```
public class LinkedList {
    // Introduction to LinkedList, learn a…
    // Inserting a node in LinkedList
    // Deleting a node in LinkedList
    // Find the length of the linkedlist […
    // Search an element in the LL
    // Middle of a LinkedList
    class Solution {
        public ListNode middleNode(ListNode head) {
            ListNode slow = head;
            ListNode fast = head;
            while (fast != null && fast.next != null) {
                slow = slow.next;
                fast = fast.next.next;
            }
            return slow;
        }
    }

    // Reverse a LinkedList [Iterative]
    class Solution {
        public ListNode reverseList(ListNode head) {
            ListNode curr = head;
            ListNode prev = null;
            while (curr != null) {
                ListNode temp = curr.next;
                curr.next = prev;
                prev = curr;
                curr = temp;
            }
            return prev;
        }
    }

    // Reverse a LL [Recursive]
    class Solution {
        public ListNode reverseList(ListNode head) {
            ListNode curr = head;
            ListNode prev = null;
            while (curr != null) {
                ListNode temp = curr.next;
                curr.next = prev;
                prev = curr;
                curr = temp;
            }
            return prev;
        }

        static Node reverse(Node head) {
```

```
        if (head == null) {
            return head;
        }
        if (head.next == null) {
            return head;
        }
        Node newHeadNode = reverse(head.next);
        head.next.next = head;
        head.next = null;
        return newHeadNode;
    }
}

// Detect a loop in LL
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null) {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head;
        while (fast.next != null && fast.next.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                return true;
            }
        }
        return false;
    }
}

// Find the starting point in LL
public class Solution {
    public ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;
        int flag = 0;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                flag = 1;
                break;
            }
        }
        if (flag == 0) {
            return null;
        }
```

```
        ListNode first = head;
        ListNode second = slow;
        while (first != second) {
            first = first.next;
            second = second.next;
        }
        return first;
    }
}

// Length of Loop in LL
class Solution {

    static int countNodesinLoop(Node head) {
        if (head.next == null) {
            return 0;
        }
        if (head.next == head) {
            return 1;
        }
        Node slow = head, fast = head;
        int count = 1;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast)
                break;
        }
        if (slow != fast) {
            return 0;
        }
        Node temp = slow.next;
        while (temp != fast) {
            temp = temp.next;
            count++;
        }
        return count;
    }
}

// Check if LL is palindrome or not

class Solution {
    public boolean isPalindrome(ListNode head) {
        ListNode slow = head;
        ListNode slowprev = head;

        ListNode fast = head;
        while (fast != null && fast.next != null) {
```

```
            slowprev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        ListNode revHead = reverse(slowprev);
        ListNode start = head;
        ListNode tail = revHead;
        while (start != null) {
            if (start.val != tail.val) {
                return false;
            } else {
                start = start.next;
                tail = tail.next;
            }
        }
        return true;
    }

    public ListNode reverse(ListNode head) {
        ListNode prev = null;
        ListNode nextN = head;
        ListNode curr = head;
        while (curr != null) {
            nextN = curr.next;
            curr.next = prev;
            prev = curr;
            curr = nextN;
        }
        return prev;
    }
}

// Segrregate odd and even nodes in LL
class Solution {
    public ListNode oddEvenList(ListNode head) {
        if (head == null) {
            return null;
        }
        ListNode odd = head;
        if (head.next == null) {
            return head;
        }
        ListNode evenStart = head.next;
        ListNode even = head.next;
        while (odd != null && odd.next != null && even != null && even.next != null) {
            even = odd.next;
            odd.next = odd.next.next;
            odd = odd.next;
            even.next = odd.next;
```

```
            even = even.next;
        }
        odd.next = evenStart;
        return head;
    }
}

// Remove Nth node from the back of the LL
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode start = new ListNode();
        start.next = head;
        ListNode fast = start;
        ListNode slow = start;
        for (int i = 1; i <= n; ++i) {
            fast = fast.next;
        }
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }
        slow.next = slow.next.next;
        return start.next;
    }
}

// Delete the middle node of LL
class Solution {
    public ListNode deleteMiddle(ListNode head) {
        if (head == null || head.next == null) {
            return null;
        }
        ListNode slow = head;
        ListNode fast = head;
        ListNode slowP = head;
        while (fast != null && fast.next != null) {
            slowP = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        slowP.next = slow.next;
        return head;

    }
}

// Sort LL
public class Solution {
    public ListNode sortList(ListNode head) {
```

```
        if (head == null || head.next == null) {
            return head;
        }
        ListNode prev = null, slow = head, fast = head;
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }
        prev.next = null;
        ListNode l1 = sortList(head);
        ListNode l2 = sortList(slow);
        return merge(l1, l2);
    }

    ListNode merge(ListNode l1, ListNode l2) {
        ListNode l = new ListNode(0), p = l;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                p.next = l1;
                l1 = l1.next;
            } else {
                p.next = l2;
                l2 = l2.next;
            }
            p = p.next;
        }
        if (l1 != null) {
            p.next = l1;
        }
        if (l2 != null) {
            p.next = l2;
        }
        return l.next;
    }

}

// Sort a LL of 0's 1's and 2's by
class Solution {
    // Function to sort a linked list of 0s, 1s and 2s.
    static Node segregate(Node head) {
        // add your code here
        if (head == null || head.next == null) {
            return head;
        }
        Node zeroD = new Node(0);
        Node oneD = new Node(0);
        Node twoD = new Node(0);
```

```
        Node zero = zeroD, one = oneD, two = twoD;
        Node curr = head;
        while (curr != null) {
            if (curr.data == 0) {
                zero.next = curr;
                zero = zero.next;
                curr = curr.next;
            } else if (curr.data == 1) {
                one.next = curr;
                one = one.next;
                curr = curr.next;
            } else {
                two.next = curr;
                two = two.next;
                curr = curr.next;
            }
        }
        // Attach three lists
        zero.next = (oneD.next != null)
                ? (oneD.next)
                : (twoD.next);
        one.next = twoD.next;
        two.next = null;
        head = zeroD.next;
        return head;

    }
}

// Find the intersection point of Y LL
public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        int len1 = 0;
        int len2 = 0;
        ListNode currA = headA;
        ListNode currB = headB;
        while (currA != null) {
            len1++;
            currA = currA.next;
        }
        while (currB != null) {
            len2++;
            currB = currB.next;
        }
        ListNode cA = headA;
        ListNode cB = headB;
        if (len1 > len2) {
            int req = len1 - len2;
            while (req != 0) {
```

```
            req--;
            cA = cA.next;
        }
        while (cA != cB) {
            cA = cA.next;
            cB = cB.next;

        }
        return cA;
    }

    else {
        int req = len2 - len1;
        while (req != 0) {
            req--;
            cB = cB.next;
        }
        while (cA != cB) {
            cA = cA.next;
            cB = cB.next;
        }
        return cA;
    }

  }
}

// Add 1 to a number represented by LL
class Solution {
    public static Node reverse(Node head) {
        Node prev = null;
        Node current = head;
        Node next;
        while (current != null) {
            next = current.next; // storing next node
            current.next = prev; // linking current node to previous
            prev = current; // updating prev
            current = next; // updating current
        }
        return prev;
    }

    public static Node addOne(Node head) {
        head = reverse(head); // reversing list to make addition easy
        Node current = head;
        int carry = 1;
        while (carry != 0) {
            current.data += 1;
            if (current.data < 10) {
```

```
                return reverse(head);
            } else {
                current.data = 0;
            }
            if (current.next == null) {
                break;
            } else
                current = current.next;
        }

        current.next = new Node(1); // adding new node for the carried 1
        return reverse(head);
    }
}

// Add 2 numbers in LL
class Solution {
    public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode();
        ListNode temp = dummy;
        int sum = 0, carry = 0;
        while (l1 != null || l2 != null || carry == 1) {
            if (l1 != null) {
                sum += l1.val;
                l1 = l1.next;
            }
            if (l2 != null) {
                sum += l2.val;
                l2 = l2.next;
            }
            sum += carry;
            carry = sum / 10;
            ListNode h = new ListNode(sum % 10);
            temp.next = h;
            temp = h;
            sum = 0;
        }
        return dummy.next;
    }
}

// Reverse LL in group of given size K
class Solution {

    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null)
            return null;
        ListNode current = head;
        ListNode next = null;
```

```
        ListNode prev = null;
        int count = 0;
        next = current.next;
        while (count < k && current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }
        if (next != null) {
            boolean res = checkFurther(next, k);
            if (res == true)
                head.next = reverseKGroup(next, k);
            else
                head.next = next;
        }
        return prev;
    }

    public boolean checkFurther(ListNode head, int k) {
        ListNode curr = head;
        while (k > 0) {
            if (curr != null) {
                k--;
                curr = curr.next;
            } else {
                return false;
            }
        }
        return true;
    }
}

// Rotate a LL
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null) {
            return null;
        }
        if (head.next == null && k >= 0) {
            return head;
        }
        if (k == 0) {
            return head;
        }
        ListNode curr = head;
        ListNode prev = head;
        int len = 0;
```

```
        while (curr != null) {
            len++;
            prev = curr;
            curr = curr.next;
        }
        if (k > len) {
            k = k % len;
        }
        if (k == len || k == 0) {
            return head;
        }
        int req = len - k - 1;
        ListNode cur = head;
        while (req != 0) {
            req--;
            cur = cur.next;
        }
        ListNode newHead = cur.next;
        cur.next = null;
        prev.next = head;
        return newHead;
    }
}

// Flattening of LL
class GfG {
    Node flatten(Node root) {
        if (root == null || root.next == null)
            return root;
        root.next = flatten(root.next);
        root = mergeTwoLists(root, root.next);
        return root;
    }

    Node mergeTwoLists(Node a, Node b) {

        Node temp = new Node(0);
        Node res = temp;

        while (a != null && b != null) {
            if (a.data < b.data) {
                temp.bottom = a;
                temp = temp.bottom;
                a = a.bottom;
            } else {
                temp.bottom = b;
                temp = temp.bottom;
                b = b.bottom;
            }
```

```
        }

        if (a != null)
            temp.bottom = a;
        else
            temp.bottom = b;
        return res.bottom;
    }
}

// Clone a Linked List with random pointer
class Solution {
    public Node copyRandomList(Node head) {
        Node temp = head;
        while (temp != null) {
            Node newNode = new Node(temp.val);
            newNode.next = temp.next;
            temp.next = newNode;
            temp = temp.next.next;
        }
        Node itr = head;
        while (itr != null) {
            if (itr.random != null) {
                itr.next.random = itr.random.next;
            }
            itr = itr.next.next;
        }
        Node dummy = new Node(0);
        itr = head;
        temp = dummy;
        Node fast;
        while (itr != null) {
            fast = itr.next.next;
            temp.next = itr.next;
            itr.next = fast;
            temp = temp.next;
            itr = fast;
        }
        return dummy.next;
    }
}

// Introduction to DLL, learn about st…
// Insert a node in DLL
class GfG {
    // Function to insert a new node at given position in doubly linked list.
    void addNode(Node head_ref, int pos, int data) {
        // Your code here
        Node curr = head_ref;
```

```
        int k = pos;
        while (k != 0) {
            k--;
            curr = curr.next;
        }
        Node ninode = new Node(data);
        Node nfn = curr.next;

        curr.next = ninode;
        ninode.prev = curr;
        if (nfn != null) {
            ninode.next = nfn;

            nfn.prev = ninode;
        }
    }
}

// Delete a node in DLL
class Solution {
    // function returns the head of the linkedlist
    Node deleteNode(Node head, int x) {
        // Your code here
        Node curr = head;
        Node prev = head;
        int k = x - 1;
        if (k == 0) {
            Node nf = head.next;
            nf.prev = null;
            return nf;
        }
        while (k != 0) {
            prev = curr;
            curr = curr.next;
            k--;
        }
        Node nf = curr.next;
        Node nb = prev;

        nb.next = nf;
        if (nf != null) {
            nf.prev = nb;
        }
        return head;

    }
}

// Reverse a DLL
```

```
    public static Node reverseDLL(Node head) {
        // Your code here
        Node temp = null;
        Node current = head;
        while (current != null) {
            temp = current.prev;
            current.prev = current.next;
            current.next = temp;
            current = current.prev;
        }

        if (temp != null) {
            head = temp.prev;
        }
        return head;
    }
}

// Delete all occurrences of a key in doubly LL
class Solution {
    static Node deleteAllOccurOfX(Node head, int x) {
        // Write your code here
        if (head == null) {
            return null;
        }
        Node current = head;
        Node next;
        while (current != null) {
            if (current.data == x) {
                next = current.next;
                head = deleteNode(head, current);
                current = next;
            } else {
                current = current.next;
            }
        }

        return head;
    }

    static Node deleteNode(Node head, Node del) {
        if (head == null || del == null) {
            return null;
        }
        if (head == del) {
            head = del.next;
        }
        if (del.next != null) {
            del.next.prev = del.prev;
```

```
        }
        if (del.prev != null) {
            del.prev.next = del.next;
        }
        del = null;
        return head;
    }
}

// Find pairs with given sum in DLL
class Solution {
    public static ArrayList<ArrayList<Integer>> findPairsWithGivenSum(int target, Node
head) {
        // code here
        ArrayList<ArrayList<Integer>> ls = new ArrayList<>();
        Node back = head;
        Node front = head;
        while (back.next != null) {
            back = back.next;
        }
        while (back != front) {
            if (front.data > back.data) {
                break;
            }
            int sum = front.data + back.data;
            if (sum == target) {
                ArrayList<Integer> ns = new ArrayList<>();
                ns.add(front.data);
                ns.add(back.data);
                ls.add(ns);
                front = front.next;
                back = back.prev;
            } else if (sum < target) {
                front = front.next;
            } else {
                back = back.prev;
            }
        }
        return ls;
    }
}

// Remove duplicates from sorted DLL
class Solution {
    Node removeDuplicates(Node head) {
        // Code Here.
        if (head == null) {
            return null;
        }
```

```
        Node current = head;
        while (current.next != null) {
            if (current.data == current.next.data) {
                deleteNode(head, current.next);
            } else {
                current = current.next;
            }
        }
        return head;
    }

    Node deleteNode(Node head, Node del) {
        if (head == null || del == null) {
            return null;
        }
        if (head == del) {
            head = del.next;
        }
        if (del.next != null) {
            del.next.prev = del.prev;
        }
        if (del.prev != null) {
            del.prev.next = del.next;
        }
        del = null;
        return head;
    }
}

// Reverse LL in group of given size K
class Solution {

    public ListNode reverseKGroup(ListNode head, int k) {
        if (head == null)
            return null;
        ListNode current = head;
        ListNode next = null;
        ListNode prev = null;
        int count = 0;
        next = current.next;
        while (count < k && current != null) {
            next = current.next;
            current.next = prev;
            prev = current;
            current = next;
            count++;
        }
        if (next != null) {
            boolean res = checkFurther(next, k);
```

```
            if (res == true)
                head.next = reverseKGroup(next, k);
            else
                head.next = next;
        }
        return prev;
    }

    public boolean checkFurther(ListNode head, int k) {
        ListNode curr = head;
        while (k > 0) {
            if (curr != null) {
                k--;
                curr = curr.next;
            } else {
                return false;
            }
        }
        return true;
    }
}

// Rotate a LL

class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null) {
            return null;
        }
        if (head.next == null && k >= 0) {
            return head;
        }
        if (k == 0) {
            return head;
        }
        ListNode curr = head;
        ListNode prev = head;
        int len = 0;
        while (curr != null) {
            len++;
            prev = curr;
            curr = curr.next;
        }
        if (k > len) {
            k = k % len;
        }
        if (k == len || k == 0) {
            return head;
        }
```

```
        int req = len - k - 1;
        ListNode cur = head;
        while (req != 0) {
            req--;
            cur = cur.next;
        }
        ListNode newHead = cur.next;
        cur.next = null;
        prev.next = head;
        return newHead;
    }
}

// Flattening of LL
class GfG {
    Node flatten(Node root) {
        // Your code here
        return f(root);
    }

    Node f(Node root) {
        if (root == null || root.next == null) {
            return root;
        }
        Node fNode = f(root.next);
        Node nList = mergeTwolists(root, fNode);
        return nList;
    }

    Node mergeTwolists(Node h1, Node h2) {
        Node h3 = new Node(0);
        Node h4 = h3;
        Node ptr1 = h1;
        Node ptr2 = h2;
        while (ptr1 != null && ptr2 != null) {
            if (ptr1.data <= ptr2.data) {
                h3.bottom = ptr1;
                ptr1 = ptr1.bottom;
                h3 = h3.bottom;
            } else {
                h3.bottom = ptr2;
                ptr2 = ptr2.bottom;
                h3 = h3.bottom;
            }
        }
        while (ptr1 != null) {
            h3.bottom = ptr1;
            ptr1 = ptr1.bottom;
            h3 = h3.bottom;
```

```
        }
        while (ptr2 != null) {
            h3.bottom = ptr2;
            ptr2 = ptr2.bottom;
            h3 = h3.bottom;
        }
        // System.out.println();
        return h4.bottom;
    }
}
// Clone a Linked List with random and…
```