

## Linked List, Stack, Queue - Assignment

### 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.

Code:

```
package linkedlist;
```

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

```
public class CustomLinkedList {  
    private Node head;
```

```
    public void insertAtBeginning(int data) {  
        Node newNode = new Node(data);  
        newNode.next = head;  
        head = newNode;  
    }
```

```
    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;
}
```

```
public void insertAtPosition(int data, int position) {
if (position < 0) {
System.out.println("Invalid position");
return;
}
if (position == 0) {
insertAtBeginning(data);
return;
}
Node newNode = new Node(data);
Node current = head;
for (int i = 0; i < position - 1 && current != null; i++) {
current = current.next;
}
if (current == null) {
System.out.println("Invalid position");
return;
}
newNode.next = current.next;
current.next = newNode;
}
```

```
public void deleteNode(int data) {
if (head == null) {
System.out.println("List is empty");
```

```

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

```

```

public void updateNode(int position, int newData) {
    Node current = head;
    for (int i = 0; i < position && current != null; i++) {
        current = current.next;
    }
    if (current == null) {
        System.out.println("Invalid position");
        return;
    }
    current.data = newData;
}

```

```

public void displayAllNodes() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " ");
        current = current.next;
    }
}

```

```

    }
    System.out.println();
}

public static void main(String[] args) {
    CustomLinkedList list = new CustomLinkedList();

    list.insertAtBeginning(10);
    list.displayAllNodes();
    list.insertAtBeginning(20);
    list.displayAllNodes();

    list.insertAtEnd(30);
    list.displayAllNodes();

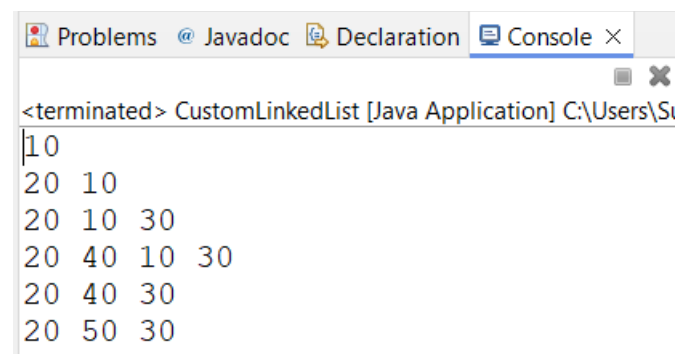
    list.insertAtPosition(40, 1);
    list.displayAllNodes();

    list.deleteNode(10);
    list.displayAllNodes();

    list.updateNode(1, 50);
    list.displayAllNodes();
}
}

```

### Output:



```

<terminated> CustomLinkedList [Java Application] C:\Users\St...
10
20 10
20 10 30
20 40 10 30
20 40 30
20 50 30

```

## Task 2: Stack and Queue Operations

a) 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.

Code:

```
package stack;  
import java.util.EmptyStackException;
```

```
public class CustomStack {
```

```
    private static class Node {
```

```
        int data;
```

```
        Node next;
```

```
        Node(int data) {
```

```
            this.data = data;
```

```
        }
```

```
    }
```

```
    private Node top;
```

```
    public void push(int data) {
```

```
        Node newNode = new Node(data);
```

```
        newNode.next = top;
```

```
        top = newNode;
```

```
    }
```

```
    public int pop() {
```

```
        if (isEmpty()) {
```

```
            throw new EmptyStackException();
```

```
        }
```

```
        int data = top.data;
```

```
        top = top.next;
```

```
        return data;
```

```
}
```

```
public int peek() {  
    if (isEmpty()) {  
        throw new EmptyStackException();  
    }  
    return top.data;  
}
```

```
public boolean isEmpty() {  
    return top == null;  
}
```

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

```
    stack.push(10);  
    System.out.println("Top element after insertion: " + stack.peek());  
    stack.push(20);  
    System.out.println("Top element after insertion: " + stack.peek());  
    stack.push(30);  
    System.out.println("Top element after insertion: " + stack.peek());
```

```
    while (!stack.isEmpty()) {  
        System.out.println("Popped: " + stack.pop());  
    }  
}
```

**Output:**

---

```
Top element after insertion: 10
Top element after insertion: 20
Top element after insertion: 30
Popped: 30
Popped: 20
Popped: 10
```

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

**Code:**

```
package queue;

import java.util.LinkedList;
import java.util.Queue;

public class CustomQueue<T> {
    private Queue<T> queue;

    public CustomQueue() {
        this.queue = new LinkedList<>();
    }

    public void enqueue(T data) {
        queue.offer(data);
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        return queue.poll();
    }
}
```

```
public T peek() {  
    if (isEmpty()) {  
        throw new IllegalStateException("Queue is empty");  
    }  
    return queue.peek();  
}
```

```
public boolean isEmpty() {  
    return queue.isEmpty();  
}  
  
public void displayAll() {  
    for (T element : queue) {  
        System.out.print(element + " ");  
    }  
    System.out.println();  
}  
  
public int size() {  
    return queue.size();  
}
```

```
public static void main(String[] args) {  
    CustomQueue<Object> queue = new CustomQueue<>();  
    queue.enqueue("First");  
    queue.enqueue(2);  
    queue.enqueue("Third");
```

```
    System.out.print("All elements in the queue: ");  
    queue.displayAll();
```

```
    System.out.println("Dequeued: " + queue.dequeue());  
    System.out.println("Peeked: " + queue.peek());
```

```
    System.out.println("Size of the queue: " + queue.size());  
    while (!queue.isEmpty()) {
```



```
System.out.println("Dequeued: " + queue.dequeue());  
}  
System.out.println("Size of the queue after queue is empty: " + queue.size());  
}  
}
```

### Output:

```
All elements in the queue: First 2 Third  
Dequeued: First  
Peeked: 2  
Size of the queue: 2  
Dequeued: 2  
Dequeued: Third  
Size of the queue after queue is empty: 0
```

### 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.

#### Code:

```
package priorityqueue;  
  
import java.util.PriorityQueue;  
import java.util.Comparator;  
  
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 + ", Urgency: " + urgency;
}
}

```

```

public class PriorityQueueEmergencyEx {
public static void main(String[] args) {
Comparator<Patient> urgencyComparator =
Comparator.comparingInt(Patient::getUrgency).reversed();

```

```

PriorityQueue<Patient> priorityQueue = new PriorityQueue<>(urgencyComparator);
priorityQueue.offer(new Patient("John", 5));
priorityQueue.offer(new Patient("Alice", 10));
priorityQueue.offer(new Patient("Bob", 7));
priorityQueue.offer(new Patient("Emily", 3));

```

```

while (!priorityQueue.isEmpty()) {
System.out.println("Serving: " + priorityQueue.poll());
}
}
}

```

### Output:

---

```

Serving: Patient: Alice, Urgency: 10
Serving: Patient: Bob, Urgency: 7
Serving: Patient: John, Urgency: 5
Serving: Patient: Emily, Urgency: 3

```