```java
//Easy Problems

// Node class for singly linked list
class ListNode {
    int data;
    ListNode next;
    ListNode(int data) { this.data = data; }
}

// Node class for doubly linked list
class DoublyListNode {
    int data;
    DoublyListNode next, prev;
    DoublyListNode(int data) { this.data = data; }
}

public class LinkedListProblems {

    // 1. Middle of a linked list
    public ListNode findMiddle(ListNode head) {
        if (head == null) return null;
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow; // middle node
    }

    // 2. Reverse a singly linked list
    public ListNode reverseList(ListNode head) {
        ListNode prev = null, curr = head;
        while (curr != null) {
            ListNode nextNode = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextNode;
        }
        return prev;
    }

    // 3. Reverse a doubly linked list
    public DoublyListNode reverseDoublyList(DoublyListNode head) {
        DoublyListNode curr = head, prev = null;
        while (curr != null) {
            DoublyListNode nextNode = curr.next;
            curr.next = prev;
            curr.prev = nextNode;
            prev = curr;
            curr = nextNode;
        }
        return prev;
    }

    // 4. Rotate a linked list by k nodes
    public ListNode rotateList(ListNode head, int k) {
        if (head == null || k == 0) return head;
        ListNode curr = head;
        int len = 1;
        while (curr.next != null) {
            curr = curr.next;
            len++;
        }
        k = k % len;
        if (k == 0) return head;

        curr.next = head; // make circular
        ListNode newTail = head;
        for (int i = 0; i < len - k - 1; i++) newTail = newTail.next;
        ListNode newHead = newTail.next;
        newTail.next = null;
        return newHead;
    }

    // 5. Nth node from End
    public ListNode nthFromEnd(ListNode head, int n) {
        ListNode first = head, second = head;
        for (int i = 0; i < n; i++) {
            if (first == null) return null;
            first = first.next;
        }
        while (first != null) {
            first = first.next;
            second = second.next;
        }
        return second;
    }

    // 6. Delete Last Occurrence of key
    public ListNode deleteLastOccurrence(ListNode head, int key) {
        ListNode curr = head, last = null;
        while (curr != null) {
            if (curr.data == key) last = curr;
            curr = curr.next;
        }
        if (last == null) return head; // key not found
        if (last == head && head.next == null) return null;
        curr = head;
        ListNode prev = null;
        while (curr != last) {
            prev = curr;
            curr = curr.next;
        }
        if (prev != null) prev.next = curr.next;
        else head = curr.next; // last is head
        return head;
    }

    // 7. Delete middle node (given head)
    public ListNode deleteMiddle(ListNode head) {
        if (head == null || head.next == null) return null;
        ListNode slow = head, fast = head, prev = null;
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        prev.next = slow.next;
        return head;
    }

    // 8. Merge alternate positions of two lists
    public ListNode mergeAlternate(ListNode first, ListNode second) {
        ListNode head1 = first, head2 = second;
        while (head1 != null && head2 != null) {
            ListNode next1 = head1.next, next2 = head2.next;
            head1.next = head2;
            if (next1 == null) break;
            head2.next = next1;
            head1 = next1;
            head2 = next2;
        }
        return first;
    }

    // 9. Circular list traversal
    public void traverseCircularList(ListNode head) {
        if (head == null) return;
        ListNode curr = head;
        do {
            System.out.print(curr.data + " ");
            curr = curr.next;
        } while (curr != head);
        System.out.println();
    }

    // 10. Queue using linked list
    class QueueLL {
        ListNode front, rear;

        public void enqueue(int x) {
            ListNode node = new ListNode(x);
            if (rear != null) rear.next = node;
            rear = node;
            if (front == null) front = rear;
        }

        public int dequeue() {
            if (front == null) throw new RuntimeException("Queue empty");
            int val = front.data;
            front = front.next;
            if (front == null) rear = null;
            return val;
        }

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

    // 11. Stack using singly linked list
    class StackLL {
        ListNode top;
        public void push(int x) {
            ListNode node = new ListNode(x);
            node.next = top;
            top = node;
        }
        public int pop() {
            if (top == null) throw new RuntimeException("Stack empty");
            int val = top.data;
            top = top.next;
            return val;
        }
        public boolean isEmpty() { return top == null; }
    }

    // 12. Pairwise swap nodes
    public ListNode pairwiseSwap(ListNode head) {
        if (head == null || head.next == null) return head;
        ListNode prev = head, curr = head.next;
        head = curr; // new head
        while (true) {
            ListNode next = curr.next;
            curr.next = prev;
            if (next == null || next.next == null) {
                prev.next = next;
                break;
            }
            prev.next = next.next;
            prev = next;
            curr = prev.next;
        }
        return head;
    }

    // 13. Count occurrences of a key
    public int countOccurrences(ListNode head, int key) {
        int count = 0;
        while (head != null) {
            if (head.data == key) count++;
            head = head.next;
        }
        return count;
    }
}


//Medium Problems

// Node class for singly linked list
class ListNode {
    int data;
    ListNode next;
    ListNode(int data) { this.data = data; }
}

// Node class for doubly linked list
class DoublyListNode {
    int data;
    DoublyListNode next, prev;
    DoublyListNode(int data) { this.data = data; }
}

public class LinkedListMediumProblems {

    // 1. Detect Loop in Linked List
    public boolean detectLoop(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) return true;
        }
        return false;
    }

    // 2. Length of the Loop
    public int loopLength(ListNode head) {
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                int length = 1;
                ListNode temp = slow;
                while (temp.next != slow) {
                    length++;
                    temp = temp.next;
                }
                return length;
            }
        }
        return 0;
    }

    // 3. Reverse in groups of k
    public ListNode reverseInGroups(ListNode head, int k) {
        if (head == null) return null;
        ListNode curr = head, prev = null, next = null;
        int count = 0;
        while (curr != null && count < k) {
            next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
            count++;
        }
        if (next != null) head.next = reverseInGroups(next, k);
        return prev;
    }

    // 4. Intersection Point of two linked lists
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        if (headA == null || headB == null) return null;
        ListNode a = headA, b = headB;
        while (a != b) {
            a = (a == null) ? headB : a.next;
            b = (b == null) ? headA : b.next;
        }
        return a;
    }

    // 5. Delete a node without head pointer
    public void deleteWithoutHead(ListNode node) {
        if (node == null || node.next == null) return;
        node.data = node.next.data;
        node.next = node.next.next;
    }

    // 6. Merge two sorted linked lists
    public ListNode mergeSorted(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0), tail = dummy;
        while (l1 != null && l2 != null) {
            if (l1.data < l2.data) {
                tail.next = l1; l1 = l1.next;
            } else {
                tail.next = l2; l2 = l2.next;
            }
            tail = tail.next;
        }
        tail.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }

    // 7. Sort a List of 0s, 1s, and 2s
    public ListNode sort012(ListNode head) {
        int[] count = new int[3];
        ListNode curr = head;
        while (curr != null) {
            count[curr.data]++;
            curr = curr.next;
        }
        curr = head;
        for (int i = 0; i < 3; i++) {
            while (count[i]-- > 0) {
                curr.data = i;
                curr = curr.next;
            }
        }
        return head;
    }

    // 8. Check Palindrome Linked List
    public boolean isPalindrome(ListNode head) {
        if (head == null) return true;
        ListNode slow = head, fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode secondHalf = reverseList(slow);
        ListNode firstHalf = head;
        while (secondHalf != null) {
            if (firstHalf.data != secondHalf.data) return false;
            firstHalf = firstHalf.next;
            secondHalf = secondHalf.next;
```

```java
        }
        return true;
    }

    private ListNode reverseList(ListNode head) {
        ListNode prev = null, curr = head;
        while (curr != null) {
            ListNode nextNode = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextNode;
        }
        return prev;
    }

    // 9. Remove all occurrences of a given value
    public ListNode removeAll(ListNode head, int key) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy, curr = head;
        while (curr != null) {
            if (curr.data == key) prev.next = curr.next;
            else prev = curr;
            curr = curr.next;
        }
        return dummy.next;
    }

    // 10. Split a Circular Linked List into two halves
    public ListNode[] splitCircularList(ListNode head) {
        if (head == null || head.next == head) return new ListNode[]{head, null};
        ListNode slow = head, fast = head;
        while (fast.next != head && fast.next.next != head) {
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode head1 = head;
        ListNode head2 = slow.next;
        slow.next = head1;
        ListNode temp = head2;
        while (temp.next != head) temp = temp.next;
        temp.next = head2;
        return new ListNode[]{head1, head2};
    }

    // 11. Pair Sum in Doubly Linked List
    public void pairSum(DoublyListNode head, int sum) {
        DoublyListNode first = head;
        DoublyListNode second = head;
        while (second.next != null) second = second.next;
        while (first != null && second != null && first != second && second.next != first
        ) {
            int s = first.data + second.data;
            if (s == sum) {
                System.out.println("(" + first.data + "," + second.data + ")");
                first = first.next; second = second.prev;
            } else if (s < sum) first = first.next;
            else second = second.prev;
        }
    }

    // 12. Add two numbers represented by linked lists
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0), curr = dummy;
        int carry = 0;
        while (l1 != null || l2 != null || carry != 0) {
            int sum = carry;
            if (l1 != null) { sum += l1.data; l1 = l1.next; }
            if (l2 != null) { sum += l2.data; l2 = l2.next; }
            carry = sum / 10;
            curr.next = new ListNode(sum % 10);
            curr = curr.next;
        }
        return dummy.next;
    }

    // 13. Multiply two numbers represented by linked lists
    public ListNode multiplyTwoNumbers(ListNode l1, ListNode l2) {
        int num1 = 0, num2 = 0;
        ListNode temp = l1;
        while (temp != null) { num1 = num1 * 10 + temp.data; temp = temp.next; }
        temp = l2;
        while (temp != null) { num2 = num2 * 10 + temp.data; temp = temp.next; }
        int product = num1 * num2;
        if (product == 0) return new ListNode(0);
        ListNode dummy = new ListNode(0), curr = null;
        while (product > 0) {
            ListNode node = new ListNode(product % 10);
            node.next = curr;
            curr = node;
            product /= 10;
        }
        return curr;
    }

    // 14. Swap Kth nodes from beginning and end
    public ListNode swapKthNode(ListNode head, int k) {
        int len = 0;
        ListNode curr = head;
        while (curr != null) { len++; curr = curr.next; }
        if (k > len) return head;
        if (2*k - 1 == len) return head; // same node
        ListNode prevX = null, currX = head;
        for (int i = 1; i < k; i++) { prevX = currX; currX = currX.next; }
        ListNode prevY = null, currY = head;
        for (int i = 1; i < len - k + 1; i++) { prevY = currY; currY = currY.next; }
        if (prevX != null) prevX.next = currY; else head = currY;
        if (prevY != null) prevY.next = currX; else head = currX;
        ListNode temp = currX.next;
        currX.next = currY.next;
        currY.next = temp;
        return head;
    }

    // 15. Rotate Doubly Linked List by N nodes
    public DoublyListNode rotateDoublyList(DoublyListNode head, int n) {
        if (head == null || n == 0) return head;
        DoublyListNode tail = head;
        int len = 1;
        while (tail.next != null) { tail = tail.next; len++; }
        n = n % len;
        if (n == 0) return head;
        DoublyListNode newTail = head;
        for (int i = 1; i < n; i++) newTail = newTail.next;
        DoublyListNode newHead = newTail.next;
        newTail.next = null;
        newHead.prev = null;
        tail.next = head;
        head.prev = tail;
        return newHead;
    }

    // 16. Binary Tree to Doubly Linked List
    class TreeNode {
        int val;
        TreeNode left, right;
        TreeNode(int val) { this.val = val; }
    }
```

```java
    DoublyListNode headDLL = null, prevDLL = null;
    public void btreeToDLL(TreeNode root) {
        if (root == null) return;
        btreeToDLL(root.left);
        DoublyListNode node = new DoublyListNode(root.val);
        if (prevDLL == null) headDLL = node;
        else { prevDLL.next = node; node.prev = prevDLL; }
        prevDLL = node;
        btreeToDLL(root.right);
    }

    // 17. Linked List from 2D matrix
    public ListNode linkedListFromMatrix(int[][] mat) {
        ListNode dummy = new ListNode(0), tail = dummy;
        for (int[] row : mat) {
            for (int val : row) {
                tail.next = new ListNode(val);
                tail = tail.next;
            }
        }
        return dummy.next;
    }

    // 18. Reverse a Sublist (positions m to n)
    public ListNode reverseSublist(ListNode head, int m, int n) {
```