Experiment 4 A

Student Name: Rhythm Tyagi **UID:** 22BCS17203
Branch: CSE **Section/Group:**IOT_NTPP_602-A
Semester: 6th **Date of Performance:**20-01-25
Subject Name: AP2 **Subject Code:** 22CSP-351

Aim: 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*.

Objective: The objective is to remove all duplicate nodes from a sorted linked list, retaining only distinct elements, and return the modified list while maintaining the original sorted order.element.

Algorithm:

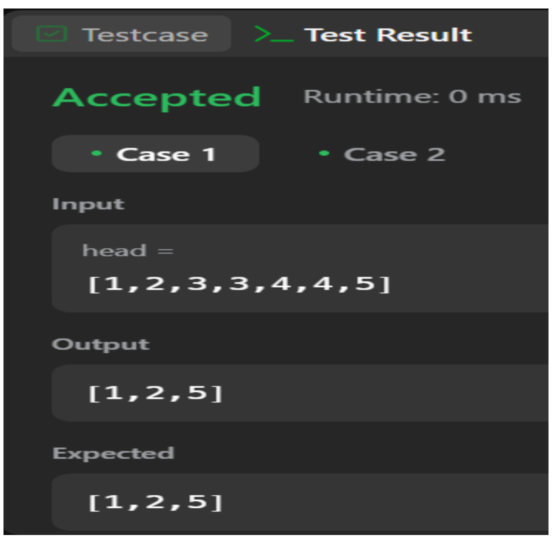
1. Handle Edge Case: If the list is empty or has one node, return it.
2. Create Dummy Node: Helps manage edge cases where the first node(s) are duplicates.
3. Use Two Pointers:
   - prev tracks the last unique node.
   - head traverses the list.
4. Skip Duplicates:
   - If head.val == head.next.val, move head forward until a new value is found.
   - Link prev.next to head.next to remove duplicates.
5. Update Pointers: Move prev when encountering a unique node.
6. Return the Updated List

Code

```
class Solution {
   public ListNode deleteDuplicates(ListNode head) {
      if (head == null || head.next == null) return head; // If list is empty or has only one node
      ListNode dummy = new ListNode(0, head); // Dummy node to handle edge cases
      ListNode prev = dummy; // Pointer to track the last unique node
       while (head != null) {
         if (head.next != null && head.val == head.next.val) {
```

```
            while (head.next != null && head.val == head.next.val) {
                head = head.next;
            }
            prev.next = head.next; // Remove duplicates
        } else {
            prev = prev.next; // Move prev pointer when it's a unique node
        }
        head = head.next; // Move to the next node
    }
    return dummy.next; // Return new head after removal
} }
```

Output:



Learning Outcomes:

a. Understanding how to remove duplicates from a sorted linked list while preserving unique elements.
b. Applying dummy nodes and two-pointer techniques for efficient in-place list modifications.
c. Enhancing problem-solving skills with edge case handling and pointer manipulation

Experiment 4 B

Aim: Given the head of a linked list, return *the list after sorting it in **ascending order***.

Objective: The objective is to sort a given linked list in ascending order using the Merge Sort algorithm, ensuring an efficient O(nlogn) time complexity while maintaining stability and in-place modification.

Algorithm:

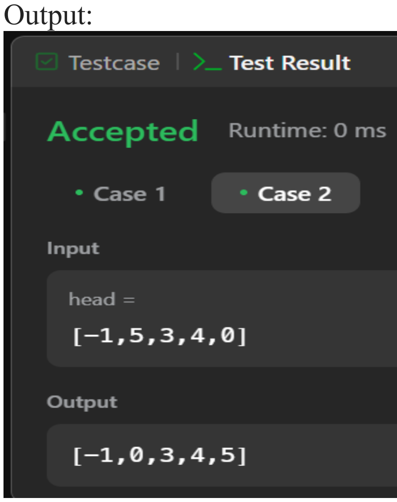
1. **Base Case**: If the list is empty or has one node, return it.
2. **Find Middle**: Use the slow and fast pointer technique to find the middle node.
3. **Split the List**: Divide the list into two halves at the middle node.
4. **Recursively Sort**: Apply the sortList function to both halves.
5. **Merge Halves**: Use the merge function to combine two sorted lists.
6. **Return the Sorted List**.

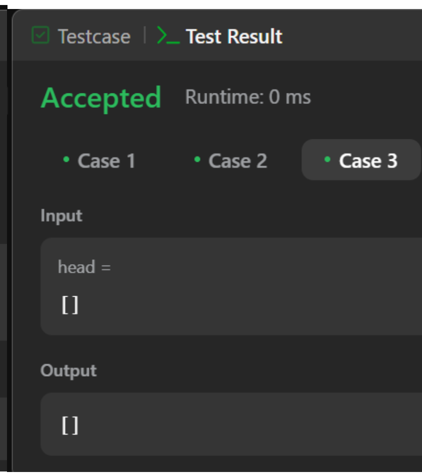
CODE:

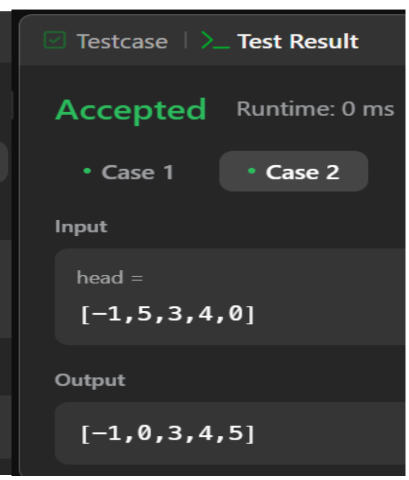
```
class Solution {
   public ListNode sortList(ListNode head) {
      if (head == null || head.next == null) return head; // Base case for recursion
      // Step 1: Split the list into two halves
      ListNode mid = getMiddle(head);
      ListNode rightHalf = mid.next;
      mid.next = null; // Split the list
      // Step 2: Recursively sort both halves
      ListNode left = sortList(head);
      ListNode right = sortList(rightHalf);
      // Step 3: Merge sorted halves
      return merge(left, right);
   }
   // Function to find the middle of the linked list (using slow & fast pointers)
   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; // Middle node
   }
   // Function to merge two sorted linked lists
   private ListNode merge(ListNode l1, ListNode l2) {
```

```
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;
        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                tail.next = l1;
                l1 = l1.next;
            } else {
                tail.next = l2;
                l2 = l2.next;
            }
            tail = tail.next;
        }
        // Append remaining nodes
        tail.next = (l1 != null) ? l1 : l2;
        return dummy.next;
    }
}
```

Output:







Learning Outcomes:

1. Understanding Merge Sort and its application to linked lists.
2. Using slow and fast pointers to find the middle of a linked list.
3. Learning how to split and merge linked lists efficiently.
4. Implementing recursive sorting with optimal time complexity O(nlogn).

Experiment 4 C

Aim: Given an array nums with n objects colored red, white, or blue, sort them **in-place** so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

Objective: To sort an array containing 0s, 1s, and 2s in-place using the Dutch National Flag algorithm, ensuring all 0s appear first, followed by 1s, and then 2s in O(n) time.

Algorithm:

1. Initialize three pointers:
   - low = 0 (boundary for 0s)
   - mid = 0 (current element)
   - high = n - 1 (boundary for 2s)
2. Iterate while mid <= high:
   - If nums[mid] == 0:
     - Swap nums[mid] with nums[low]
     - Increment low and mid
   - Else if nums[mid] == 1:
     - Move mid forward
   - Else (nums[mid] == 2):
     - Swap nums[mid] with nums[high]
     - Decrement high (but keep mid unchanged to recheck)
3. Continue until mid > high → The array is sorted.

CODE:

```
class Solution {
    public void sortColors(int[] nums) {
        int low = 0, mid = 0, high = nums.length - 1;

        while (mid <= high) {
            if (nums[mid] == 0) {
                swap(nums, low, mid);
                low++;
                mid++;
```

```
            } else if (nums[mid] == 1) {
                mid++;
            } else { // nums[mid] == 2
                swap(nums, mid, high);
                high--;
            }
        }
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}
```

Output: