# Experiment 3

**Student Name:** Tarun Baliyan **UID:** 22BET10184

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

**Semester:** 5th **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:**

## from typing import List, Optional

## # Definition for singly-linked list.

## class ListNode:

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

## self.val = val

## self.next = next

## class Solution:

## def mergeTwoLists(self, l1: Optional[ListNode], l2: Optional[ListNode]) -> Optional[ListNode]:

## dummy = ListNode(0)

## current = dummy

## while l1 and l2:

## if l1.val < l2.val:

## current.next = l1

## l1 = l1.next

## else:

## current.next = l2

## l2 = l2.next

## current = current.next

## current.next = l1 or l2

## return dummy.next

## def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:

## if not lists:

## return None

## while len(lists) > 1:

## merged\_lists = []

## for i in range(0, len(lists), 2):

## l1 = lists[i]

## l2 = lists[i + 1] if i + 1 < len(lists) else None

## merged\_lists.append(self.mergeTwoLists(l1, l2))

## lists = merged\_lists

## return lists[0]

## 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:**

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

# If the list is empty or has only one node, return None.

if not head or not head.next:

return None

# Initialize two pointers and a previous pointer.

slow = head

fast = head

prev = None

# Move fast pointer two steps and slow pointer one step each iteration.

while fast and fast.next:

prev = slow

slow = slow.next

fast = fast.next.next

# 'slow' is now at the middle node.

# Remove the middle node by linking prev to slow.next.

prev.next = slow.next

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:**

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 rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:

# If the list is empty or has only one node, no rotation is needed.

if not head or not head.next or k == 0:

return head

# First, compute the length of the list and get the tail.

length = 1

tail = head

while tail.next:

tail = tail.next

length += 1

# Compute the effective rotations needed.

k %= length

if k == 0:

return head

# Form a circular linked list.

tail.next = head

# Find the new tail: the (length - k)-th node.

steps\_to\_new\_tail = length - k

new\_tail = head

for \_ in range(steps\_to\_new\_tail - 1):

new\_tail = new\_tail.next

# The new head is the node after the new tail.

new\_head = new\_tail.next

# Break the circle.

new\_tail.next = None

return new\_head

1. **Result:**

## 