**Problem 1 – Priority Queue**

Review and understand the Priority Queue code provided to you. Make sure to understand all the functions and solve the following:

1. In the main of **PQTester.java**, add a class “intComparator” that implements comparator and compares

integers. Check the way “stringLengthComparator” is implemented as the logic is similar.

1. In the main of **PQTester.java** create PQ2 an instance of SortedPQ with an Integer key and Student value with intComparator. Insert the students s1-s4 into PQ2 with their grades as keys then print the elements of the queue.
2. In SortedPQ.java add the boolean method **exists()** that takes in as input a key and a value. Your method should return true if the key already exists in the queue, and false otherwise.
3. Using PQ2 test your **exists()** method on both cases (false and true)

**Problem 2 – Priority Queue and Heap**

In this program you will have to use the generic implementation parameters of the priority queue (use HeapPQ). Remember in the generic implementation, the first parameter indicates the Key while the second is the element itself (in this problem is the student object). You have to write a program that processes a data file of students’ grades. The data arrive in random order; each line stores information about student’s

last name, first name, ID number, grade as a percentage, and a letter grade (a **student.dat** file is provided). For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Smith | Kelly | 438975 | 98.6 | A |
| Johnson | Gus | 210498 | 72.4 | C |
| Reges | Stu | 098736 | 88.2 | B |
| Smith | Marty | 346282 | 84.1 | B |
| Reges | Abe | 298575 | 78.3 | C |

You are supposed first to design a student class that enables you to construct student objects. Your program should prompt the user about the key parameter which can be first name, last name, student ID, or numbered grade. You can prompt to ask the user to select the order (ascending or descending). Thus your program should sort the data either by first name, last name, student ID number, or grade in ascending and descending order using heapsort. You might use the default **Comparator** and/or your own defined **Comparators** to achieve the sort ordering.

For example: Sorting by last name

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Johnson | Gus | 210498 | 72.4 | C |
| Reges | Stu | 098736 | 88.2 | B |
| Reges | Abe | 298575 | 78.3 | C |
| Smith | Kelly | 438975 | 98.6 | A |
| Smith | Marty | 346282 | 84.1 | B |

Sorting by student ID number

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reges | Stu | 098736 | 88.2 | B |
| Johnson | Gus | 210498 | 72.4 | C |
| Reges | Abe | 298575 | 78.3 | C |
| Smith | Marty | 346282 | 84.1 | B |
| Smith | Kelly | 438975 | 98.6 | A |

Hint: you can use the HeapPQ implementation described in the class. In your driver program you declare a PQ for each sorting key taking into consideration the key itself. Also, you have to decide whether the default comparator can be used or you have to define your comparator class and pass its object when you construct the PQ.

**Problem 2 – HeapPriorityQueue**

**A.** Write a method in the HeapIntPriorityQueue class called merge that accepts another HeapIntPriorityQueue as a parameter and adds all elements from the other queue into the current queue, maintaining proper heap order such that the elements will still come out in ascending order when they are removed. Your code should not modify the queue passed in as a parameter. (Recall that objects of the same class can access each other's private fields.)

**B.** Write a method in the HeapIntPriorityQueue class called toString that returns a string representation of the elements in the queue, such as "[42, 50, 45, 78, 61]". The order of the elements in the string does not matter as long as they are all present in the proper format.

.C. Write a method in the HeapIntPriorityQueue class called toArray that returns the elements of the queue as a filled array. The order of the elements in the array is not important as long as all elements from the queue are present in the array, with no extra empty slots before or afterward.

**D**. Write a method named replace that could be added to the HeapPriorityQueue class. This method accepts an element value value1 and a replacement value value2, and finds and replaces one occurrence of value1 with value2 if value1 is present in the heap. You must maintain the heap's ordering after your method's work is done. For example, if a heap priority queue pq contains [/, 12, 41, 35, 56, 71, 52, 40, 84, 60, 78, 99, 66] in its internal heap array, the call of pq.replace(56, 30); would change pq's array to store [/, 12, 30, 35, 41, 71, 40, 52, 84, 60, 78, 99, 66]. A subsequent call of pq.replace(35, 88); would change pq to store [12, 30, 40, 41, 71, 52, 88, 84, 60, 78, 99, 66]. If the value1 is not found in the heap, no change occurs to the heap. You may assume that neither of the values passed is null.

You are allowed to call methods on your priority queue. This method should run in O(N) time where N is the number of elements in your queue. Assume that you are adding to the following class:

**public** **class** **HeapPriorityQueue**<E **extends** Comparable<E>> {

**private** E[] elements;

**private** int size;

​

**public** HeapPriorityQueue() {...}

​

**public** void add(E value) {...}

**public** boolean isEmpty() {...}

**public** E peek() {...}

**public** E remove() {...}

**public** int size() {...}

**public** String toString() {...}

​

**private** void bubbleUp(int index) {...}

**private** void bubbleDown(int index) {...}

**private** int parent(int index) {...}

**private** int leftChild(int index) {...}

**private** int rightChild(int index) {...}

**private** boolean hasParent(int index) {...}

**private** boolean hasLeftChild(int index) {...}

**private** boolean hasLeftRightChild(int index) {...}

**private** void swap(E[] array, int index1, int index2) {...}

}

E. Write a method called isConsecutive that accepts a PriorityQueue of integers as a parameter and returns true if the queue contains a sequence of consecutive integers starting from the front of the queue. Consecutive integers are integers that come one after the other, as in 5, 6, 7, 8, 9, etc., so if the queue stores [7, 8, 9, 10, 11], your method should return true. (Also return true if passed an empty queue.) If your method modifies the state of the queue during its computation, it should restore the queue before it returns. You may use one stack or queue as auxiliary storage.