Experiment - 2(A)

Student Name: DEVENSH YADAV **UID:** 22BCS16082
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 Solution:
def deleteDuplicates(self, head):
    current = head
    while current and current.next:
        # If current node's value is equal to the next node's value
```

```
            if current.val == current.next.val:
                current.next = current.next.next  # Skip the duplicate node
            else:
                current = current.next  # Move to the next node
        return 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.

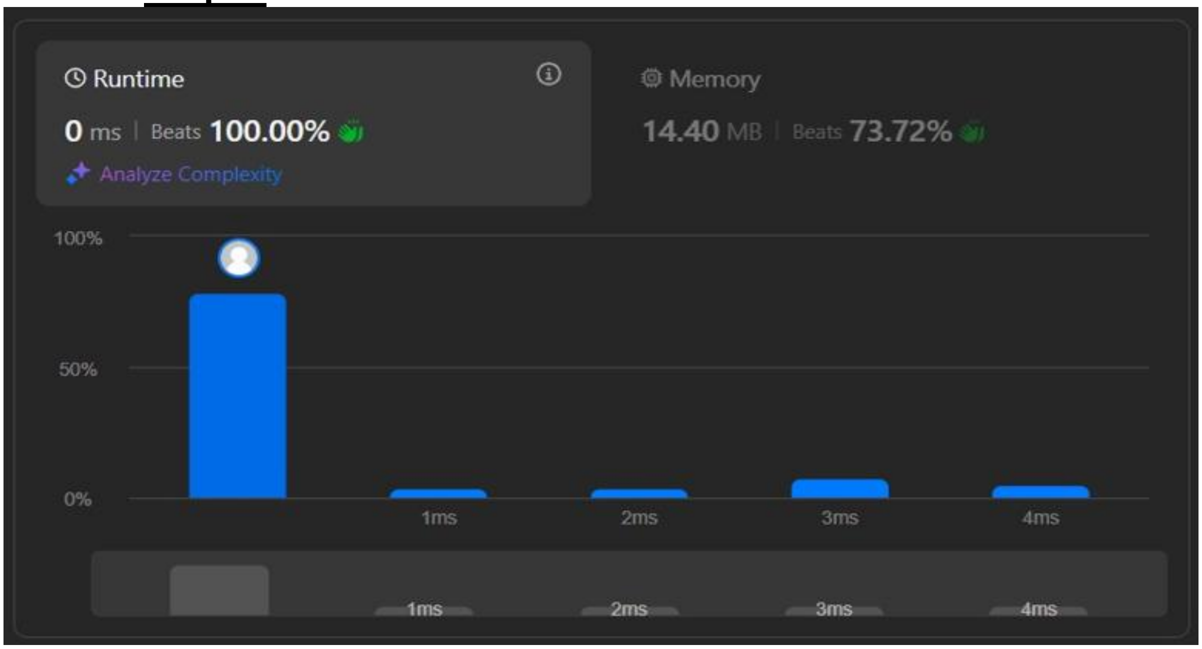
3. **Algorithm:**

- **Initialization:**
  - Start with three pointers:
    - prev initially set to None.
    - current initialized to head.
    - next_node will help in keeping track of the next node in the list.
- **Traverse the List:**
  - For each node in the linked list:
    - Save the next node: next_node = current.next.
    - Reverse the direction of the current node's next pointer: current.next = prev.
    - Move prev to the current node: prev = current.
    - 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 ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
class Solution:
    def reverseList(self, head):
        prev = None
        current = head
        while current:
            next_node = current.next  # Save next node
            current.next = prev  # Reverse the current node's pointer
            prev = current  # Move prev to current node
            current = next_node  # Move current to next node
        return prev  # Return the new head
```

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 ⌊n / 2⌋th node, where n is the length of the list, and ⌊x⌋ represents the largest integer less than or equal to x.

3. **Algorithm:**

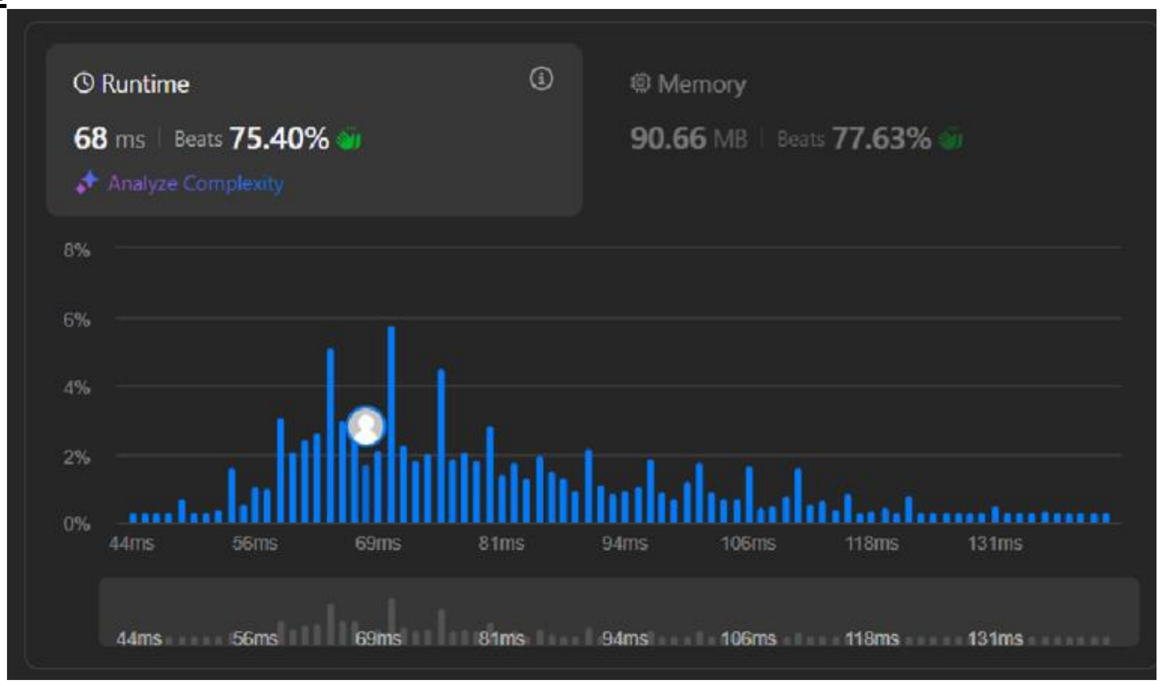
- **Find the Length of the List:**
  - Traverse the linked list and calculate the length n.
- **Determine the Middle Node:**
  - The middle node is at the index n // 2.
- **Handle Edge Case for Small Lists:**
  - If the list contains only one node (n == 1), return None (the list becomes empty).
- **Traverse 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 ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteMiddle(self, head):
        # If the list contains only one node, return None (empty list 
after removal)
        if not head or not head.next:
            return None

        # Initialize two pointers: slow (to find the middle) and fast (to 
find the end)
        slow = head
        fast = head
        prev = None
```

```
        # Traverse the list with fast moving two steps at a time and slow
moving one step
        while fast and fast.next:
            fast = fast.next.next
            prev = slow
            slow = slow.next

        # Remove the middle node by connecting prev to slow.next
        if prev:
            prev.next = slow.next

        return head
```

5. 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).