prajwalhonashetti143@gmail.com

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.

```
// Node class definition
class Node {
   int data;
   Node next;

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

// Custom linked list class definition
public class CustomLinkedList {
   private Node head;

   public CustomLinkedList() {
      this.head = null;
   }

   // Method to insert a node at the beginning of the 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 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;
    }
}

// Method to insert a node at a specific position in the list
public void InsertAtPosition(int data, int position) {
    Node newNode = new Node(data);
    if (position == 0) {
        newNode.next = head;
        head = newNode;
        return;
    }

    Node temp = head;
    for (int i = 0; i < position - 1 && temp != null; i++) {
        temp = temp.next;
    }

    if (temp == null) {
        throw new IndexOutOfBoundsException("Position out of bounds");
    }

    newNode.next = temp.next;
    temp.next = newNode;
}

// Method to delete a node by value
public void DeleteNode(int key) {
    if (head == null) {
        return;
    }

    if (head.data == key) {
        head = head.next;
        return;
    }

    Node temp = head;
    while (temp.next != null && temp.next.data != key) {
        temp = temp.next;
    }
    if (temp.next == null) {
        System.out.println("Node with data " + key + " not found.");
        return;
    }

    temp.next = temp.next.next;
}

// Method to update a node's value
public void UpdateNode(int oldData, int newData) {
    Node temp = head;
    while (temp != null && temp.data != oldData) {
        temp = temp.next;
    }

    if (temp != null) {
        temp.data = newData;
    } else {
        System.out.println("Node with data " + oldData + " not found.");
    }
}

// Method to display all nodes in the list
public void DisplayAllNodes() {
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}

// Main method for testing
public static void main(String[] args) {
    // Create a new custom linked list
    CustomLinkedList list = new CustomLinkedList();

    // Insertions
    list.InsertAtBeginning(1);
    list.InsertAtEnd(2);
    list.InsertAtEnd(3);
    list.InsertAtPosition(4, 2);
    System.out.print("List after insertions: ");
```

```
        list.DisplayAllNodes();

        // Updates
        list.UpdateNode(4, 5);
        System.out.print("List after updating 4 to 5: ");
        list.DisplayAllNodes();
        // Deletions
        list.DeleteNode(2);
        System.out.print("List after deleting 2: ");
        list.DisplayAllNodes();
        list.DeleteNode(5);
        System.out.print("List after deleting 5: ");
        list.DisplayAllNodes();
        list.DeleteNode(1);
        System.out.print("List after deleting 1: ");
        list.DisplayAllNodes();
    }
}
```

Explanation

Node Class:

- Represents a node in the linked list.
- Contains data (integer value) and next (pointer to the next node).

CustomLinkedList Class:

- Contains a head pointer to the first node in the list.
- **InsertAtBeginning(int data):** Inserts a new node at the beginning.
- **InsertAtEnd(int data):** Inserts a new node at the end.
- **InsertAtPosition(int data, int position):** Inserts a new node at a specific position.
- **DeleteNode(int key):** Deletes the first node with the specified data.
- **UpdateNode(int oldData, int newData):** Updates a node's data.
- **DisplayAllNodes():** Displays all nodes in the list.

Main Method:

- Demonstrates the usage of the **CustomLinkedList** methods.
- Performs a series of insertions, updates, and deletions.
- Displays the state of the list after each operation to verify correctness.

Step-by-Step Output Explanation

After Insertions:

- Insert 1 at the beginning: 1 -> null
- Insert 2 at the end: 1 -> 2 -> null
- Insert 3 at the end: 1 -> 2 -> 3 -> null
- Insert 4 at position 2: 1 -> 2 -> 4 -> 3 -> null

The output shows the list after all insertions: 1 -> 2 -> 4 -> 3 -> null

After Updating Node 4 to 5:

- Update node with data 4 to 5: 1 -> 2 -> 5 -> 3 -> null
- The output shows the list after the update: 1 -> 2 -> 5 -> 3 -> null

List after updating 4 to 5: 1 -> 2 -> 5 -> 3 -> null

After Deleting Node with Data 2:

- Delete node with data 2: 1 -> 5 -> 3 -> null
- The output shows the list after deleting node 2: 1 -> 5 -> 3 -> null

List after deleting 2: 1 -> 5 -> 3 -> null

After Deleting Node with Data 5:

- Delete node with data 5: 1 -> 3 -> null
- The output shows the list after deleting node 5: 1 -> 3 -> null

List after deleting 5: 1 -> 3 -> null

After Deleting Node with Data 1:

- Delete node with data 1: 3 -> null
- The output shows the list after deleting node 1: 3 -> null

List after deleting 1: 3 -> 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.

```
import java.util.LinkedList;
public class CustomStack {
   private LinkedList<Integer> stack;

   // Constructor
   public CustomStack() {
      stack = new LinkedList<>();
   }

   // Push method to add an element to the stack
   public void Push(int value) {
      stack.push(value);
   }

   // Pop method to remove and return the top element of the stack
   public int Pop() {
      if (stack.isEmpty()) {
         throw new IllegalStateException("Stack is empty");
      }
      return stack.pop();
   }

   // Peek method to return the top element of the stack without removing it
   public int Peek() {
      if (stack.isEmpty()) {
         throw new IllegalStateException("Stack is empty");
      }
      return stack.peek();
   }

   // IsEmpty method to check if the stack is empty
   public boolean IsEmpty() {
      return stack.isEmpty();
   }

   // Main method to demonstrate the LIFO behavior
   public static void main(String[] args) {
      CustomStack stack = new CustomStack();
```

```
        // Push integers onto the stack
        stack.Push(1);
        stack.Push(2);
        stack.Push(3);
        stack.Push(4);
        stack.Push(5);

        // Display and pop elements until the stack is empty
        while (!stack.IsEmpty()) {
            System.out.println("Top element is: " + stack.Peek());
            System.out.println("Popped element: " + stack.Pop());
        }
    }
}
```

Explanation

CustomStack Class:

- The stack is implemented using a LinkedList to store the elements.
- **Push(int value**): Adds an element to the top of the stack.
- **Pop():** Removes and returns the top element of the stack. Throws an exception if the stack is empty.
- **Peek():** Returns the top element of the stack without removing it. Throws an exception if the stack is empty.
- **IsEmpty():** Checks if the stack is empty.

Main Method:

- Creates a **CustomStack** instance.
- Pushes integers 1 to 5 onto the stack.
- Uses a while loop to pop and display elements until the stack is empty, demonstrating the LIFO behavior.

Output

```
Top element is: 5
Popped element: 5
Top element is: 4
Popped element: 4
Top element is: 3
Popped element: 3
Top element is: 2
Popped element: 2
Top element is: 1
Popped element: 1
```

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.**

```
import java.util.LinkedList;

public class CustomQueue<T> {

    private LinkedList<T> queue;



    // Constructor

    public CustomQueue() {

        queue = new LinkedList<>();

    }



    // Enqueue method to add an element to the end of the queue

    public void Enqueue(T value) {

        queue.addLast(value);

    }



    // Dequeue method to remove and return the front element of the queue

    public T Dequeue() {

        if (queue.isEmpty()) {

            throw new IllegalStateException("Queue is empty");

        }

        return queue.removeFirst();

    }



    // Peek method to return the front element of the queue without removing it

    public T Peek() {

        if (queue.isEmpty()) {
```

```
        throw new IllegalStateException("Queue is empty");
    }
    return queue.getFirst();
}

// IsEmpty method to check if the queue is empty
public boolean IsEmpty() {
    return queue.isEmpty();
}

// Main method to demonstrate the FIFO behavior with different data types
public static void main(String[] args) {
    // Create a CustomQueue for integers
    CustomQueue<Integer> intQueue = new CustomQueue<>();
    intQueue.Enqueue(1);
    intQueue.Enqueue(2);
    intQueue.Enqueue(3);

    System.out.println("Integer Queue:");
    while (!intQueue.IsEmpty()) {
        System.out.println("Front element is: " + intQueue.Peek());
        System.out.println("Dequeued element: " + intQueue.Dequeue());
    }

    // Create a CustomQueue for strings
    CustomQueue<String> stringQueue = new CustomQueue<>();
    stringQueue.Enqueue("A");
    stringQueue.Enqueue("B");
    stringQueue.Enqueue("C");
```

```
        System.out.println("\nString Queue:");
        while (!stringQueue.IsEmpty()) {
            System.out.println("Front element is: " + stringQueue.Peek());
            System.out.println("Dequeued element: " + stringQueue.Dequeue());
        }
    }
}
```

Explanation

CustomQueue Class:

- The queue is implemented using a LinkedList to store the elements.
- The class is generic (<T>) to allow it to handle different data types.
- **Enqueue(T value):** Adds an element to the end of the queue.
- **Dequeue(): R**emoves and returns the front element of the queue. Throws an exception if the queue is empty.
- **Peek():** Returns the front element of the queue without removing it. Throws an exception if the queue is empty.
- **IsEmpty():** Checks if the queue is empty.

Main Method:

- Demonstrates the usage of the **CustomQueue** with integers and strings.
- Creates an **intQueue** instance and enqueues integers 1 to 3.
- Dequeues and displays elements from **intQueue** until it is empty, demonstrating FIFO behavior.
- Creates a **stringQueue** instance and enqueues strings "A" to "C".
- Dequeues and displays elements from **stringQueue** until it is empty, demonstrating FIFO behavior.

Output

- **Integer Queue:**
- **Front element is: 1**
- **Dequeued element: 1**
- **Front element is: 2**
- **Dequeued element: 2**
- **Front element is: 3**
- **Dequeued element: 3**

- **String Queue:**
- **Front element is: A**
- **Dequeued element: A**
- **Front element is: B**
- **Dequeued element: B**
- **Front element is: C**
- **Dequeued element: C**

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.**

Patient Class

```
public 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 + '}';
    }
}
```

CustomPriorityQueue Class

```
import java.util.ArrayList;
import java.util.List;
```

```
public class CustomPriorityQueue {
  private List<Patient> queue;

  public CustomPriorityQueue() {
    queue = new ArrayList<>();
  }

  // Enqueue method to add a patient to the queue
  public void Enqueue(Patient patient) {
    queue.add(patient);
    queue.sort((p1, p2) -> Integer.compare(p2.getUrgency(), p1.getUrgency()));
  }

  // Dequeue method to remove and return the highest priority patient
  public Patient Dequeue() {
    if (queue.isEmpty()) {
      throw new IllegalStateException("Queue is empty");
    }
    return queue.remove(0);
  }

  // Peek method to return the highest priority patient without removing them
  public Patient Peek() {
    if (queue.isEmpty()) {
      throw new IllegalStateException("Queue is empty");
    }
    return queue.get(0);
  }

  // IsEmpty method to check if the queue is empty
  public boolean IsEmpty() {
    return queue.isEmpty();
  }

  // Main method to demonstrate the priority queue
  public static void main(String[] args) {
    CustomPriorityQueue emergencyRoom = new CustomPriorityQueue();

    // Add patients to the queue
    emergencyRoom.Enqueue(new Patient("Alice", 2));
    emergencyRoom.Enqueue(new Patient("Bob", 5));
    emergencyRoom.Enqueue(new Patient("Charlie", 1));
    emergencyRoom.Enqueue(new Patient("David", 4));
    emergencyRoom.Enqueue(new Patient("Eve", 3));
```

```
        // Serve patients based on priority
        System.out.println("Serving patients in order of priority:");
        while (!emergencyRoom.IsEmpty()) {
            System.out.println(emergencyRoom.Dequeue());
        }
    }
}
```

Explanation

Patient Class:

- Represents a patient with a name and an urgency level.
- Contains getters for **name** and **urgency**.
- Overrides **toString** for a readable representation.

CustomPriorityQueue Class:

- Uses an **ArrayList** to manage patients.
- **Enqueue**(Patient patient): Adds a patient to the queue and sorts the list based on urgency, ensuring that the highest urgency patients are at the front.
- **Dequeue**(): Removes and returns the patient with the highest urgency (first in the list). Throws an exception if the queue is empty.
- **Peek**(): Returns the patient with the highest urgency without removing them. Throws an exception if the queue is empty.
- **IsEmpty**(): Checks if the queue is empty.

Main Method:

- Demonstrates the priority queue by adding several patients with different urgency levels.
- Serves (dequeues) patients in order of priority until the queue is empty, showing that higher urgency patients are served first.

Output

- **Serving patients in order of priority:**
- **Patient{name='Bob', urgency=5}**
- **Patient{name='David', urgency=4}**
- **Patient{name='Eve', urgency=3}**
- **Patient{name='Alice', urgency=2}**
- **Patient{name='Charlie', urgency=1}**