Data Structures and Operations(Linked List,Stack,Queue): Day 5:

Task 1: Implementing a Linked List

Write a class **CustomLinkedList** that implements a singly linked list with methods for **InsertAtBeginning**, **InsertAtEnd**, **InsertAtPosition**, **DeleteNode**, **UpdateNode**, and **DisplayAllNodes**. Test the class by performing a series of insertions, updates, and deletions.

```java
package wipro.com.assignmennt04;
class CustomLinkedList
{
    // Node class representing each node in the linked list
    class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }

    // Head of the linked list
    private Node head;

    // Constructor to initialize the linked list
    public CustomLinkedList() {
        this.head = null;
    }

    // Method to insert a node at the beginning of the linked list
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    // Method to insert a node at the end of the linked list
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
        }
    }
```

```java
// Method to insert a node at a specific position in the linked list
public void insertAtPosition(int data, int position) {
    Node newNode = new Node(data);
    if (position == 0) {
        insertAtBeginning(data);
        return;
    }
    Node temp = head;
    for (int i = 0; i < position - 1; i++) {
        if (temp != null) {
            temp = temp.next;
        } else {
            throw new IndexOutOfBoundsException("Position out of bounds");
        }
    }
    newNode.next = temp.next;
    temp.next = newNode;
}

// Method to delete a node by its value
public void deleteNode(int data) {
    if (head == null) {
        return;
    }
    if (head.data == data) {
        head = head.next;
        return;
    }
    Node temp = head;
    while (temp.next != null && temp.next.data != data) {
        temp = temp.next;
    }
    if (temp.next != null) {
        temp.next = temp.next.next;
    }
}

// Method to update a node's value by its position
public void updateNode(int position, int newData) {
    Node temp = head;
    for (int i = 0; i < position; i++) {
        if (temp != null) {
            temp = temp.next;
        } else {
            throw new IndexOutOfBoundsException("Position out of bounds");
        }
    }
    if (temp != null) {
        temp.data = newData;
    }
}

// Method to display all nodes in the linked list
public void displayAllNodes() {
    Node temp = head;
```

```java
        while (temp != null) {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        }
        System.out.println("null");
    }

    // Main method to test the CustomLinkedList class
    public static void main(String[] args)
    {
        CustomLinkedList list = new CustomLinkedList();

        // Test insertions
        list.insertAtBeginning(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
        list.insertAtPosition(25, 2);

        // Display the list
        System.out.println("List after insertions:");
        list.displayAllNodes();

        // Test update
        list.updateNode(2, 26);
        System.out.println("List after updating node at position 2 to 26:");
        list.displayAllNodes();

        // Test deletion
        list.deleteNode(26);
        System.out.println("List after deleting node with value 26:");
        list.displayAllNodes();

        // Test deletion of head node
        list.deleteNode(10);
        System.out.println("List after deleting head node with value 10:");
        list.displayAllNodes();
    }
}
```

Output:

```
List after insertions:
10 -> 20 -> 25 -> 30 -> null
List after updating node at position 2 to 26:
10 -> 20 -> 26 -> 30 -> null
List after deleting node with value 26:
10 -> 20 -> 30 -> null
List after deleting head node with value 10:
20 -> 30 -> null
```

Task 2: Stack and Queue Operations

1) Create a **CustomStack** class with operations **Push**, **Pop**, **Peek**, and **IsEmpty**.Demonstrate its LIFO behavior by pushing integers onto the stack, then popping and displaying them until the stack is empty.

```java
package wipro.com.assignmennt04;
import java.util.EmptyStackException;
public class CustomStack
 {

    private static final int MAX_SIZE = 100; // Maximum size of the stack
    private int[] stackArray; // Array to store elements of the stack
    private int top; // Index of the top element in the stack

    // Constructor to initialize the stack
    public CustomStack() {
        stackArray = new int[MAX_SIZE];
        top = -1; // Stack is initially empty
    }

    // Method to push an element onto the stack
    public void push(int data) {
        if (top == MAX_SIZE - 1) {
            throw new StackOverflowError("Stack is full");
        }
        stackArray[++top] = data;
    }

    // Method to pop the top element from the stack
    public int pop() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return stackArray[top--];
    }

    // Method to peek at the top element of the stack without removing it
    public int peek() {
        if (isEmpty()) {
            throw new EmptyStackException();
        }
        return stackArray[top];
    }

    // Method to check if the stack is empty
    public boolean isEmpty() {
        return top == -1;
    }
    // Main method to demonstrate the behavior of the custom stack
    public static void main(String[] args) {
        CustomStack stack = new CustomStack();

        // Push integers onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);
        stack.push(40);
        stack.push(50);

        // Pop and display elements until the stack is empty
        System.out.println("Elements popped from the stack (LIFO order):");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

Output:

```
Elements popped from the stack (LIFO order):
50
40
30
20
10
```

2) Develop a **CustomQueue** class with methods for **Enqueue**, **Dequeue**, **Peek**, and **IsEmpty**. Show how your queue can handle different data types by enqueuing strings and integers, then dequeuing and displaying them to confirm FIFO order.

```
package wipro.com.assignmennt04;
import java.util.NoSuchElementException;

public class CustomQueue<T>
 {

    private static final int MAX_SIZE = 100; // Maximum size of the queue
    private Object[] queueArray; // Array to store elements of the queue
    private int front, rear, size; // Indexes for front, rear, and size of the queue

    // Constructor to initialize the queue
    public CustomQueue() {
        queueArray = new Object[MAX_SIZE];
```

```java
        front = 0;
        rear = -1;
        size = 0;
    }

    // Method to enqueue an element into the queue
    public void enqueue(T data) {
        if (size == MAX_SIZE) {
            throw new IllegalStateException("Queue is full");
        }
        rear = (rear + 1) % MAX_SIZE;
        queueArray[rear] = data;
        size++;
    }

    // Method to dequeue an element from the queue
    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        T removedElement = (T) queueArray[front];
        queueArray[front] = null;
        front = (front + 1) % MAX_SIZE;
        size--;
        return removedElement;
    }

    // Method to peek at the front element of the queue without removing it
    public T peek() {
        if (isEmpty()) {
            throw new NoSuchElementException("Queue is empty");
        }
        return (T) queueArray[front];
    }

    // Method to check if the queue is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Main method to demonstrate the behavior of the custom queue
    public static void main(String[] args)
 {
        CustomQueue<Object> queue = new CustomQueue<>();

        // Enqueue strings and integers into the queue
        queue.enqueue("First");
        queue.enqueue(10);
        queue.enqueue("Second");
        queue.enqueue(20);
        queue.enqueue("Third");
        queue.enqueue(30);

        // Dequeue and display elements to confirm FIFO order
        System.out.println("Elements dequeued from the queue (FIFO order):");
        while (!queue.isEmpty()) {
            System.out.println(queue.dequeue());
        }
    }
}
```

Output:

```
Elements dequeued from the queue (FIFO order):
First
10
Second
20
Third
30
```

Task 3: Priority Queue Scenario

a) Implement a priority queue to manage emergency room admissions in a hospital. Patients with higher urgency should be served before those with lower urgency.

```
package wipro.com.assignmennt04;
import java.util.*;

class Patient
{
    private String name;
    private int urgency;

    public Patient(String name, int urgency) {
        this.name = name;
        this.urgency = urgency;
    }

    public String getName() {
        return name;
    }

    public int getUrgency() {
        return urgency;
    }

    @Override
    public String toString() {
        return "Patient{" +
                "name='" + name + '\'' +
                ", urgency=" + urgency +
                '}';
    }
}

class EmergencyRoom {
    private PriorityQueue<Patient> priorityQueue;

    public EmergencyRoom() {
        // Creating a priority queue with a custom comparator
        priorityQueue = new
PriorityQueue<>(Comparator.comparingInt(Patient::getUrgency).reversed());
    }

    public void admitPatient(String name, int urgency) {
        Patient newPatient = new Patient(name, urgency);
        priorityQueue.offer(newPatient);
        System.out.println(name + " admitted to the emergency room.");
    }

    public Patient treatPatient() {
        if (priorityQueue.isEmpty()) {
            System.out.println("No patients in the emergency room.");
            return null;
        }
        Patient nextPatient = priorityQueue.poll();
        System.out.println("Treating patient: " + nextPatient.getName());
        return nextPatient;
    }

    public void displayPatients() {
        System.out.println("Patients in the emergency room:");
        for (Patient patient : priorityQueue) {
            System.out.println(patient);
        }
    }
}

public class Main{
    public static void main(String[] args)
 {
        EmergencyRoom emergencyRoom = new EmergencyRoom();

        emergencyRoom.admitPatient("Vishal",3);
        emergencyRoom.admitPatient("Sarthak",1);
        emergencyRoom.admitPatient("Om", 2);
        emergencyRoom.admitPatient("Shubham",4);
        emergencyRoom.displayPatients();
        emergencyRoom.treatPatient();
        emergencyRoom.treatPatient();
        emergencyRoom.displayPatients();
    }
}
```

Output:

```
Vishal admitted to the emergency room.
Sarthak admitted to the emergency room.
Om admitted to the emergency room.
Shubham admitted to the emergency room.
Patients in the emergency room:
Patient{name='Shubham', urgency=4}
Patient{name='Vishal', urgency=3}
Patient{name='Om', urgency=2}
Patient{name='Sarthak', urgency=1}
Treating patient: Shubham
Treating patient: Vishal
Patients in the emergency room:
Patient{name='Om', urgency=2}
Patient{name='Sarthak', urgency=1}
```