**Priority Queue**

Preparation for LunchRoom

1. Use a **priority queue** in any problem where elements enter in a random order but are removed according to their priority. For example:

* letters (of the alphabet) enter the queue randomly, but come out in alphabetical order.
* in the lunch line, all seniors get served first, juniors next, sophomores next, and freshman last.
* in a hospital, patients who are more likely to die, but are not dead yet, get the organ transplant first.
* in a print queue, shorter print runs get processed before the longer print runs.

"According to their priority" means something different depending on the objects in the program. To make the "priority part" of the priority queue work, Java requires that each element in the queue be a Comparable. As you know, the Comparable interface has a compareTo method. You get to define what "priority" means in the given situation. *A higher priority means a lower compareTo return value.* You have seen this behavior before with strings, for "a".compareTo("b") returns a negative number. Write reasonable compareTo methods for each example below.

|  |  |
| --- | --- |
| **class** Student **implements**   Comparable<Student> | **class** Patient **implements**   Comparable<Patient> |
| { | { |
| **private** String myGradYear; | **private** boolean isAlive; |
| **public** **int** compareTo(Student obj) | **private** double probDying; |
| { | **public** **int** compareTo(Patient obj) |
|  | { |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

2. If two items to be removed are of the same priority (i.e., two seniors are in the lunch line), which one gets removed first? If two items are of equal priority, the one that is removed first is unpredictable. They are *not* necessarily removed in FIFO order, which is what most people expect. Java implements a priorityQueue as a heap, which we will study later. A heap does *not* guarantee that the items of the same priority are removed FIFO.

3. Look at your cheat sheet. What methods are in the PriorityQueue class? \_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Look at your code for McRonaldVIP. Describe the changes you would have to make in the code to implement the VIP feature as a priority queue.

5. You are not expected to know any specific implementation of the PriorityQueue class. You should, however, be able to discuss the effects of common implementations on the Big-O efficiency of the PriorityQueue's methods. Five common implementations are:

|  |  |
| --- | --- |
| insertion |  |
| deletion |  |

i. **An array with elements in random order.** Insertion is done at the end of the list. Removing requires a linear search, then a shift.

|  |  |
| --- | --- |
| insertion | **O(1)** |
| remove() | **O(1)** |

ii. **An array with elements sorted by priority**, highest priority elements at the end. Insertion requires finding the insertion point and then shifting some of the array elements. Deletion means removing the last element in the array.

|  |  |
| --- | --- |
| add() | **O(n)** |
| peek() | **O(1)** |

|  |  |
| --- | --- |
| insertion |  |
| deletion |  |

iii. **A linked list with elements in random order.** Insertion is done at the front of the list. Deletion requires a linear search.

iv. **A linked list with elements sorted by priority,** highest priority elements in front. Insertion requires a traverse. Deletion means removal of the first node.

|  |  |
| --- | --- |
| add() |  |
| remove() |  |

v. **A minimum heap**. A heap is a binary tree structure with the property that the value in every node is less than or equal to the value in each of its children. We’ll study min-heaps when we study trees. After either insertion or deletion, the binary tree is restored to order (reheaped) in O(log n) time. java.util.PriorityQueue actually is a heap.

|  |  |  |
| --- | --- | --- |
| insertion | ***O(log n)*** |  |
| deletion | ***O(log n)*** |  |

**Assignment**

LunchRoom.java. Use java.util.PriorityQueue<E>. The lunch room has decided to serve all Seniors first, then all Juniors, then all Sophomores, and finally the Freshmen. When there are two or more Seniors (for example) in the queue, the one who arrived earliest should get served first. Output the number of each kind of Customer served, the average wait time, and the longest wait time for each kind of customer. You will need

class Customer implements Comparable<Customer>  
 Some people also make 4 subclasses to model each kind of Customer.

**Sample Run**

(1 service area, arrival time < 0.2, service time between 2 and 7 minutes)

*<. . . snip . . .>*

25: [20:Fr, 25:Fr]  
26: [26:Se, 25:Fr]  
27: [26:Se, 25:Fr, 27:Ju]  
28: [26:Se, 25:Fr, 27:Ju]  
29: [26:Se, 25:Fr, 27:Ju]  
30: [26:Se, 30:Se, 27:Ju, 25:Fr]  
31: [26:Se, 30:Se, 27:Ju, 25:Fr, 31:Fr]  
32: [30:Se, 25:Fr, 27:Ju, 31:Fr]

*<. . . snip . . .>*

Customer Total Longest Average Wait  
Senior 47 12 3.4893617021276597  
Junior 56 22 5.0  
Sophomor 53 84 20.830188679245282  
Freshman 55 250 123.6

**Extension**

Make a class MyPriorityQueue which implements a priority queue using one of the possibilities above. Test your resource class with LunchRoom as the driver. How many changes in LunchRoom will you have to make?