Day 5:

Task 1: Implementing a Linked List

1) 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.

```
public class CustomLinkedList {

    private Node head;
    private int size;

    private static class Node {
        int data;
        Node next;

        public Node(int data) {
            this.data = data;
        }
    }

    public void InsertAtBeginning(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
        size++;
    }

    public void InsertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }
        Node current = head;
        while (current.next != null) {
            current = current.next;
        }
        current.next = newNode;
        size++;
    }


    public void InsertAtPosition(int data, int position) throws
IndexOutOfBoundsException {
        if (position < 0 || position > size) {
            throw new IndexOutOfBoundsException("Invalid position");
        }
        Node newNode = new Node(data);
        if (position == 0) {
            InsertAtBeginning(data);
            return;
        }
        Node current = head;
        for (int i = 0; i < position - 1; i++) {
            current = current.next;
        }
        newNode.next = current.next;
        current.next = newNode;
        size++;
    }

    public void DeleteNode(int data) {
        if (head == null) {
            return;
        }
        if (head.data == data) {
            head = head.next;
            size--;
            return;
        }
        Node current = head;
        while (current.next != null && current.next.data != data) {
            current = current.next;
        }
        if (current.next != null) {
            current.next = current.next.next;
            size--;
        }
    }

    public void UpdateNode(int data, int position) throws
IndexOutOfBoundsException {
        if (position < 0 || position >= size) {
            throw new IndexOutOfBoundsException("Invalid position");
        }
        Node current = head;
        for (int i = 0; i < position; i++) {
            current = current.next;
        }
        current.data = data;
    }

    public void DisplayAllNodes() {
        if (head == null) {
            System.out.println("List is empty");
            return;
        }
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " -> ");
            current = current.next;
        }
        System.out.println("null");
    }


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

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.

```
public class CustomStack {
    private int[] items;
    private int top;

    public CustomStack(int capacity) {
        items = new int[capacity];
        top = -1; // Stack is initially empty
    }


    public void push(int item) {
        if (isFull()) {
            throw new StackOverflowError("Stack is full");
        }
        items[++top] = item;
    }

    public int pop() {
        if (isEmpty()) {
            throw new EmptyStackException("Stack is empty");
        }
        return items[top--];
    }

    public int peek() {
        if (isEmpty()) {
            throw new EmptyStackException("Stack is empty");
        }
        return items[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    private boolean isFull() {
        return top == items.length - 1;
    }

    public static void main(String[] args) {
        CustomStack stack = new CustomStack(10);

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Stack elements (LIFO):");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

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.

```java
public class CustomQueue<T> {

    private Node<T> front;
    private Node<T> rear;
    private int size;

    private static class Node<T> {
        T data;
        Node<T> next;

        public Node(T data) {
            this.data = data;
        }
    }


    public void enqueue(T data) {
        Node<T> newNode = new Node<>(data);
        if (isEmpty()) {
            front = rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
        size++;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new EmptyStackException("Queue is empty");
        }
        T data = front.data;
        front = front.next;
        if (front == null) {
            rear = null;
        }
        size--;
        return data;
    }

    public T peek() {
        if (isEmpty()) {
            throw new EmptyStackException("Queue is empty");
        }
        return front.data;
    }

    public boolean isEmpty() {
        return front == null;
    }

    public int getSize() {
        return size;
    }
    public static void main(String[] args) {
        CustomQueue<Object> queue = new CustomQueue<>();

        queue.enqueue("apple");
        queue.enqueue(20);
        queue.enqueue("banana");
        queue.enqueue(30);

        System.out.println("Queue elements (FIFO):");

        while (!queue.isEmpty()) {
            System.out.println(queue.dequeue());
        }
    }
}
```

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.

```
public class EmergencyRoomPriorityQueue {

    private Node[] patients;
    private int size;
    private static final int DEFAULT_CAPACITY = 10;

    private static class Node {
        Patient patient;
        int priority;

        public Node(Patient patient, int priority) {
            this.patient = patient;
            this.priority = priority;
        }
    }

    public EmergencyRoomPriorityQueue() {
        patients = new Node[DEFAULT_CAPACITY];
    }

    private void swap(int index1, int index2) {
        Node temp = patients[index1];
        patients[index1] = patients[index2];
        patients[index2] = temp;
    }

    public void enqueue(Patient patient, int priority) {
        if (size == patients.length) {
            resize();
        }
        Node newNode = new Node(patient, priority);
        patients[size] = newNode;
        size++;
        siftUp(size - 1);
    }

    private void resize() {
        Node[] newArray = new Node[patients.length * 2];
        System.arraycopy(patients, 0, newArray, 0, patients.length);
        patients = newArray;
    }

    private void siftUp(int index) {
        while (index > 0 && patients[parent(index)].priority <
patients[index].priority) {
            swap(index, parent(index));
            index = parent(index);
        }
    }

    private int parent(int index) {
        return (index - 1) / 2;
    }

    public Patient dequeue() {
        if (isEmpty()) {
            throw new EmptyStackException("Queue is empty");
        }
        Patient topPatient = patients[0].patient;
        swap(0, size - 1);
        size--;
        siftDown(0);
        return topPatient;
    }

    private void siftDown(int index) {
        int leftChild = leftChild(index);
        int rightChild = rightChild(index);
        int largest = index;
        if (leftChild < size && patients[leftChild].priority >
patients[largest].priority) {
            largest = leftChild;
        }
        if (rightChild < size && patients[rightChild].priority >
patients[largest].priority) {
            largest = rightChild;
        }
        if (largest != index) {
            swap(index, largest);
            siftDown(largest);
        }
    }

    private int leftChild(int index) {
        return 2 * index + 1;
    }

    private int rightChild(int index) {
        return 2 * index + 2;
    }

    public boolean isEmpty() {
        return size == 0;
    }

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

```
public class Patient {
    String name;
    String condition;
}
```