Experiment - 2(A)

Student Name: Masud Alom **UID:** 22BCS16095
Branch: CSE **Section/Group:** NTPP_602-A
Semester: 6 **Date of Performance:** 17-02-25
Subject Name: Advanced Programming Lab-2 **Subject Code:** 22CSH-359

1. **Title:** Linked Lists (Remove duplicates from a sorted list) https://leetcode.com/problems/remove-duplicates-from-sorted-list

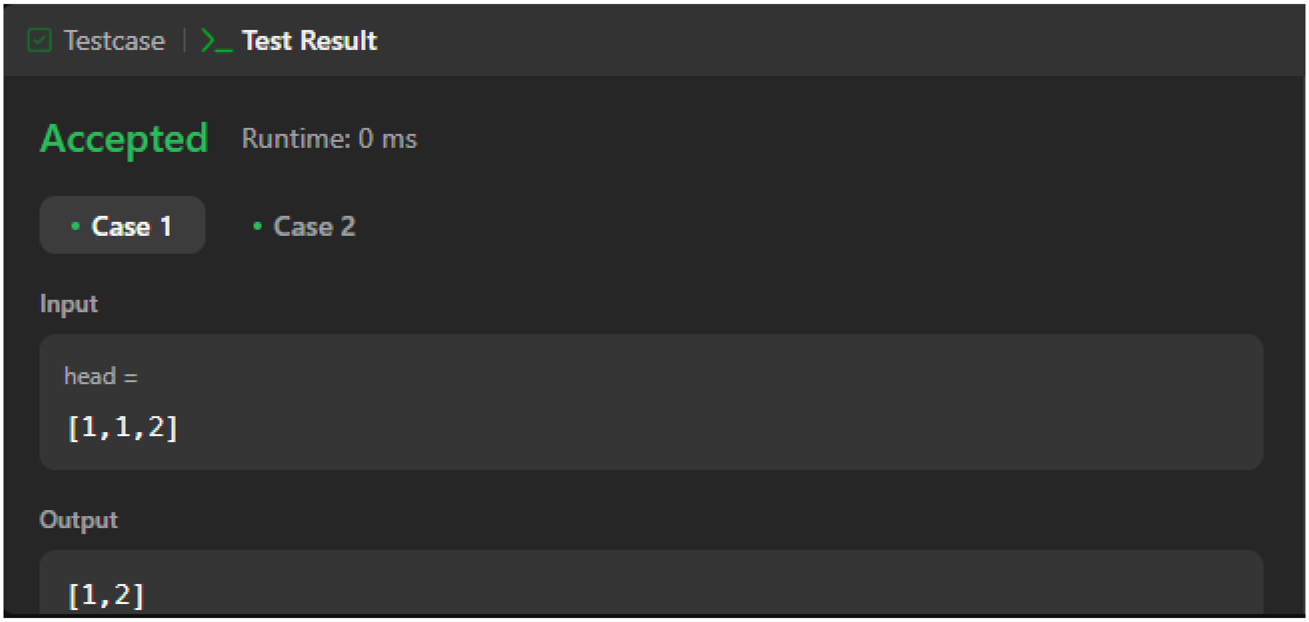
2. **Objective:** Given the head of a sorted linked list, the task is to remove all duplicates such that each element appears only once. Return the modified linked list, which is still sorted.

3. **Algorithm:**
   - **Iterate Through the Linked List:**
   - Start with the head of the list.
   - Traverse the list using a pointer current starting from the head.
   - **Check for Duplicates:**
   - For each node, compare the value of the current node with the value of the next node.
   - If the values are equal (i.e., a duplicate), update the current node's next pointer to skip the next node.
   - If the values are not equal, simply move the current pointer to the next node.
   - **End of List:**
   - Continue this process until you reach the end of the list (i.e., current.next is None).
   - **Return Modified List:**
   - The linked list will be modified in-place with the duplicates removed.
   - Return the modified head of the linked list.

4. Implementation/Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }

    static ListNode deserialize(String data) {
        if (data.length() <= 2) return null;
        String[] values = data.substring(1, data.length() - 1).split(",");
        ListNode dummy = new ListNode(0), current = dummy;
        for (String value : values) current = current.next = new
ListNode(Integer.parseInt(value.trim()));
        return dummy.next;
    }

    static void printList(ListNode head) {
        while (head != null) { System.out.print(head.val + " -> "); head =
head.next; }
        System.out.println("null");
    }
}

class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        ListNode current = head;
        while (current != null && current.next != null)
            current = (current.val == current.next.val) ? current.next =
current.next.next : current.next;
        return head;
    }

    public static void main(String[] args) {
        ListNode head = ListNode.deserialize("[1,1,2,3,3]");
        ListNode.printList(new Solution().deleteDuplicates(head));
    }
}
```

5. **Output:**



6. **Time Complexity:** O(n) 7. **Space Complexity:** O(1)

7. **Learning Outcomes:**

- **In-place Modifications in Linked Lists:**
- Understand how to perform operations on linked lists without using additional memory (i.e., modifying the list in-place).
- **Traversing Linked Lists:**
- Gain experience with traversing a linked list and manipulating the next pointers.
- **Handling Edge Cases:**
- The solution works even for edge cases where the linked list is empty (i.e., head is None) or has only one node.

Experiment 2(B)

1. **Title:** Reverse a linked list (https://leetcode.com/problems/reverse-linked-list/)

2. **Objective:** Given the head of a singly linked list, the task is to reverse the list and return the reversed list.

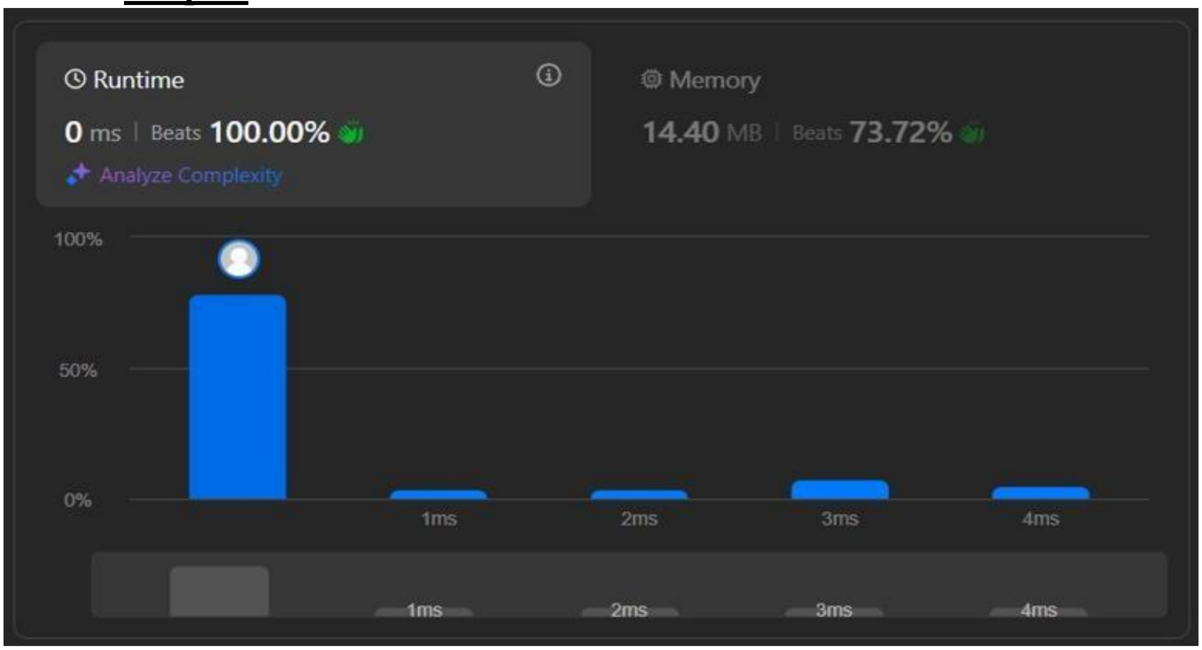
3. **Algorithm:**

- **Initialization:**
- Start with three pointers:
  - o prev initially set to None. o current initialized to head.
  - o next_node will help in keeping track of the next node in the list.
- **Traverse the List:**
- For each node in the linked list:
  - o Save the next node: next_node = current.next.
  - o Reverse the direction of the current node's next pointer: current.next = prev. o Move prev to the current node: prev = current.
  - o Move current to the next node: current = next_node.
- **End of List:**
- When current becomes None, the list is reversed, and prev will be the new head of the reversed list.
- **Return the Reversed List:**
- The new head of the reversed linked list is prev.

4. **Implementation/Code:**

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

5. Output:



6. Time Complexity: O(n)

7. Space Complexity: O(1)

8. Learning Outcomes:

- **Reversing a Linked List:**
- Learn how to reverse the direction of pointers in a singly linked list, both iteratively and recursively.
- **Recursive vs Iterative Approaches:**
- Understand the trade-offs between iterative and recursive approaches for solving linked list problems.
- **Linked List Manipulation:**
- Gain experience in manipulating linked list nodes and pointers to achieve desired outcomes (such as reversal).

Experiment 2(C)

1. **Title:** Delete middle node of a list (https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list)
2. **Objective:** Given the head of a linked list, the task is to delete the middle node and return the head of the modified list. The middle node is the $\lfloor n / 2 \rfloor$th node, where n is the length of the list, and $\lfloor x \rfloor$ represents the largest integer less than or equal to x.

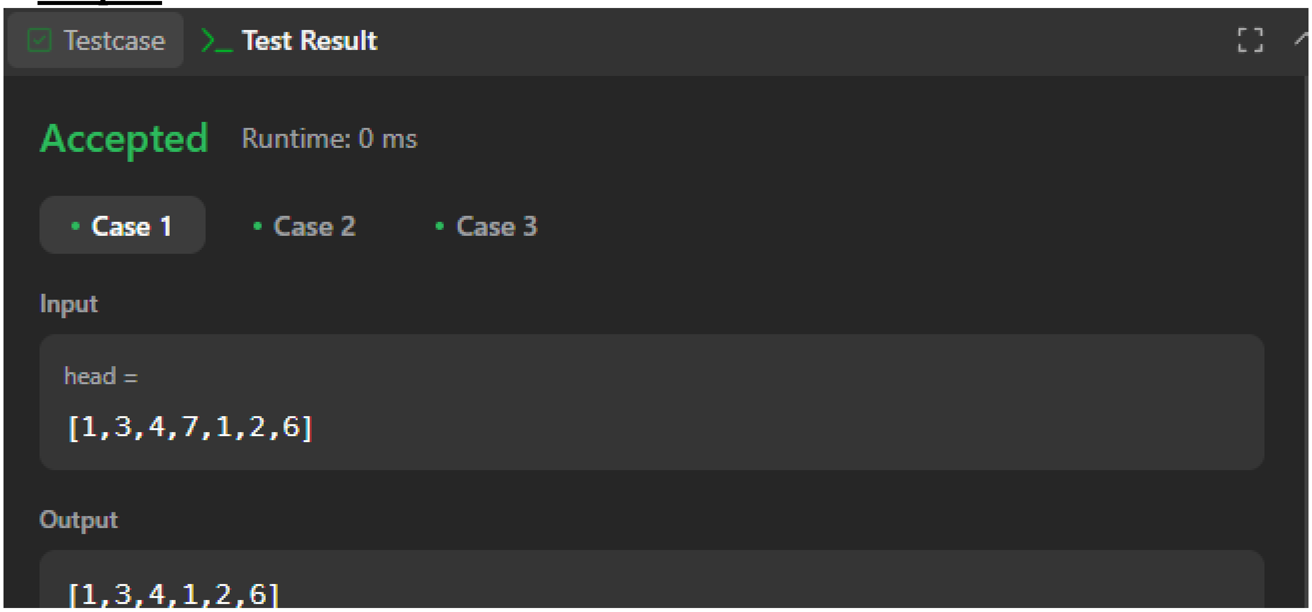
3. **Algorithm:**

- **Find the Length of the List:**
- Traverse the linked list and calculate the length n.
- **Determine the Middle Node:**
- The middle node is at the index n // 2.
- **Handle Edge Case for Small Lists:**
- If the list contains only one node (n == 1), return None (the list becomes empty).
- **Traverse to the Node Before the Middle Node:**
- Use a pointer to traverse to the node just before the middle node.
- **Delete the Middle Node:**
- Modify the next pointer of the node before the middle node to point to the node after the middle node.
- **Return the Modified List:** • Return the modified head of the linked list after the middle node is removed.

4. **Implementation/Code:**

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

        // Initialize slow and fast pointers
        ListNode slow = head, fast = head;
        ListNode prev = null;

        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;
    }
}
```

4. Output:



9. Time Complexity: O(n)   **7. Space Complexity:** O(1)

10. Learning Outcomes:

- **In-place Modifications in Linked Lists:**
- Learn how to modify linked lists in place, without creating new nodes or arrays.
- **Two Pointer Technique:**
- Understand the usage of the slow and fast pointers to find specific nodes in a linked list (in this case, the middle node).