```java
//Saket M Kharche
// Class representing a Singly Linked List
class LinkedList {
    Node head; // Reference to the first node (head) of the linked list

    // 🟢 Inner class representing a node in the linked list
    class Node {
        int data;  // Stores the value of the node
        Node next; // Pointer to the next node in the list

        // Constructor to create a new node with given data
        Node(int data) {
            this.data = data;  // Assign data to the node
            this.next = null;  // Initially, the next pointer is set to null
        }
    }

    // 1️⃣ Insert a new node at the beginning of the linked list (O(1))
    public void insertAtBeginning(int data) {
        Node newNode = new Node(data); // Step 1: Create a new node with the given data
        newNode.next = head; // Step 2: New node points to the current head (previous first node)
        head = newNode; // Step 3: Update the head to point to the new node
    }

    // 2️⃣ Insert a new node at the end of the linked list (O(n))
    public void insertAtEnd(int data) {
        Node newNode = new Node(data); // Step 1: Create a new node with given data
        if (head == null) { // Step 2: If the list is empty, set new node as the head
            head = newNode;
            return; // Exit method
        }
        Node temp = head; // Step 3: Start from the head
        while (temp.next != null) { // Step 4: Traverse to the last node
            temp = temp.next;
        }
        temp.next = newNode; // Step 5: Set last node's next pointer to the new node
    }

    // 3️⃣ Insert a new node at a specific position (O(n))
    public void insertAtPosition(int data, int position) {
        if (position == 0) { // If position is 0, insert at the beginning
            insertAtBeginning(data);
            return;
        }
        Node newNode = new Node(data); // Step 1: Create a new node
        Node temp = head; // Step 2: Start from the head

        // Step 3: Traverse the list to find the (position - 1)th node
        for (int i = 0; i < position - 1 && temp != null; i++) {
            temp = temp.next;
        }
        if (temp == null) return; // If position is out of bounds, do nothing

        // Step 4: Insert new node in between
        newNode.next = temp.next; // Set new node's next to point to the current next node
        temp.next = newNode; // Update previous node to point to the new node
    }

    // 4️⃣ Delete a node by value (O(n))
    public void deleteByValue(int key) {
        if (head == null) return; // If the list is empty, nothing to delete
        if (head.data == key) { // Step 1: If the head node has the key, delete it
            head = head.next; // Move head to the next node
            return;
        }
        Node temp = head; // Step 2: Start from head

        // Step 3: Traverse to find the node just before the node to be deleted
        while (temp.next != null && temp.next.data != key) {
            temp = temp.next;
        }
        if (temp.next == null) return; // If key is not found, do nothing

        // Step 4: Remove the node from the linked list
        temp.next = temp.next.next;
    }

    // 5️⃣ Delete the first node (O(1))
    public void deleteFromBeginning() {
        if (head != null) {
            head = head.next; // Move head to the next node
        }
    }

    // 6️⃣ Delete the last node (O(n))
    public void deleteFromEnd() {
        if (head == null || head.next == null) { // If list is empty or has only one node
            head = null;
            return;
        }
        Node temp = head; // Step 1: Start from head
        while (temp.next.next != null) { // Step 2: Traverse to the second-last node
            temp = temp.next;
        }
        temp.next = null; // Step 3: Remove the last node by setting next to null
    }

    // 7️⃣ Display the linked list (O(n))
    public void display() {
        Node temp = head; // Start from head
        while (temp != null) { // Traverse through the list
            System.out.print(temp.data + " -> "); // Print the data of the current node
            temp = temp.next; // Move to the next node
        }
        System.out.println("null"); // Print null to indicate end of the list
    }

    // 8️⃣ Search for an element in the linked list (O(n))
    public boolean search(int key) {
        Node temp = head; // Start from head
        while (temp != null) { // Traverse through the list
            if (temp.data == key) return true; // If key is found, return true
            temp = temp.next; // Move to the next node
        }
        return false; // Key not found
    }

    // 9️⃣ Reverse the linked list (O(n))
    public void reverse() {
        Node prev = null, curr = head, next;
        while (curr != null) { // Traverse through the list
            next = curr.next; // Step 1: Store next node
            curr.next = prev; // Step 2: Reverse the link
            prev = curr; // Step 3: Move prev forward
            curr = next; // Step 4: Move curr forward
        }
        head = prev; // Step 5: Update head to new first node
    }

    // 🔟 Get the length of the linked list (O(n))
    public int length() {
        int count = 0;
        Node temp = head; // Start from head
        while (temp != null) { // Traverse through the list
            count++; // Increase count for each node
            temp = temp.next; // Move to the next node
        }
        return count;
    }

    // Main method to test the linked list
    public static void main(String[] args) {
        LinkedList list = new LinkedList(); // Create a new linked list

        // Insert elements
        list.insertAtEnd(10);
        list.insertAtEnd(20);
        list.insertAtEnd(30);
        System.out.print("Linked List: ");
        list.display(); // Output: 10 -> 20 -> 30 -> null

        // Insert at beginning
        list.insertAtBeginning(5);
        System.out.print("After inserting 5 at beginning: ");
        list.display(); // Output: 5 -> 10 -> 20 -> 30 -> null

        // Insert at position
        list.insertAtPosition(25, 3);
        System.out.print("After inserting 25 at position 3: ");
        list.display(); // Output: 5 -> 10 -> 20 -> 25 -> 30 -> null

        // Delete by value
        list.deleteByValue(20);
        System.out.print("After deleting 20: ");
        list.display(); // Output: 5 -> 10 -> 25 -> 30 -> null

        // Reverse linked list
        list.reverse();
        System.out.print("After reversing: ");
        list.display(); // Output: 30 -> 25 -> 10 -> 5 -> null

        // Search for an element
        System.out.println("Is 25 in the list? " + list.search(25)); // Output: true
        System.out.println("Is 100 in the list? " + list.search(100)); // Output: false

        // Find length
        System.out.println("Length of Linked List: " + list.length()); // Output: 4
    }
}
```