the value of the entry.
          @return reference to the value of the entry.
      **/
      V getValue();
}
```

The keys will be compared using a total order relation established by a comparator associated with the specific keys of a class instance. No assumption is imposed on the values that are associated with keys in a given priority queue (the other component of each entry). Based on that order relation of keys, the smaller the value of the key, the higher is the priority of an entry.

This type of collection is very useful for applications in which prioritized access is relevant. Common service-oriented systems usually use priority queues when deciding what request to serve next whenever a large number of requests are waiting for attention by the system and some may have higher urgency than others. For example, network routers, operating systems, projectile defense systems, communication systems, etc. In addition, several important algorithms use a priority queue as a fundamental data structure, such as finding a minimum spanning tree, and finding the shortest paths in a graph, etc. They are also very useful in simulation of systems based on discrete events.

In this lab, we shall work with different implementations of the `PriorityQueue` ADT, specified as follows:

```
public interface PriorityQueue<K, V> {
      /** The size of the collection **/
      int size();

      /** Returns true if empty; false, otherwise. **/
      boolean isEmpty();

      /** Add a new entry (key, value) to the queue.
          @param key the key for the entry
          @param value the value for the entry
          @return Reference to the newly-created Entry object
          @throw IllegalArgumentException if key parameter is invalid
      **/
      Entry<K, V> insert(K key, V value) throws IllegalArgumentException;

      /** Accesses entry in the collection having highest priority
          (or minimum key, according to a particular order relation)
          @return Reference to the entry with minimum key, or null if empty
      **/
      Entry<K, V> min();

      /** Removes entry in the collection having highest priority
         (or minimum key, according to a particular order relation)
          @return Reference to the removed entry, or null if empty
      **/
      Entry<K, V> removeMin();
}
```

About this Lab

You have received a Java project that contains some partial implementations of PriorityQueue as specified in lectures. They are based on a hierarchy of classes to achieve a good level of code reusability on those different implementations. There are some minor differences from what was discussed in the lectures; some of them, just for the purpose of this lab and for testing purposes. These are:

1. A new version of priority queue is introduced; this is given as the Java interface `DisplayablePriorityQueue`. The only purpose of this is to allow us to display the priority queue's content. Notice that it is an interface that extends the `PriorityQueue` interface, and that this new interface only specifies a new method: **`display()`**. The purpose of this method is to display the current content (entries) in the priority queue and hence allow us to see what is inside it. The order of how the elements are displayed, or the structure of the displayed elements, will depend on particular final implementation details. Therefore, different implementations may need to implement their own display method.

2. The `AbstractPriorityQueue` class now implements `DisplayablePriorityQueue`. Also, in that class, the `EntryComparator` class has been defined, which compares entries by using a key comparator received by the constructor (that's right, a comparator that uses another comparator; it's things like this that make programming so much fun!).
3. The implementations based on lists will use `ArrayList` and, in order to achieve more code reusability, we have introduced the abstract class `AbstractListPriorityQueue`. That abstract class implements some methods that apply to all subclasses that will eventually extend it with the objective to provide a full implementation of `PriorityQueue`. There are three different implementations based on lists, these are: `UnsortedListPriorityQueue`, `SortedListPriorityQueue`, and `HeapListPriorityQueue`. You are expected to understand the concepts behind these implementations as discussed in lectures, including the *heap-based* version as implemented in the Java class `HeapListPriorityQueue`. NOTE: You can find a general discussion about complete binary trees and heaps at the end of this document.
4. Class `HeapPriorityQueue<K,V>` is another implementation of the `PriorityQueue<K,V>` ADT that is also based on the heap data structure. Notice that it has an internal field of type `Heap<Entry<K, V>>` (more details about the Heap class are provided further below). This class is introduced here for the purpose of visualization of the tree structure when implementing the priority queue using the complete binary tree approach. With this class, we can take advantage of the operation to display the content of a tree as seen in earlier lab activities. It is useful for learning purposes but perhaps not as a real implementation of the priority queue, since it has more overhead than the `HeapListPriorityQueue` (although asymptotically the execution times should be the same).
5. To support its implementation, two other classes have been introduced to implement the heap as a particular type of `BinaryTree`, hence taking advantage of the abstract implementation of the `BinaryTree`. These are the two Java classes: `CompleteBinaryTree`, which is a subclass of `AbstractBinaryTree`, and the `Heap` class, which is a subclass of `CompleteBinaryTree` that implements the min-heap ADT (a complete binary tree satisfying the heap property - see below).
   a) Class `CompleteBinaryTree` is a `BinaryTree` which is *complete*. As discussed in lectures, we know that it can be efficiently represented using an array (or an `ArrayList`). Therefore, positions are not nodes as were in class `LinkedBinaryTree` (used in a previous lab). Instead, we implement Position using an internal class: `CBTPosition<E>` (complete binary tree position), and the internal array of the `CompleteBinaryTree` class is declared as `ArrayList<CBTPosition<E>>`. Such positions have two internal fields: element and index; a partial implementation is included below. The element is as usual (to hold the element at the position), but the index is needed here because the positions are stored in the internal list (array or index list), and also several operations of the `BinaryTree` require reference to a position, so any position object used here needs to know where in the list it is located. That is precisely the purpose of that index.

```
protected static class CBTPosition<E> implements Position<E> {
      private E element;  // the element at this position
      private int index;  // index of position in the array list

      public CBTPosition(E element, int index) {
            this.element = element;
```

```
                this.index = index;
            }
            ... getters and setters....
    }
```

See for example the implementation of **`parent()`** in class CompleteBinaryTree.

Also notice that this class has an **`add`** operation. It adds a new element to the tree by just adding another node with the new element at the end of the tree. That only requires adding the element to the end of the internal list (the `ArrayList`).

b) Class `Heap` basically has the same operations as the `PriorityQueue` ADT, but it does not work with key-value pairs, it works with values directly (which could be entries). The values are assumed to be comparable based on an order relation that is defined by a Comparator object; that is particularly important for the result of the **`min()`** and **`removeMin()`** operations. The Heap is a complete binary tree that holds data in its nodes that satisfy the **heap property**: *that every position (different from the root) in the tree stores a value which is "greater or equal" to the value stored at its parent position*. Remember that such property guarantees that the value at the root position is always a min value. This is also called the *min-heap property*.

Now, if you go to the `Heap` class (which is a subclass of `CompleteBinaryTree`), there you will the public operations: **`min()`** and **`removeMin()`**. The first one accesses the element in the heap with minimum value and the second removes it from the heap. Both return a reference to that min value, or return null if the heap is empty. Also notice that `CompleteBinaryTree`'s **`add`** operation is overriden, with the goal of properly placing the new element inside the heap (not just at the end of the complete binary tree), hence guaranteeing that the internal structure continues satisfying the heap property.

The partial implementation of the `Heap` class also includes a partial implementation of **`downHeap()`** and **`upHeap()`**, which are internal auxiliary methods for the operations **`add()`** and **`removeMin()`** inside this class. These auxiliary methods will make use of another internal method, **`swapPositionsInList()`**, as needed by their respective algorithms. Notice that when a position is moved to another location, the internal index of the position needs to be modified accordingly.

```
/**  Interchanges two position in the array.
* @param r one of the position
* @param c the other position
*/
private void swapPositionsInList(CBTPosition<E> r, CBTPosition<E> c)
{
     int ir = r.getIndex();
     int ic = c.getIndex();
     r.setIndex(ic);   // when positions change, so do the indexes
     c.setIndex(ir);

     // swap elements of location ir and ic in the arraylist
     list.set(ir, list.set(ic, r));
}
```

Import the partial project provided as an Eclipse project in your system. Notice that this partial implementation includes classes for trees; we will use them here as described earlier. Also, in package `testerClasses` there are two particular tester classes that we will use in the following exercises. Those are: `HeapTester1` and `PriorityQueueTester1`.

Exercise 1: `downHeap`

Go inside the class `Heap`, which is in package `heap`, and look for the method **`downHeap`**. In particular, look for the comment stating that code is missing. Add the missing code (see idea for this algorithm in the discussion at the end of this document or in the lecture handouts), and test the code by running the `HeapTester1` tester. If correct, you should see the output shown in `outputFile1.txt`.

Exercise 2: max-heap

In package `testerClasses` add a new comparator class named `IntegerComparator2`, inverting the comparison from IntegerComparator1. That class should cause that if you run the same program again (`HeapTester1`), but changing the 1 for a 2 in line `**`, the output will be as in file `outputFile2.txt`. The heap now behaves as a max-heap!

If you take a look inside `PriorityQueueTester1`, you should see the following four lines:

```
DisplayablePriorityQueue<Integer, String> pq = new UnsortedListPriorityQueue<>(new IntegerComparator1());   //1
//DisplayablePriorityQueue<Integer, String> pq = new SortedListPriorityQueue<>(new IntegerComparator1());     //2
//DisplayablePriorityQueue<Integer, String> pq = new HeapListPriorityQueue<>(new IntegerComparator1());       //3
//DisplayablePriorityQueue<Integer, String> pq = new HeapPriorityQueue<>(new IntegerComparator1());           //4
```

In the following exercises, we will refer to them as Line 1, Line 2, Line 3, and Line 4, based on the number at the right side. Only one of those should be uncommented at any moment.

Exercise 3: Fix error #1

Run the class program `PriorityQueueTester1` (make sure that, of the above four lines, the only uncommented one is Line 1). The program shows the following output:

```
PQ content after adding entry: key = 20 and value = twenty
[20, twenty]

PQ content after removing highest priority element [20, twenty]
EMPTY
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
        at java.util.ArrayList.rangeCheck(ArrayList.java:653)
        at java.util.ArrayList.remove(ArrayList.java:492)
        at priorityQueue.AbstractListPriorityQueue.removeMin(AbstractListPriorityQueue.java:66)
        at testerClasses.PriorityQueueTester1.removeMin(PriorityQueueTester1.java:65)
        at testerClasses.PriorityQueueTester1.main(PriorityQueueTester1.java:27)
```

Discover what the error is. Once you correct it, the output produced should be as in file `outputFile3.txt`.

Exercise 4: Change the Comparator

Change the `IntegerComparator1` for `IntegerComparator2` in class `PriorityQueueTester1`. Run again this program, and compare its output with the output on the previous exercise. Analyze the results and make sure they are as expected.

Exercise 5: Fix error #2

Remove the comment characters (//) from line 2 inside `PriorityQueueTester1` (make sure that is the only one that is uncommented out of the four lines). Execute the program and you will see the following error message:

```
PQ content after adding entry: key = 20 and value = twenty
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: -1, Size: 0
	at java.util.ArrayList.rangeCheckForAdd(ArrayList.java:661)
	at java.util.ArrayList.add(ArrayList.java:473)
	at priorityQueue.SortedListPriorityQueue.insert(SortedListPriorityQueue.java:32)
	at testerClasses.PriorityQueueTester1.add(PriorityQueueTester1.java:61)
	at testerClasses.PriorityQueueTester1.main(PriorityQueueTester1.java:26)
```

Discover what the error is and correct it. Run the program again, and the output should be as in file `outputFile4.txt`. Once that is done, repeat execution for the last two lines; one by one, uncomment one line while commenting the other three. Execute the program in each case. The results should be the same (as far as the order in which the elements are added and removed).

Exercise 6: Change the Comparator, again

Run the previous program again, but using IntegerComparator2 instead of IntegerComparator1 for the last three lines. Analyze the results and make sure they are as expected.

Exercise 7: `what`

Consider class program `PriorityQueueTester2`. It invokes static methods **`displayArray`** and **`what`**, which are inside class `TesterUtils`. Inside method **`what`**, two lines are missing. The lines are meant for the program `PriorityQueueTester2` to produce output as shown in file `outputFile5.txt`. You cannot alter any other part of that class; just add the two correct lines.

Exercise 8: Change the Comparator, yet again

Run the previous program again, but instead of using `IntegerComparator1`, use comparator `IntegerComparator2`. Analyze the results and make sure they are as expected.

Exercise 9: New tester and Comparator

Write a new tester similar to `PriorityQueueTester2`, named `PriorityQueueTester3`, which uses the following array instead:

```
String[] arr = {"barrio", "pepe", "julia", "maria", "oliva", "meme", "parada", "baile", "enjendro",
                "vagabundo", "nota", "tienda", "zapato", "caballo", "cafe", "diodo", "multiplica"};
```

The last three lines have to be exactly as in `PriorityQueueTester2`. Test with the different implementations of priority queue being considered here and with the following comparator (which you must create):

```
public class StringComparator1 implements Comparator<String> {
    public int compare(String s1, String s2) {
       return s2.compareTo(s1);
    }
}
```

The output should be as shown in file `outputFile6.txt`.

End of Exercises

General Discussion About Complete Binary Trees and the Heap

Coming soon…