**Priority Queue API**

A ***Priority Queue*** is exactly as it sounds: a FIFO, queue-type data structure, except that objects are ordered inside the queue based on their priority (rather than the order in which they were added). The item at the head of the queue will always be the highest priority element (given however priority is defined for the type of objects it contains), which can be accessed using queue's poll method. Elements in a priority queue must be *comparable* (they must implement the Comparable interface).

In the next lab we'll learn how priority queues work internally; in this lab, you'll use Java's implementation to solve some problems.

1. Solve the UIL written-test style problems in the PDF in the lab folder. Solutions below in white (download this document and change font color to view).

Question 1: B. cd

Question 2: A. ab

Question 3: B. implements Comparable

Question 4: A. 8

Question 5: C. 7

1. Solve the contest-type problems found in the lab folder; PriorityQueue objects will of course help (in the java.util package, part of the Java Collections framework). Hints can be found for some of the problems if you get completely stuck, but first try to solve them on your own.
   1. **Reminder:** a priority queue ordersits elements based on their values (unlike a regular queue, which only cares about the order in which elements were added). To get to the least-first-out, it has to know what least means.
      1. Standard Java classes (e.g. String and Integer) already have an order: their natural order (lexicographical order for Strings, numerical order for Integers).
      2. If a priority queue stores a custom type (a class you create), that class must be a Comparable type (it must implement the Comparable interface). If you can't remember how to implement Comparable, check the AP powerpoints (or Google it).
   2. More time spent on pen and paper (or in Paint or Notepad, whatever works for you) will generally lead to better outcomes when solving contest-style (or any style) problems.
   3. If you get completely stuck on one problem, you can skip to another and come back to it.