Credit Name: CSE 3020 Computer Science 4

Assignment: QueueList

How has your program changed from planning to coding to now? Please Explain

The QueueList program uses a LinkedList to model as a Queue structure, which is faster than our original design but also requires more memory.

It has 3 classes. A node class, our QueueList class which uses nodes for the queue data structure rather than a data type like int or string. Lastly we have a tester class.

First I coded the Node class

```
1 package Mastery.QueueList;
2
3 public class Node
4 {
5     private Object data;
6     private Node next;
7
8     public Node(Object item)
9     {
0         data = item;
1         next = null;
2     }
```

First I have 2 instance variables. One is for the data held by a node and the other is a pointer to the next node.

I then declared and initialized a constructor with a single parameter which initialized the data variable. The next Node was set initially to null because it has no other nodes currently to compare its position with.

```
    public Node getNext()
    {
        return(next);
    }

    public void setNext(Node newNode)
    {
        next = newNode;
    }

    public Object getData()
    {
        return data;
    }
}
```

Then we made 3 simple methods which: return the value of the node next to the current one, another that sets the value of the next node and the last which returns the data associated with a node.

Our next class defines the QueueList data structure.

```
private Node front;
private Node rear;
private int size;

public QueueList()
{
    front = null;
    rear = null;
    size = 0;
}
```

We have very similar instance variables and constructors, the only difference is that we are using nodes for the front and rear list positions rather than string, int or objects. Again we declare and initialize the QueueList constructor to have the front and rear set to null and the size set to 0. This is because there are no items yet in the list.

```
public Object dequeue()
{
    if(isEmpty())
    {
        throw new IllegalStateException("Your Queue is empty.");
    }
    else
    {
        Object item = front.getData();

        front = front.getNext();

        size --;
        return(item);
    }
}
```

our first method is to dequeue(remove an item from the front). We first check if the list is empty. If it is then we throw an exception mentioning that the queue is empty. Otherwise we save the value of the item we are removing and we move the index of the front item to

the next one down meaning that the previous front has no more relations to the queue because the node is not linked to another anymore. We then decrease the size of the list by 1 and return the value of the node removed.

```
public Object enqueue(Object item)
{
        Node newNode = new Node(item);
        if (isEmpty())
        {
            front = newNode;
        }
        else
        {
            rear.setNext(newNode);
        }
        rear = newNode;
        size++;

        return rear.getData();
    }
```

The next method is to enqueue an item (add to the back of the list). We define and initialize a value for the new node we are adding and it is the value of the parameter in the method. Again we check if the list is empty. If it is then the node we are adding is the first element and is defined as the front node. Otherwise we link the current last node to our new node and set it as the next node(this handles the pointer of the last 2 nodes.) then we define the rear to be the new node we added and we increment the size of the queue by 1 and return the value of the new node added.

```
public boolean isEmpty()
{
    if(front == null && rear == null)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

We make another method to check if our lists are empty. This method is important in stopping a lot of null pointer errors and out of bounds exceptions. For this method we return true if the front node and rear node are null and false otherwise.

```
public int getSize()
{
    return size;
}

public Object getFront()
{
    return front.getData();
}
```

Lastly we have 2 methods in this class which return the size of the queue. We do not need to do any other calculations in this method because the size is constantly adjusted in the dequeue and enqueue methods. Then we have a method which returns the value of the front node.

Now the last class is the client code and tester.

```
public class QueueListClient
{
    public static void main(String[] args)
    {
        QueueList ql = new QueueList();
        Scanner input = new Scanner(System.in);
```

We define the main method and initialize the the scanner for user input and the QueueList to do different actions with the Queues.

```
int choice = -1;

do
{
System.out.println("============================== Queue List ==============================");
System.out.println("Choose one of the following choices.");
System.out.println("(1)Enqueue Item \n(2)Dequeue Item \n(3)See Front Item \n(4)Display List Size \n(0)Quit");

choice = input.nextInt();
```

We define an int variable choice which is the action the user wants to do with the list. We then prompt the user with the option choices and their response is the value of the choice variable.

```
if(choice > 4 || choice < 0)
{
    System.out.println("Error. Select a valid choice.");
}
```

If their choice was above 4 OR below 0 we show an error message and ask them to select a valid choice.

```
else
{
    switch(choice)
    {
    case 0: System.out.println("Successfully quit the application");
            System.exit(0);break;

    case 1: System.out.println("Enter a value:");
            int item = input.nextInt();
            System.out.println(ql.enqueue(item) + " was added");break;

    case 2: System.out.println(ql.dequeue() +" was removed");break;

    case 3: System.out.println(ql.getFront());break;

    case 4: System.out.println(ql.getSize());

    }
}
}
while(choice != 0);
```

otherwise we have a switch statement to handle all the different options more efficiently than if-else statements.

- ***If they select 0, we quit the application.***
- ***If they select 1, we ask them for a value to add to the list we then call the enqueue method and print the relevant message.***
- ***If they select 2, we run the dequeue method and display a "removed" message.***
- ***If we select 3, we print the front node's value.***
- ***Lastly if they select 4 we print the size of the list.***

This repeats while they dont select 0 in a do-while loop so that they can do as many actions as they want to test all the different cases.