ITI1121
Assignment # 2

On a computer with a single processor, in order to simulate multi-tasking an operating system rapidly switches between the various programs running on a computer, called *jobs*, running each for fractions of a second a time. The result is the appearance that all the applications are running at the same time.

There are many strategies used by operating systems to choose the next application to run but, in general, they are implemented using queues. In this assignment we will implement two different kinds of queues and, using them, explore two different job scheduling strategies.

Several classes have been provided to help you:

Class Job This class stores all the information about Job. Your queues should store objects of this class. This class is complete, you do not need to modify it. There is no need to submit this class.

Class Simulation This is the actual operating system simulation class. It includes the main method for this assignment. You will not need to modify this class but you will need to make some minor adjustments to the main method. There is no need to submit this class.

Class PriorityQueue This is a skeleton of the PriorityQueue class with some of the method complete. You will need to fill in the missing methods. Your final version of this class should be submitted along with the classes you create.

1 Style

Conforming your code to proper Java conventions is important! The main ones that we are looking for are:

Spacing All code "within" something should be indented one level deeper than whatever they are in. Thus, all code within a class should be indented deeper than the class declaration. All the code within a method should be indented deeper than the method declaration. All code within an if, while, for, switch statement should be indented deeper than the if, while, for, switch statement.

Naming Every word in a class name should be capitalized (*e.g.* LinkedListQueue). For methods and variables, every word after the first word should be capitalized (*e.g.* reverseHeapify). For constants (variables declared final), should be only capital letters, word separated by underscores (*e.g.* INTEREST_RATE). All names should be meaningful (exception: it is alright to have loop variables i and j).

Comments At a minimum, all classes, public methods and public variables should have a JavaDoc comment next to them. It is good practice to comment sections of your code as well (if you get the code wrong but the comment is right you may lose fewer marks).

Marks: 3 marks for proper spacing, 3 marks for proper naming and 4 marks for proper comments. [Total 10 marks]

It is IMPORTANT that all classes compile and run! It is -50% (to the marks of that class) for every class that does not compile and -25% for every class that does not run!

2 Queue

We will be implementing two different types of queues and running them on the same simulator. Since both are queues they must have many methods in common. It is important that we inform Java of this fact as otherwise we would have to write a separate simulator for each queue. To do this we need to create an interface for queues.

Interface Queue Queues are "line ups", *e.g.* a line up to get a ticket. They have the property that elements are removed from the queue in the same order that they were put into the queue, *e.g.* the first person in a line is the first person to be served.

All queues have four common methods: `isEmpty`, `enqueue`, `dequeue` and `clear`. The implementations of each of these methods is dependant on the type of queue which is why we must use an interface. However, below is a description of what each method is supposed to do.

`isEmpty` checks to see if the queue contains an elements. It returns `true` is the queue is empty and `false` otherwise.

`enqueue` adds an element to the back queue, *e.g.* when a person enters a line they start go to the back and wait their turn. Since our queues store objects of class `Job`, `enqueue` should take in (as a parameter) an object of class `Job` and add it to the back of the queue.

`dequeue` removes an element from the front of the queue, *e.g.* the person at the front of the line gets served. Since our queues store objects of class `Job`, `dequeue` removes the front `Job` from the queue and returns it.

`clear` removes all the elements from queue, emptying it.

Marks: 2 marks for creating the interface, 2 marks for each method [Total 10 marks]

3 Linked-List Queue

As you know from class, a linked list is a like an array but it is implement by a series of nodes, containing data, which point to each other. It is very efficient to add and remove items from a linked list but it is difficult to find items in the list. This is the opposite of an array where it is very efficient to find items but it is inefficient to add and remove items. However, for linked lists it is extremely efficient to add, remove and find items at the start or end of the list which makes them perfect for queues. Thus, the linked list is the most common way to implement a queue.

Before we can build a linked-list queue we must first build a linked list. For a queue all we need is a singly linked list meaning that each node is linked to the next node.

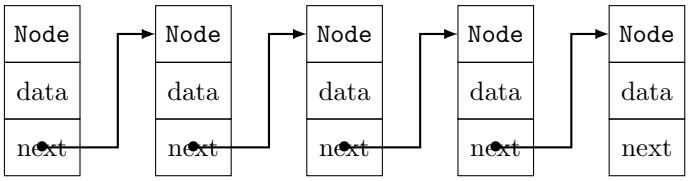

Figure 1: A diagram of a singly linked list.

Class Node A linked list is merely a group of nodes that are linked to each other. Thus, to implement a linked list we need only implement a `Node` class. Each `Node` must store some **data**, in our case the **data** should be of type `Job`. Because we are building a singly linked list, a `Node` must also store a linked to the **next** `Node`, if there is one. It there is no **next** `Node` then it should be `null`.

The `Node` class merely stores data. Thus, for methods it should provide **getters** and **setters** for the variables. It should also have **two constructors**: one which gets the values for both variables from the user and one which only gets the `Job` from the user and assumes that the next `Node` is `null`.

Class LinkedListQueue Since a queue is a "line" two important pieces of information are needed: the start of the line and the end of the line, called the **head** and **tail** respectively. These are obviously of type `Node`. The linked list, if properly implemented, will take care of storying everybody in between the head and tail.

`enqueue` works just like in a "line" in real life: a new person arrives and goes to the end of the line, becoming the new end of the line. In our case we simply add a new `Node` end to the linked list (the tail `Node`). This `Node` becomes the new tail.

`dequeue` is different than a "line" in real life. In real life, the whole line moves forward when a person at the front is served, but this is inefficient in

Java. Instead, we can imagine the service counter moving forward towards the next person in line. Thus, the head `Node` is removed from the linked list (and its `Job` returned) and the next person in the line becomes the new head of the line.

It is up to you to figure out how `isEmpty` and `clear` work.

Marks: For class `Node`, 1 mark for creating each instance variable, 1 mark for each getter/setter method, 2 marks for each constructor [Total 10 marks]

For class `LinkedListQueue`, 5 marks for `enqueue`, 5 marks for `dequeue`, 4 marks for `clear`, 3 marks for `isEmpty` and 3 marks for the constructor [Total 20 marks]

4 Priority Queue

A priority queue operates differently than a regular queue as it allows some elements to move through the queue more quickly than others. The classic example of a priority queue is a hospital emergency room where the sickest patients are the first to see the doctor even if a less sick patient has been waiting longer.

Since a queue's primary responsibility is determining the next element to be dequeued it must do so very efficiently. In a priority queue this can be tricky as we want to avoid examining every element in the queue when an element is enqueued or dequeued (this is considered inefficient). Thus, one common way to store a priority queue that does this efficiently is a *heap*.

A *heap* is an array representation of a *binary tree* (a tree where every node has at most two children) such that each subtree of the tree has the *heap property*. The *heap property* is that the root of a tree is greater (has a higher priority) than any of the other elements in the subtree. Even though it is a binary tree, a *heap* is always stored as an array. The children of an element at index i in array are stored at indices $2i$ and $2i + 1$. The parent of i can be found at index $\lfloor i/2 \rfloor$.

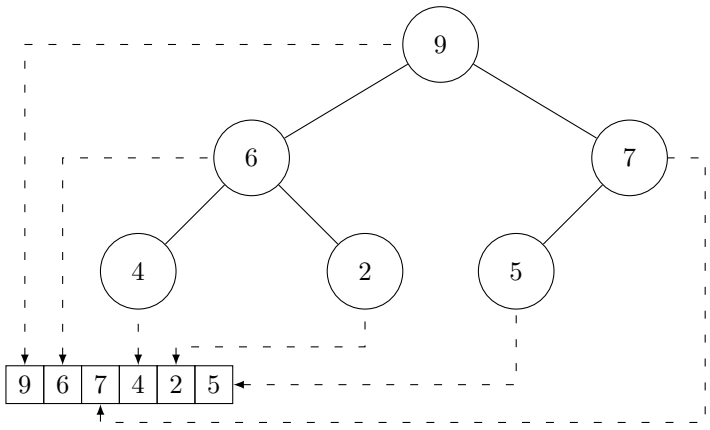

Figure 2: A diagram of a heap.

For example, in the heap in Figure 2 the root is node containing 9, which is also the largest element in the array. It is stored as position 0 in the array representation of the heap. It has two children6 and 7 stored at positions 1 and 2 in the array which are bigger than all their children (they are the largest in their respective subtrees). It leaves are 4, 2 and 5.

The trick to a heap is adding and removing elements while preserving the heap property. There are two methods that do this: `heapify` and `reverseHeapify`. Whenever an element is altered a call to one of these methods will restore the heap property. If the element altered is an *internal node* of the tree (a node with one or more children) then `heapify` should be called. If it is a *leaf* of the

tree (a node with no children) then `reverseHeapify` should be called.

`heapify` works by comparing the current internal node with its children to see which is the largest. If the root is not the largest then it is swapped with the largest of its children. Since the child has now been altered it needs to check to make sure it still satisfies the heap property. `heapify` works its way down the tree in this manner and, when it is complete, the tree should again be a heap. The details of how `heapify` works are a little complex but the method is provided for you, you just need to understand when to use it, which we will discuss a bit more below.

`reverseHeapify` works by comparing the current node with its parent. It the current node is larger than its parent then it swaps itself with its parent. The parent still might not be correct, it now needs to compare its parent with its grand parent, and so on, until it reaches the top of the tree. `reverseHeapify` is easy to implement using a `while loop` which keeps going until parent is larger than the child. While the parent is smaller than the child it swaps the parent with the child and makes then moves up to the parent (so that the parent will become the child during the next iteration of the loop). We will discuss when to use `reverseHeapify` a bit more below.

Class PriorityQueue The priority of a `Job` is determined by the owner of the `Job`. `Job` has a method `getOwner` that returns a number 0, 1 or 2 indicating the owner. The higher the number of the owner the higher the priority of the `Job`. There will likely be many `Job`s with the same priority.

Since we are using a heap (a type of array) we don't need to keep track of the head like in a linked-list queue. This is because it is always position 0 in the array. However, we do need to keep track of a**tail**. The **tail** is the position after the last used position in the array. Thus, the **tail** of an empty queue will be at position 0.

Included with the assignment is a skeleton the the class. Some of the methods needed by the heap have been implemented for you. Methods `left`, `right` and `parent` give the index of the left child, right child and parent respectively of an index i. The method *swap* swaps two elements of the array. `heapify` is also implemented for you. Also, the **constructor** has been implemented for you already, there is no need to modify it.

You will need to implement the queue methods: `isEmpty`, `enqueue`, `dequeue` and `clear`. You will also need to implement `reverseHeapify`. Since a heap is an array, you will also need to implement `resize`, which doubles the size of the array just like in Assignment #1.

`enqueue` adds an element to the end of the heap (the tail). Since the heap is stored as an array, this simply means adding an element at the tail and incrementing the tail. However, we must ensure that the heap property is maintained. Since the last element in the heap cannot have any children it must be a leaf so we can call `reverseHeapify` to enforce the heap property. Also, if the heap is too small to add a new element, we must call `resize` before adding the element.

`dequeue` removes the first element in the heap and returns it. The challenge is that we cannot leave the beginning of the array empty. We cannot shift the other elements forward either, not only is this inefficient but it will also destroy the heap property. The trick is to realize that we can replace the first element with absolutely any element in the heap so long as `heapify` is called. There is only one element that can be moved without destroying the heap property: the last element. Thus, we swap the first element and the last element of heap, remove and return the last element and then call `heapify` on the first element.

`reverseHeapify` is described above. `resize` is the same as Assignment #1. It is up to you to figure out how to implement `isEmpty` and `clear`.

Marks: For class `PriorityQueue`, 6 marks for `reverseHeapify`, 5 marks for `enqueue`, 5 marks for `dequeue`, 4 marks for `resize`, 4 marks for `clear`, 3 marks for `isEmpty` and 3 marks for the constructor [Total 30 marks]

5 Operating System Scheduler

Two scheduling strategies for an operating system scheduler are *first come first serve (FCFS)* and *fixed priority pre-emptive scheduling (FPPS)*. Since queues operate on a first come first serve basis, FCFS is implemented using a queue. Similarly, FPPS is implemented using a priority queue.

The operating system scheduler simulation is already provided for you. To use it you simply need to modify the `main` method to run the simulator using either the `LinkedListQueue` or the `PriorityQueue`.

Run the simulator a few times with each type of queue. Answer the following four questions about your experiments in a **plain text file** called `scheduler.txt`.

1. What are differences in how the jobs are managed between FCFS and FPPS?

2. What is are the advantages of FCFS over FPPS and *vice versa*?

3. What potential problems do you see happening if you were using an operating system with an FCFS scheduler?

4. What potential problems do you see happening if you were using an operating system with an FPPS scheduler?

Marks: THIS QUESTION WILL NOT BE MARKED IF YOU PROGRAM DOES NOT COMPILE AND RUN!

3 marks for question 1, 3 marks for question 2, 2 marks for question 3, 2 marks for question 4 [Total 10 marks]

6 Submit

Please submit the following files:

- Queue.java
- Node.java
- LinkedListQueue.java
- PriorityQueue.java
- scheduler.txt