Introduction to Java

CS9053

Prof. Dean Christakos

MIDTERM

Friday, June 28th, 2024

DUE: Thursday, July 11th, 2024

NO GRACE PERIOD OR EXTENSION

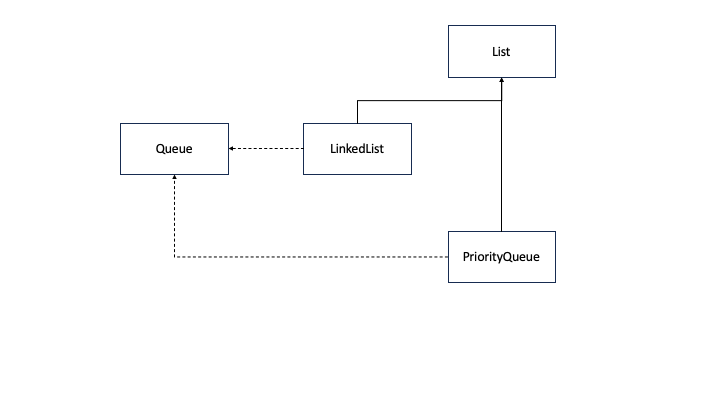
**REMEMBER TO DO THE WRITTEN QUESTIONS BEFORE THE DEADLINE! THEY ARE ON BRIGHTSPACE, NOT ON THIS DOCUMENT!**

**Coding Section**

You’re going to build a queue and a priority queue from scratch.

Many of these are implemented in Java already, but you’re not going to use those. You can’t make use of them to implement these, though you can use the ArrayList class to implement the PriorityQueue (explained below), if you want.

You will implement this class hierarchy: remember that solid lines are subclass relationships and dotted lines represent interface relationships:



All of your classes and test code will go in the cs9053.lists package already set up in the code

1. Implement the interface Queue. As a reminder, a Queue is a FIFO data structure. Elements are added to the queue with the method offer() and removed with the method remove(). They are removed in the order they are added, unless some other ordering is specified, as in a PriorityQueue. A Queue has the following methods:

boolean offer(E e) – Insets the specified element to the queue. Always returns true

E remove() – Retrieves and removes the head of the queue. Throws a NoSuchElementException if the queue is empty

E peek() – Retrieves but does not remove the head of the queue or returns null if the queue is empty

E poll() – Retrieves and removes the head of the queue, or returns null if the queue is empty

E element() – retrieves, but does not remove the head of the queue. Throws a NoSuchElementException if the queue is empty

A queue interface is parameterizable. You should be able to declare a class such as:

public class QueueImplementation<E> implements Queue<E> {

}

QueueImplementation<String> myQueue = new QueueImplemmentation<String>();

2. Implement the abstract class List. This is a class, but it just has a few abstract methods:

void add(E e) – add an element to the end of the list

E get(int i) – return the element at index i

E remove(int i) – remove the element at index I and return it

void set(E e, i) – set the element at index i to the element e

void add(E e, i) – add the element e to index i, as an “insert”

int size() – return the size of the list

boolean contains(Object o) – return true if the object o is in the list (as matched by the equals() method)

I realize that this might better qualify as an interface rather than an abstract class, but we will chalk this up to a design choice

3. Next, you are going to implement a LinkedList which is a subclass of List. Reminder that a LinkedList has a head and a tail (both null if the LinkedList is empty). You will implement the LinkedList by implementing the Node class, defined by the following UML:

Node<E>

+Node()

+Node(E val)

+Node(E val, Node<E> next)

+getNext() : E

+setNext(Node<E> n)

+getVal(): E

+setVal(E val): void

(figure out what fields are needed for yourself)

You will obviously have to implement the abstract methods of the List class.

Next, **the LinkedList implements the Queue interface**. This means that the LinkedList will also have to implement the methods of the Queue interface.

Linked lists are not random access, which means that get, remove, and set methods will have to iterate through the list to find the index element, but on the other hand, methods like offer() and remove() are going to be easy to implement.

Note: offer() and remove()should run in O(1). Figure out how best to do that for a singly-linked list.

In addition to the List methods, the LinkedList looks like this:

LinkedList<E>

+LinkedList()

+getHead() : E

+getTail() : E

Note that you shouldn’t be able to manipulate the individual nodes of a LinkedList directly from outside the LinkedList class—that is all just for internal administration.

3. By this point, you have a Queue interface, a Node class, and a LinkedList class. Finally, you are going to implement a PriorityQueue. What’s a PriorityQueue? It is a queue, with offer() and remove() methods where the items are removed in an ordering, from highest to lowest of that ordering. This means that if I have a priority queue of integers, pqi, and put these values on the queue:

pqi.offer(4);

pqi.offer(1);

pqi.offer(53);

pqi.offer(21);

pqi.offer(9);

Then the return values of remove will be:

pqi.remove() -> 53

pqi.remove() -> 21

pqi.remove() -> 9

pqi.remove() -> 4

pqi.remove() -> 1

You have a lot of flexibility in how to implement this, however you want. If you were asked to implement this in an interview, I would suggest using a max-heap (in fact, the actual PriorityQueue, which we will cover in an upcoming lecture, is the “heap” data structure in Java). However, you should probably handle this using some kind of insertion sort with binary search. But, the PriorityQueue needs to be a subclass of the List class and needs to implement the Queue interface.

Also, it should take as a constructor argument a Comparator. If it does not have a comparator, it should use the natural ordering of the data type it has been parameterized with.

After implementing the PriorityQueue and its methods, **create a two examples of a PriorityQueue of Integers – one which uses the natural ordering of integers and one which uses reverse ordering.**

This means you’re implementing this:

PriorityQueue<E>

+ PriorityQueue()

+PriorityQueue(

Comparator<? super E> c)

At the end, you should be able to do this:

Queue<Integer> q1 = new LinkedList<Integer>();

q1.offer(4); q1.offer(3); q1.offer(5);

Queue<Integer> q2 = new PriorityQueue<Integer>();

q2.offer(4); q2.offer(3); q2.offer(5);

System.out.println(“q1:”);

while (q1.size() > 0)

System.out.print(q1.remove() + “ “);

System.out.println()

System.out.println(“q2:”);

while (q2.size() > 0)

System.out.print(q2.remove() + “ “);

System.out.println()

And receive the output:

q1:

4 3 5

q2:

5 4 3

Show how to create a PriorityQueue that will reverse the order of the removal with a lambda function.