

## Experiment-3

**Student Name:** Sejjwal Yadav

**Branch:** BE-IT

**Semester:** 6th

**Subject Name:** AP LAB-II

**UID:** 22BET10241

**Section/Group:** IOT/702/A

**Date of Performance:** 30/01/25

**Subject Code:** 22ITT-351

### PROBLEM 1

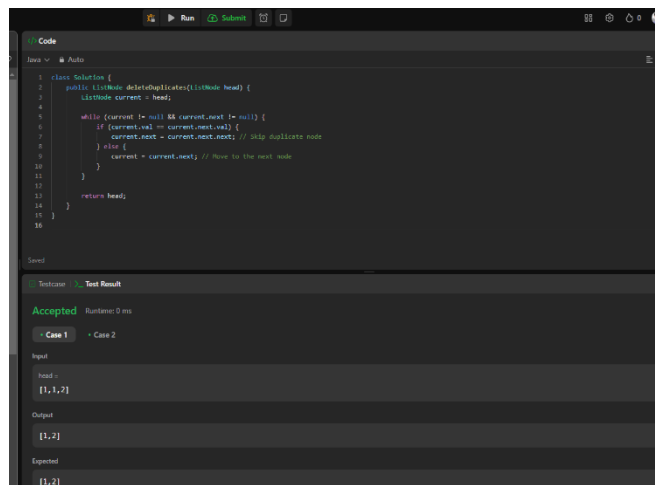
#### 1. Aim:

To develop an efficient algorithm that removes all nodes with duplicate values from a sorted linked list, ensuring that only distinct elements remain while maintaining the sorted order.

#### 2. Code:

```
class Solution {  
    public ListNode deleteDuplicates(ListNode head) {  
        ListNode current = head;  
  
        while (current != null && current.next != null) {  
            if (current.val == current.next.val) {  
                current.next = current.next.next; // Skip duplicate node  
            } else {  
                current = current.next; // Move to the next node  
            }  
        }  
  
        return head;  
    }  
}
```

#### 3. Output:



```
class Solution {  
    public ListNode deleteDuplicates(ListNode head) {  
        ListNode current = head;  
  
        while (current != null && current.next != null) {  
            if (current.val == current.next.val) {  
                current.next = current.next.next; // Skip duplicate node  
            } else {  
                current = current.next; // Move to the next node  
            }  
        }  
  
        return head;  
    }  
}
```

Testcase: Test Result  
Accepted Runtime: 0 ms  
Case 1 Case 2  
Input:  
head:  
[1,1,2]  
Output:  
[1,2]  
Expected:  
[1,2]

## PROBLEM 2

### 1. AIM:

To develop an algorithm that reverses a singly linked list, changing the direction of pointers so that the last node becomes the head and the original head becomes the tail.

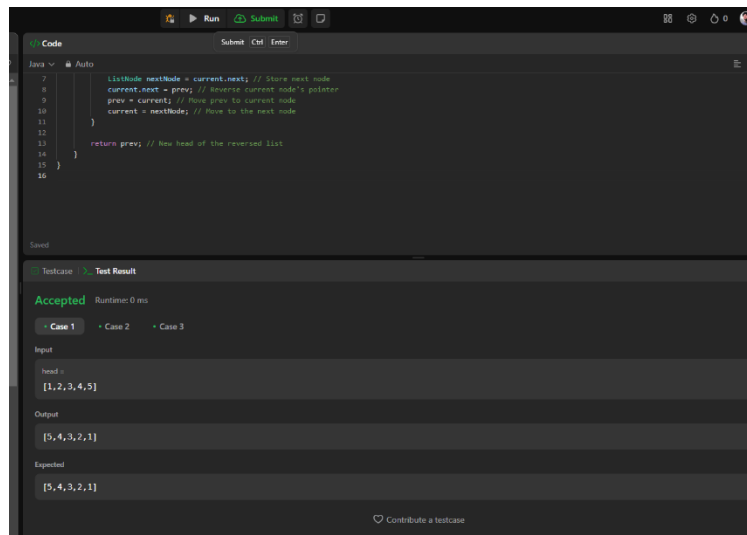
### 1. CODE:

```
class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode current = head;

        while (current != null) {
            ListNode nextNode = current.next; // Store next node
            current.next = prev; // Reverse current node's pointer
            prev = current; // Move prev to current node
            current = nextNode; // Move to the next node
        }

        return prev; // New head of the reversed list
    }
}
```

### 2. Output:



The screenshot shows a code editor with the following Java code:

```
7      ListNode nextNode = current.next; // Store next node
8      current.next = prev; // Reverse current node's pointer
9      prev = current; // Move prev to current node
10     current = nextNode; // Move to the next node
11 }
12
13 return prev; // New head of the reversed list
14 }
15 }
16 }
```

Below the code editor, the test results are displayed:

Testcase: **Accepted** Runtime: 0 ms

Case 1 Case 2 Case 3

Input:

head: [1,2,3,4,5]

Output: [5,4,3,2,1]

Expected: [5,4,3,2,1]

Contribute a test case

## PROBLEM 3

### 1. AIM:

To develop an algorithm that efficiently deletes the middle node of a singly linked list, ensuring that the list remains properly linked after deletion.

### 2. CODE:

```
class Solution {
public ListNode deleteMiddle(ListNode head) {
    if (head == null || head.next == null) {
        return null; // If the list is empty or has only one node, return null
    }

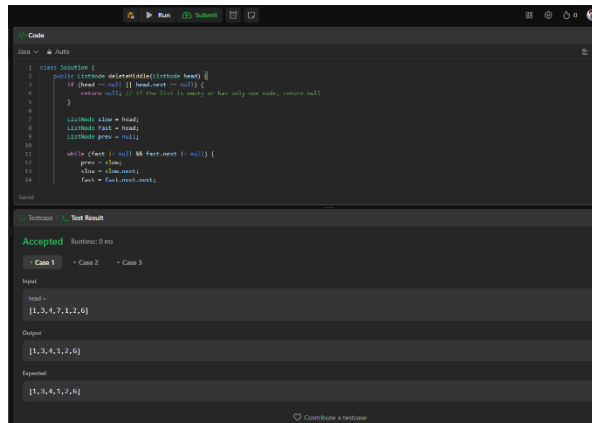
    ListNode slow = head;
    ListNode fast = head;
    ListNode prev = null;

    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;
}
```

### 3. Output:



The screenshot shows a code editor with the following code:

```
1 class Solution {
2     public ListNode deleteMiddle(ListNode head) {
3         if (head == null || head.next == null) {
4             return null; // If the list is empty or has only one node, return null
5         }
6
7         ListNode slow = head;
8         ListNode fast = head;
9         ListNode prev = null;
10
11         while (fast != null && fast.next != null) {
12             prev = slow;
13             slow = slow.next;
14             fast = fast.next.next;
15         }
16
17         // Delete the middle node
18         prev.next = slow.next;
19
20         return head;
21     }
22 }
```

Below the code, the test results are shown:

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

[1,3,4,7,1,2,6]

Output

[1,3,4,1,2,6]

Expected

[1,3,4,1,2,6]

## PROBLEM 4

### 1. AIM:

To design an algorithm that removes all nodes from a sorted singly linked list that contain duplicate values, ensuring that only distinct elements remain in the modified list.

### 2. CODE:

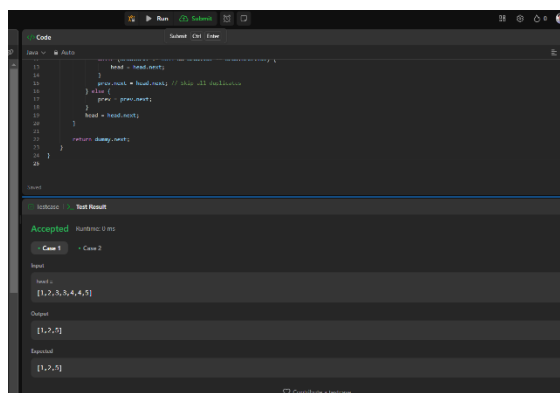
```
class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if (head == null || head.next == null) {
            return head;
        }

        ListNode dummy = new ListNode(0, head);
        ListNode prev = dummy;

        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; // Skip all duplicates
            } else {
                prev = prev.next;
            }
            head = head.next;
        }

        return dummy.next;
    }
}
```

### 3. Output:



```
11 public ListNode deleteDuplicates(ListNode head) {
12     if (head == null || head.next == null) {
13         return head;
14     }
15     ListNode dummy = new ListNode(0, head);
16     ListNode prev = dummy;
17     while (head != null) {
18         if (head.next != null && head.val == head.next.val) {
19             while (head.next != null && head.val == head.next.val) {
20                 head = head.next;
21             }
22             prev.next = head.next; // Skip all duplicates
23         } else {
24             prev = prev.next;
25         }
26         head = head.next;
27     }
28     return dummy.next;
29 }
```

Test Result

Accepted Runtime: 0 ms

Case 1

Input: [1,2,3,3,4,4,5]

Output: [1,2,5]

## PROBLEM 5

### 1. AIM:

The goal is to merge k sorted linked lists into a single sorted linked list and return the merged list.

### 2. CODE:

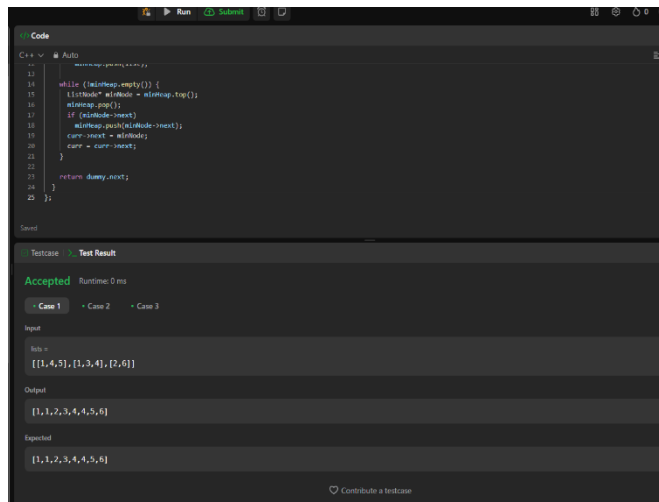
```
class Solution {
public ListNode mergeKLists(ListNode[] lists) {
    ListNode dummy = new ListNode(0);
    ListNode curr = dummy;
    Queue<ListNode> minHeap = new PriorityQueue<>((a, b) -> Integer.compare(a.val, b.val));

    for (final ListNode list : lists)
        if (list != null)
            minHeap.offer(list);

    while (!minHeap.isEmpty()) {
        ListNode minNode = minHeap.poll();
        if (minNode.next != null)
            minHeap.offer(minNode.next);
        curr.next = minNode;
        curr = curr.next;
    }

    return dummy.next;
}
```

### 3. Output:



```
Code
C++
12
13
14 while (!minHeap.empty()) {
15     ListNode* minNode = minHeap.top();
16     minHeap.pop();
17     if (minNode->next)
18         minHeap.push(minNode->next);
19     curr->next = minNode;
20     curr = curr->next;
21 }
22 return dummy->next;
23 }
```

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

Node =

[1,4,5],[1,3,4],[2,6]

Output

[1,1,2,3,4,4,5,6]

Expected

[1,1,2,3,4,4,5,6]

Contribute a testcase

## PROBLEM 6

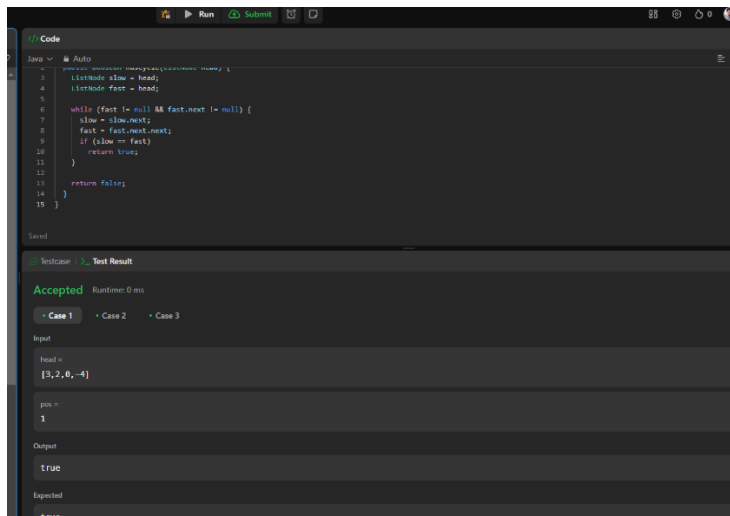
### 1. AIM:

The goal of this problem is to detect whether a given singly linked list contains a cycle. A cycle occurs if a node in the list points back to a previous node, forming a loop instead of terminating at null.

### 2. CODE:

```
class Solution {  
    public boolean hasCycle(ListNode head) {  
        ListNode slow = head;  
        ListNode fast = head;  
  
        while (fast != null && fast.next != null) {  
            slow = slow.next;  
            fast = fast.next.next;  
            if (slow == fast)  
                return true;  
        }  
  
        return false;  
    }  
}
```

### 3. Output:



```
Code  
Java -> Auto  
1 public boolean hasCycle(ListNode head) {  
2     ListNode slow = head;  
3     ListNode fast = head;  
4  
5  
6     while (fast != null && fast.next != null) {  
7         slow = slow.next;  
8         fast = fast.next.next;  
9         if (slow == fast)  
10            return true;  
11    }  
12  
13    return false;  
14 }  
15 }  
  
Test Result  
Accepted Runtime: 0 ms  
Case 1 Case 2 Case 3  
Input  
head =  
[3, 2, 0, -4]  
pos =  
1  
Output  
true  
Expected  
true
```