COMP202 SPRING 2017

PROGRAMMING HOMEWORK 17.04.2017

***Your submissions will be checked for plagiarism and the rules in the course syllabus will be effective.*

***Read the description carefully and understand it well.*

You should only modify **HeapBasedPriorityQueue.java** and **TreeBasedPriorityQueue.java

You **are not** allowed to use any java library that substitutes the parts you are asked to implement and solve. Otherwise, your grade will be 0.

1 DESCRIPTION

Fast Start: Look at AdaptablePriorityQueue.java which defines an adaptable priority queue interface. This file has useful comments. Then look at HeapBasedPriorityQueue.java and TreeBasedPriorityQueue.java which define classes that implement the interface. From then on, start implementing the necessary methods, looking at ArrayBasedHeap.java and AvlTree.java. You should only be modifying the HeapBasedPriorityQueue and TreeBasedPriorityQueue. Use the Test.java and the provided sample output to test and debug your program.

In this lab project, you are going to implement adaptable priority queues in two ways, using Heaps and using AVL Trees. The idea is to simulate an order book for a single product. There is going to be two priority queues, one for sellers and one for buyers. Each entry in the queue is going to be a Key-Value pair. Keys correspond to price (if seller, the desired price to sell, if buyer, the desired price to buy) and the values correspond to people. For example, a buyer defined with the Node(10, “Ahmet”), is named Ahmet and wants to buy the product for 10 TL, at most. Similarly, a seller defined as Node(9, “Mehmet”), is named Mehmet and wants to sell the product for 9 TL. In this example, Mehmet can sell the product to Ahmet, since he is willing to pay an higher price for the product. The priority queue of sellers is a min-heap, since every buyer is interested in buying at the lowest price and the priority queue of buyers is a max-heap, since every seller is interested in selling to the buyer who is offering the most. In this project, buyers and sellers give orders, change their orders, and cancel their orders. These correspond to a specific adaptable priority queue method (see Test.java if you are interested).

You are given implementations of an array-based Heap and an AVL Tree. You are going to write code to implement a given “AdaptablePriorityQueue” interface using the provided classes.

2 METHODS TO BE IMPLEMENTED

You are asked to implement the empty methods in “TreeBasedPriorityQueue” and “HeapBasedPriorityQueue” classes. You do not need to modify any other files given in the project (except for 1 variable in Test.java).

“TreeBasedPriorityQueue” and “HeapBasedPriorityQueue” classes both implement the “AdaptablePriorityQueue” interface but the methods are empty. You need to fill them based on their definitions (see comments). You should

use the methods of the parent classes; note that the tree based PQ extends an AVL Tree class and the heap base PQ extends an array based heap class. You do not need to modify the parent classes. Do not write any code in the "AdaptablePriorityQueue" interface.

The definitions of the methods you need to implement are given below. Shorter versions are also in the code as comments.

void setType(boolean isMin): Sets the type of the priority queue, either as minimum or maximum priority queue, depending on given type parameter. You can assume that this is only called before the PQ is built.

void insert(Key k, Value v): Inserts a new node with the given key and value to the priority queue. For example, insert(12, Ahmet) inserts a node (Ahmet,12) with key 12 and value "Ahmet" to the priority queue.

Key replaceKey(Key k, Value v): Replaces key of the node whose value is v, with the new key k, in the priority queue, and returns the old key of the corresponding node. For instance, assuming (Ahmet,12) is in the priority queue, replaceKey(8, Ahmet) changes the node to (Ahmet, 8) and returns 12. You should return null if the key is not in the priority queue.

Pair<Key, Value> pop(): Removes the appropriate node in the priority queue and returns the key-value pair corresponding to that node. Look at Pair.java as well. You should return null if the priority queue is empty.

Value popValue(): Removes the appropriate node in the priority queue and returns its value. You should return null if the priority queue is empty.

Key popKey(): Removes the appropriate node in the priority queue and returns its key. You should return null if the priority queue is empty.

Pair<Key, Value> top(): Returns the key-value pair corresponding to appropriate node in the priority queue. Look at Pair.java as well. You should return null if the priority queue is empty.

Value topValue(): Returns the value of the appropriate node in the priority queue. You should return null if the priority queue is empty.

Key topKey(): Returns the key of the appropriate node in the priority queue. You should return null if the priority queue is empty.

Value remove(Key k): Removes the node whose key is k from the priority queue and returns the value of that node. For example, assuming (Ahmet,12) is in the priority queue, remove(12) removes the node (Ahmet,12) from the priority queue and returns "Ahmet". You should return null if the key is not in the priority queue.

boolean isEmpty(): Returns true if the are no nodes in the priority queue and false otherwise.

int size(): Returns number of nodes in the priority queue.

Key findByValue(Value v): Returns the key of the node, whose value is v. For instance, assuming (Ahmet,12) is in the priority queue, findByValue("Ahmet") returns 12.

IMPORTANT: You may need to traverse the entire priority queue for some of the methods
IMPORTANT: Pay attention to null return conditions for some of the methods

3 TESTING

You are going to use the main function in the Test.java file to test your code. The order-book interactions are simulated using multiple files. Each file describes the order at a given time step. At each time step, buyers and sellers give orders (buy/sell, change, and cancel). After a timestep, the tops of each priority queue are compared. If the highest selling price is smaller or equal to the lowest buying price, a transaction occurs (the buyer gets the good for the selling price). This is repeated until the tops no longer overlap and the system moves on to the next time step. In this project, we assume that no seller or buyer offers the same price (for the sake of simplicity).

There are two conditions for you to test. The first condition is when the *testType* variable is set to "easy". This condition only has 5 time steps. There is a "easy_output.txt" file for you to match. Obviously, you are not expected to match the durations. Note that the only thing that your implementation influences is the "X sold to Y for z" lines so make sure they match.

In the second condition, set the *testType* variable to be something other than "easy". This condition has 200 time steps. This is a more involved test and will run your implementations towards more edge cases. Make sure that your methods run without any issues on this condition. If your method is running slower than 200 milliseconds on a modern machine, there might be something wrong.

We are also going to test your code with un-released data.

4 AVL TREE

An AVL Tree is given to help you implement the "TreeBasedPriorityQueue". The TreeBasedPriorityQueue class extends the AvlTree class. AvlTree.java file contains an implementation of an AVL Tree with necessary methods and variables included. You do not need to make any modifications to AvlTree.java file. Instead, you should use the methods and variables in the AVL Tree class while implementing the "TreeBasedPriorityQueue". The explanations of some of the methods given in AvlTree.java are as follows:

void insert(Key key, Value value): Inserts a new node with the given key and value to the AVL Tree. For instance, insert(12,Ahmet) inserts a new node (Ahmet,12) with key 12 and value "Ahmet" to the AVL Tree.

void delete(Key delKey): Removes the node whose key is delKey from the AVL Tree. For instance, assuming

(Ahmet,12) is in the AVL Tree, delete(12) removes the node (Ahmet,12) from the AVL Tree.

Value get(Key key): Returns the value of the node whose key is k. For instance, assuming (Ahmet,12) is in the AVL Tree, get(12) returns "Ahmet".

void printTree(): Displays key, value, balance and height information for all nodes in the AVL Tree, by performing an inorder traversal. Useful for debugging.

5 ARRAY BASED HEAP

An Array Based Heap is given to help you implement the "HeapBasedPriorityQueue". The HeapBasedPriorityQueue class extends the ArrayBasedHeap class. ArrayBasedHeap.java file contains an implementation of an Array Based Heap with necessary methods and variables included. You do not need to make any modifications on ArrayBasedHeap.java file. Instead, you should use the methods and variables in Array Based Heap class while implementing the "HeapBasedPriorityQueue". The explanations of some of the methods given in ArrayBasedHeap.java are as follows:

void insert(Key key, Value value): Inserts a new node with the given key and value to the heap. For instance, insert(12,Ahmet) inserts a new node (Ahmet,12) with key 12 and value "Ahmet" to the heap.

Value remove(): Removes the first node in the heap and returns it value.

Node peek(): Returns the first node in the heap.

Key getKey(**int** i): Returns key of the i^{th} node

Value getValue(**int** i): Returns value of the i^{th} node

void printLevelOrder(): Displays keys and values off all nodes in the heap, by performing level order traversal. Useful for debugging.

6 PAIR

Pair.java contains a Pair class which is essentially a data structure that holds a key-value pair, used for getting both of these variables as a returned variable for top() and pop() methods.

RULES

1) You **are not** allowed to use any java library that substitutes the parts you are asked to implement and solve.
2) Your code will be tested with different input files.
3) As this is a homework, you may end up discussing the solutions however, copying code is forbidden and your submission will be checked against plagiarism.