# Experiment 3

**Student Name:** Avil saini **UID:** 22BET10380

**Branch:** IT **Section/Group:** 22BET\_IOT-702/B

**Semester:** 6th **Date of Performance:** 30/01/25

**Subject Name:** Advance Programming-II **Subject Code:** 22ITP-367

**Problem: 1.3.1: Merge Two Sorted Lists.**

**Problem Statement:** You are given the heads of two sorted linked lists list 1 and list 2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

1. **Objective:** Merge two sorted linked lists into a single sorted linked list by combining their nodes sequentially.
2. **Code:**

## class Solution {

## public:

## ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

## ListNode dummy; // Dummy node to simplify edge cases

## ListNode\* tail = &dummy; // Pointer to track the last node of merged list

## 

## while (list1 && list2) {

## if (list1->val < list2->val) {

## tail->next = list1;

## list1 = list1->next;

## } else {

## tail->next = list2;

## list2 = list2->next;

## }

## tail = tail->next;

## }

## 

## // Attach the remaining nodes (if any)

## tail->next = list1 ? list1 : list2;

## 

## return dummy.next;

## }

## };

## 3. Result:

## 

**Problem 1.3.2: Remove Duplicates from Sorted List II**

**Problem Statement:** Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

1. **Objective:** Find the minimum number of jumps required to reach the last index of the array.
2. **Code:**

## from typing import Optional

## # Definition for singly-linked list.

## class ListNode:

## def \_\_init\_\_(self, val=0, next=None):

## self.val = val

## self.next = next

## class Solution:

## def deleteDuplicates(self, head: Optional[ListNode]) -> Optional[ListNode]:

## current = head

## while current:

## # As long as the next node exists and has the same value,

## # skip the next node.

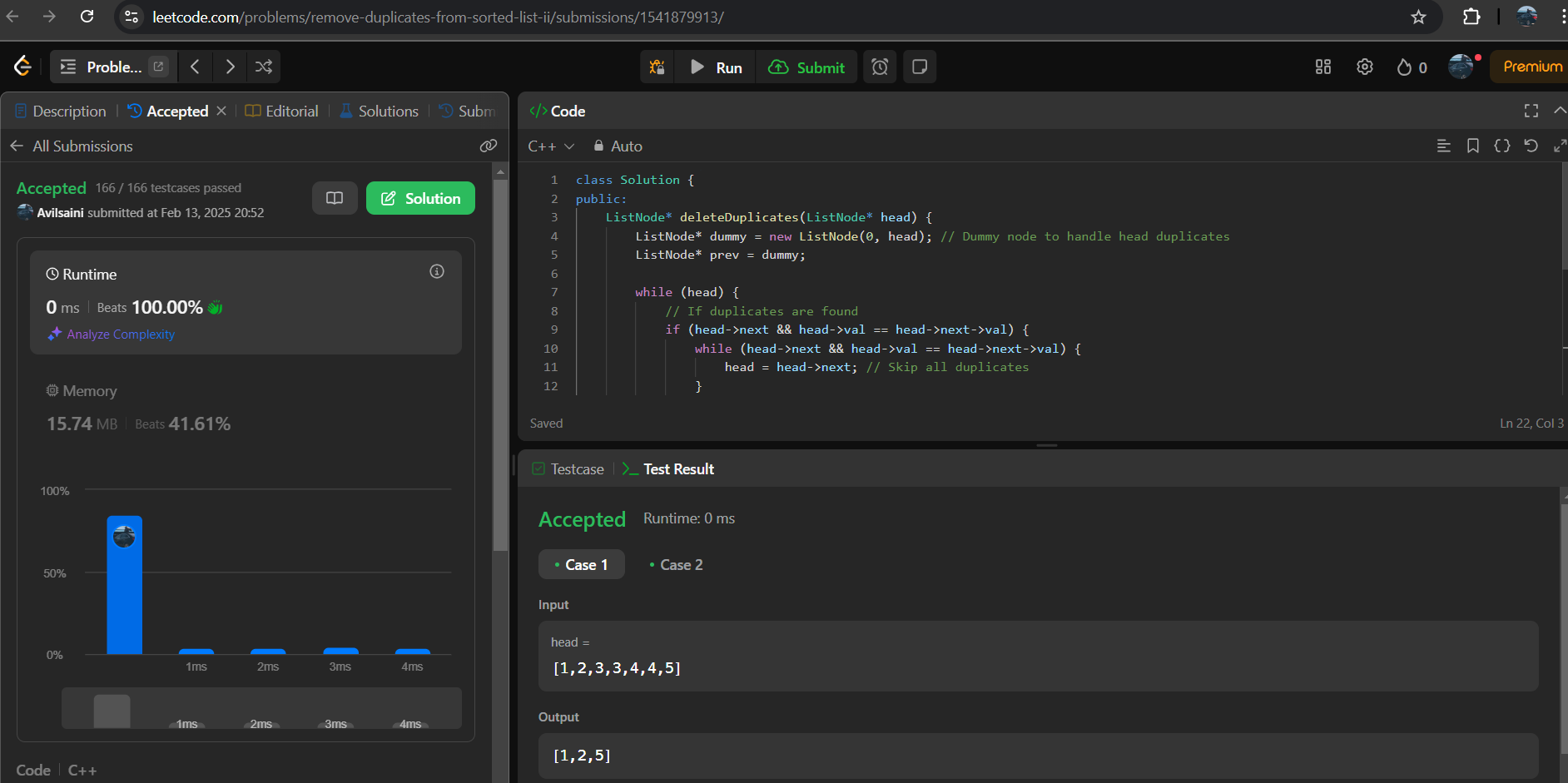
## while current.next and current.next.val == current.val:

## current.next = current.next.next

## current = current.next

## return head

1. **Result:**



**Problem 1.3.3: Reverse a linked list**

**Problem Statement:** Given the head of a singly linked list, reverse the list, and return the reversed list.

1. **Objective:** Given singly linked list so that the order of its nodes is reversed. In other words, you need to adjust the pointers of each node so that the head of the original list becomes the tail of the new list, and vice versa, effectively turning the list around.
2. **Code:**

from typing import Optional

# Definition for singly-linked list.

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

class Solution:

def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:

prev = None

curr = head

while curr:

nxt = curr.next # Store the next node

curr.next = prev # Reverse the pointer

prev = curr # Move prev to current

curr = nxt # Move to the next node

return prev

1. **Result:**

## 

**Problem 1.3.4: Medium Delete middle node of a list**

**Problem Statement:** Given the head of a linked list, remove its middle node (defined as the ⌊n/2⌋-th node in a zero-indexed list) and return the head of the updated list.

1. **Objective:** Efficiently delete the middle node in a single pass, ensuring the list remains properly linked while maintaining O(n) time and O(1) space complexity.
2. **Code:**

class Solution {

public:

ListNode\* deleteMiddle(ListNode\* head) {

if (!head || !head->next) return nullptr; // If list has 0 or 1 node, return nullptr

ListNode\* slow = head;

ListNode\* fast = head;

ListNode\* prev = nullptr; // To track the node before the middle

while (fast && fast->next) {

prev = slow;

slow = slow->next; // Move slow one step

fast = fast->next->next; // Move fast two steps

}

prev->next = slow->next; // Skip the middle node

delete slow; // Free memory

return head;

}

};

1. **Result:**

## 

**Problem 1.3.5: Rotate a list**

**Problem Statement:** Given the head of a linked list and an integer k, rotate the list to the right by k positions, and return the new head.

1. **Objective:** Rearrange the list in-place by adjusting pointers so that the rotation is completed in O(n) time and O(1) space.
2. **Code:**

class Solution {

public:

ListNode\* rotateRight(ListNode\* head, int k) {

if (!head || !head->next || k == 0) return head; // Edge cases

// Step 1: Compute the length of the linked list

int length = 1;

ListNode\* tail = head;

while (tail->next) {

tail = tail->next;

length++;

}

// Step 2: Optimize k

k = k % length;

if (k == 0) return head; // No rotation needed

// Step 3: Find new tail (length - k - 1)th node

ListNode\* newTail = head;

for (int i = 0; i < length - k - 1; i++)

newTail = newTail->next;

// Step 4: Update pointers to rotate the list

ListNode\* newHead = newTail->next;

newTail->next = nullptr;

tail->next = head; // Make it circular and then break

return newHead;

}

};

1. **Result:**

## 