NAME: PRAKASH SINGH

UID : 22BCS16633

SECTION: 611/B

206. Reverse Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode next = current.next; // Store next node
            current.next = prev; // Reverse the link
            prev = current; // Move prev forward
            current = next; // Move current forward
        }

        return prev; // New head of reversed list
    }
}
```

83. Remove Duplicates from Sorted List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode() {}
 *     ListNode(int val) { this.val = val; }
 *     ListNode(int val, ListNode next) { this.val = val; this.next = next; }
 * }
 */
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null) return null; // Edge case: Empty list

        ListNode current = head; // Start from the head

        while (current != null && current.next != null) {
            if (current.val == current.next.val) {
                current.next = current.next.next; // Skip duplicate node
            } else {
                current = current.next; // Move to next distinct node
            }
        }

        return head;
    }
}
```

21. Merge Two Sorted Lists

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode dummy = new ListNode(0); // Dummy node to simplify edge cases
        ListNode current = dummy;

        while (list1 != null && list2 != null) {
            if (list1.val < list2.val) {
                current.next = list1;
                list1 = list1.next;
            } else {
                current.next = list2;
                list2 = list2.next;
            }
            current = current.next;
        }

        // Attach any remaining nodes
        current.next = (list1 != null) ? list1 : list2;

        return dummy.next; // The merged list starts from dummy.next
    }
}
```

[2095. Delete the Middle Node of a Linked List]

```
class Solution {
    public ListNode deleteMiddle(ListNode head) {
        if (head == null || head.next == null) {
            return null; // If there's only one node, return null
        }

        ListNode slow = head, fast = head, prev = null;

        // Move fast pointer twice as fast as slow pointer
        while (fast != null && fast.next != null) {
            prev = slow;
            slow = slow.next;
            fast = fast.next.next;
        }

        // Delete the middle node
        prev.next = slow.next;

        return head;
    }
}
```

[61. Rotate List]

```
class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        if (head == null || head.next == null || k == 0) {
            return head; // No rotation needed
        }

        // Step 1: Compute the length of the linked list
        ListNode temp = head;
        int length = 1; // At least one node is present
        while (temp.next != null) {
            temp = temp.next;
            length++;
        }

        // Step 2: Optimize k to avoid unnecessary rotations
        k = k % length;
        if (k == 0) {
            return head; // No rotation needed
        }

        // Step 3: Find the new tail (length - k) and new head
        temp.next = head; // Connect tail to head to form a cycle
        int stepsToNewTail = length - k;
        ListNode newTail = head;

        for (int i = 1; i < stepsToNewTail; i++) {
            newTail = newTail.next;
        }
        // Step 4: Break the cycle and update head
        ListNode newHead = newTail.next;
        newTail.next = null;

        return newHead;
    }
}
```

92. Reverse Linked List II

```
class Solution {
    public ListNode reverseBetween(ListNode head, int left, int right) {
        if (head == null || left == right) {
            return head; // No need to reverse if there's only one node or left == right
        }

        // Dummy node to handle edge cases (e.g., reversing from head)
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;

        // Step 1: Move prev to the node just before "left"
        for (int i = 1; i < left; i++) {
            prev = prev.next;
        }

        // Step 2: Reverse the sublist between "left" and "right"
        ListNode curr = prev.next;
        ListNode nextNode = null;
        ListNode prevNode = null;

        for (int i = left; i <= right; i++) {
            nextNode = curr.next;
            curr.next = prevNode;
            prevNode = curr;
            curr = nextNode;
        }

        // Step 3: Reconnect the reversed part back into the list
        prev.next.next = curr; // Connect tail of reversed sublist to remaining part
        prev.next = prevNode;  // Connect start of reversed sublist to previous part

        return dummy.next; // Return new head (dummy.next handles case where head is reversed)
    }
}
```

141. Linked List Cycle

```
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false; // No cycle if list is empty or has only one node
        }

        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;        // Move slow by 1 step
            fast = fast.next.next;   // Move fast by 2 steps

            if (slow == fast) {      // Cycle detected
                return true;
            }
        }

        return false; // No cycle
    }
}
```

148. Sort List

```
class Solution {
    public ListNode sortList(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        // Step 1: Split the list into two halves
        ListNode mid = getMiddle(head);
        ListNode rightHalf = mid.next;
        mid.next = null; // Split the list

        // Step 2: Recursively sort both halves
        ListNode left = sortList(head);
        ListNode right = sortList(rightHalf);

        // Step 3: Merge the sorted halves
        return merge(left, right);

    }

    // Function to find the middle node of the list
    private ListNode getMiddle(ListNode head) {
        ListNode slow = head, fast = head.next;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }
        return slow;
    }

    // Function to merge two sorted linked lists
    private ListNode merge(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode curr = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                curr.next = l1;
                l1 = l1.next;
            } else {
                curr.next = l2;
```

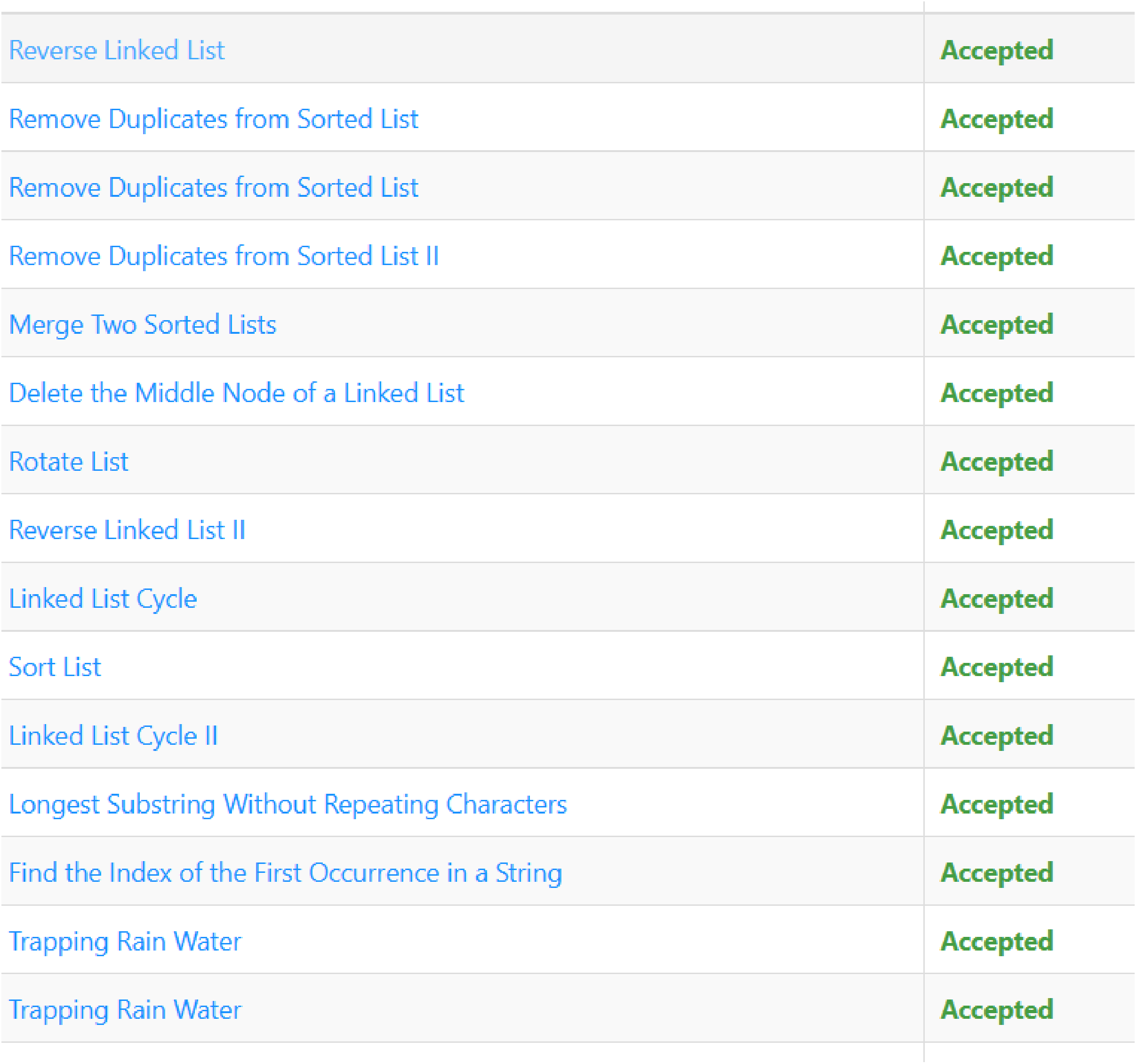
```
                l2 = l2.next;
            }
            curr = curr.next;
        }

        // Attach remaining nodes
        if (l1 != null) curr.next = l1;
        if (l2 != null) curr.next = l2;

        return dummy.next;
    }
}
```

142. Linked List Cycle II

```
public class Solution {
    public ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null) return null;

        ListNode slow = head, fast = head;

        // Step 1: Detect cycle using Floyd's algorithm
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                break;
            }
        }
        // No cycle found
        if (fast == null || fast.next == null) return null;

        // Step 2: Find cycle's starting node
        ListNode entry = head;
        while (entry != slow) {
            entry = entry.next;
            slow = slow.next;
        }

        return entry; // The start of the cycle
    }
}
```

Print Linked List

```
class Solution {
    public static void display(Node head) {
        if (head == null) return; // Base case

        System.out.print(head.data + " "); // Print current node
        display(head.next); // Recursive call
    }
}
```



Reverse Linked List	Accepted
Remove Duplicates from Sorted List	Accepted
Remove Duplicates from Sorted List	Accepted
Remove Duplicates from Sorted List II	Accepted
Merge Two Sorted Lists	Accepted
Delete the Middle Node of a Linked List	Accepted
Rotate List	Accepted
Reverse Linked List II	Accepted
Linked List Cycle	Accepted
Sort List	Accepted
Linked List Cycle II	Accepted
Longest Substring Without Repeating Characters	Accepted
Find the Index of the First Occurrence in a String	Accepted
Trapping Rain Water	Accepted
Trapping Rain Water	Accepted