LinkedList - Day 2

Algorithm - Singly LinkedList

Insertion at the beginning of the linked list:
1. Create a new node with the given data.
2. Set the "next" pointer of the new node to the current head of the linked list.
3. Update the head of the linked list to point to the new node.

Insertion at the end of the linked list:
1. Create a new node with the given data.
2. If the linked list is empty (head is null), set the new node as the head.
3. Otherwise, traverse the linked list to the last node.
4. Set the "next" pointer of the last node to the new node.

Insertion at a specific position in the linked list:
1. Create a new node with the given data.
2. If the position is less than or equal to 0, perform "Insertion at the beginning" (see step 1 above).
3. Traverse the linked list until reaching the node at the (position - 1)th position.
4. Set the "next" pointer of the new node to the "next" pointer of the current node at the (position - 1)th position.
5. Set the "next" pointer of the current node at the (position - 1)th position to the new node.

Deletion at the beginning of the linked list:
1. If the linked list is empty (head is null), indicate that there are no elements to delete.
2. Otherwise, update the head of the linked list to the next node after the current head.

Deletion at the end of the linked list:
1. If the linked list is empty (head is null), indicate that there are no elements to delete.
2. If the linked list contains only one node (head's "next" pointer is null), set the head to null.
3. Otherwise, traverse the linked list to the second-to-last node.
4. Set the "next" pointer of the second-to-last node to null.

Deletion at a specific position in the linked list:
1. If the linked list is empty (head is null), indicate that there are no elements to delete.
2. If the position is less than or equal to 0, perform "Deletion at the beginning" (see step 1 above).
3. Traverse the linked list until reaching the node at the (position - 1)th position.
4. If the current node at the (position - 1)th position is null or its "next" pointer is null, indicate that the position is out of range.
5. Otherwise, set the "next" pointer of the current node at the (position - 1)th position to the "next" pointer of the next node.

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

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

public class SinglyLinkedList {
    private Node head;

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

    // Insertion at the end of the linked list
    public void insertAtEnd(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    // Insertion at a specific position in the linked list
    public void insertAtPosition(int data, int position) {
        if (position <= 0) {
            insertAtBeginning(data);
        } else {
            Node newNode = new Node(data);
            Node current = head;
            Node prev = null;
            int count = 0;
            while (current != null && count < position) {
                prev = current;
                current = current.next;
                count++;
            }
            prev.next = newNode;
            newNode.next = current;
        }
    }

    // Deletion at the beginning of the linked list
    public void deleteAtBeginning() {
```

```
    if (head != null) {
        head = head.next;
    } else {
        System.out.println("Linked list is empty. No elements to delete.");
    }
}

// Deletion at the end of the linked list
public void deleteAtEnd() {
    if (head == null) {
        System.out.println("Linked list is empty. No elements to delete.");
    } else if (head.next == null) {
        head = null;
    } else {
        Node current = head;
        Node prev = null;
        while (current.next != null) {
            prev = current;
            current = current.next;
        }
        prev.next = null;
    }
}

// Deletion at a specific position in the linked list
public void deleteAtPosition(int position) {
    if (head == null) {
        System.out.println("Linked list is empty. No elements to delete.");
    } else if (position <= 0) {
        deleteAtBeginning();
    } else {
        Node current = head;
        Node prev = null;
        int count = 0;
        while (current != null && count < position) {
            prev = current;
            current = current.next;
            count++;
        }
        if (current != null) {
            prev.next = current.next;
        } else {
            System.out.println("Position is out of range.");
        }
    }
}

// Utility method to display the linked list
public void displayLinkedList() {
    Node current = head;
    while (current != null) {
        System.out.print(current.data + " -> ");
        current = current.next;
    }
    System.out.println("null");
```

```
    }

    public static void main(String[] args) {
        SinglyLinkedList linkedList = new SinglyLinkedList();

        // Insertion operations
        linkedList.insertAtBeginning(10);
        linkedList.insertAtEnd(20);
        linkedList.insertAtPosition(15, 1);

        // Display the linked list
        linkedList.displayLinkedList();

        // Deletion operations
        linkedList.deleteAtBeginning();
        linkedList.deleteAtEnd();
        linkedList.deleteAtPosition(1);

        // Display the updated linked list
        linkedList.displayLinkedList();
    }
}
```

Stack using LinkedList:

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

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

public class StackUsingLinkedList {
    private Node top;

    // Constructor to initialize an empty stack
    public StackUsingLinkedList() {
        this.top = null;
    }

    // Check if the stack is empty
    public boolean isEmpty() {
        return top == null;
    }

    // Push an element onto the stack
    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;
        top = newNode;
    }

    // Pop an element from the stack and return its value
    public int pop() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot pop.");
            return -1; // Assuming -1 is not a valid element in the stack
        }

        int data = top.data;
        top = top.next;
        return data;
    }

    // Peek the top element of the stack without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Stack is empty. Cannot peek.");
            return -1; // Assuming -1 is not a valid element in the stack
        }

        return top.data;
    }

    // Utility method to display the stack from top to bottom
```

```
    public void displayStack() {
        Node current = top;
        System.out.print("Stack (top to bottom): ");
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

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

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

        stack.displayStack(); // Should print: Stack (top to bottom): 30 20 10

         System.out.println("Peeked element: " + stack.peek()); // Should print:
Peeked element: 30

        int poppedElement = stack.pop();
         System.out.println("Popped element: " + poppedElement); // Should print:
Popped element: 30

        stack.displayStack(); // Should print: Stack (top to bottom): 20 10
    }
}
```

Queue using LinkedList:

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

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

public class QueueUsingLinkedList {
    private Node front;
    private Node rear;

    // Constructor to initialize an empty queue
    public QueueUsingLinkedList() {
        this.front = null;
        this.rear = null;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }

    // Enqueue (add) an element to the rear of the queue
    public void enqueue(int data) {
        Node newNode = new Node(data);
        if (isEmpty()) {
            front = newNode;
            rear = newNode;
        } else {
            rear.next = newNode;
            rear = newNode;
        }
    }

     // Dequeue (remove) an element from the front of the queue and return its
value
    public int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot dequeue.");
            return -1; // Assuming -1 is not a valid element in the queue
        }

        int data = front.data;
        front = front.next;
        if (front == null) {
              rear = null; // If the queue is now empty, update rear to null as
well
        }
        return data;
    }
```

```
    // Peek the front element of the queue without removing it
    public int peek() {
        if (isEmpty()) {
            System.out.println("Queue is empty. Cannot peek.");
            return -1; // Assuming -1 is not a valid element in the queue
        }

        return front.data;
    }

    // Utility method to display the queue from front to rear
    public void displayQueue() {
        Node current = front;
        System.out.print("Queue (front to rear): ");
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        QueueUsingLinkedList queue = new QueueUsingLinkedList();

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);

        queue.displayQueue(); // Should print: Queue (front to rear): 10 20 30

          System.out.println("Dequeued element: " + queue.dequeue()); // Should
print: Dequeued element: 10

        queue.displayQueue(); // Should print: Queue (front to rear): 20 30

         System.out.println("Peeked element: " + queue.peek()); // Should print:
Peeked element: 20
    }
}
```