DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

CHANDIGARH UNIVERSITY

Discover. Learn. Empower.

Experiment-3

Student Name: Sejjwal Yadav **UID:** 22BET10241
Branch: BE-IT **Section/Group:IOT/702/A**
Semester:6th **Date of Performance:30/01/25**
Subject Name: AP LAB-II **Subject Code:** 22ITT-351

PROBLEM 1

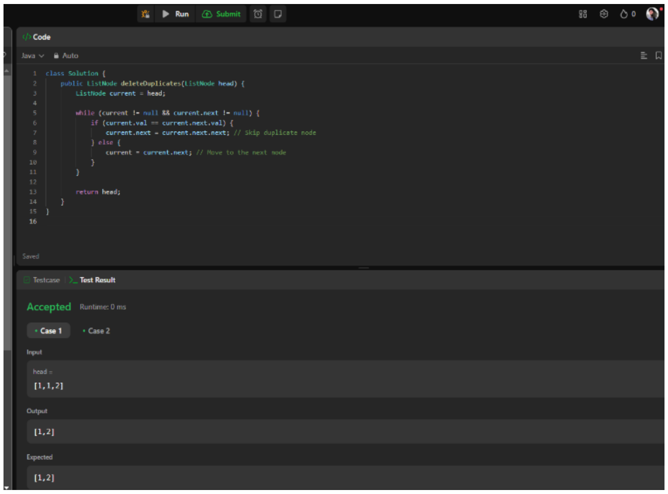
1. Aim:

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;
    }
}
```

Accepted Runtime: 0 ms

• Case 1 • Case 2

Input

head =

```
[1,1,2]
```

Output

```
[1,2]
```

Expected

```
[1,2]
```

PROBLEM 2

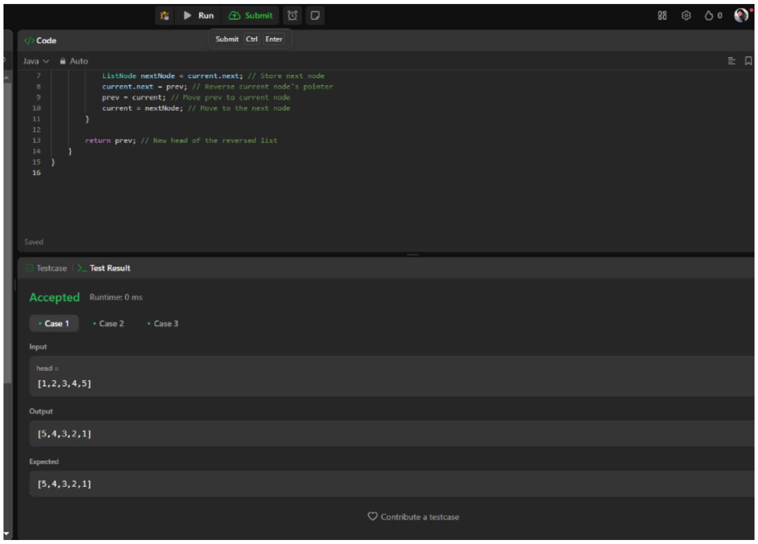
1. AIM:

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:



```
            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
    }
}
```

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]
```

PROBLEM 3

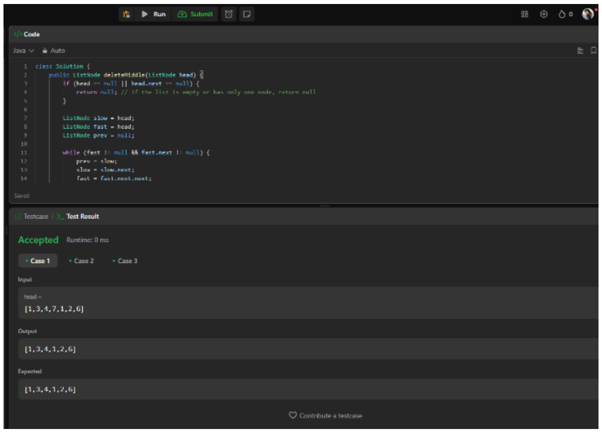
1. AIM:

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:



```
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;
```

Accepted Runtime: 0 ms

Input

```
head =
[1,3,4,7,1,2,6]
```

Output

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

Expected

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

PROBLEM 4

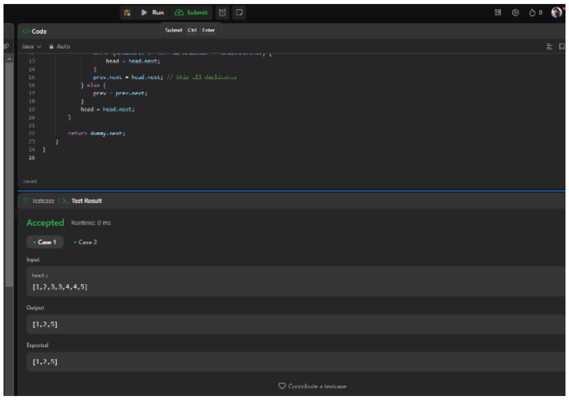
1. AIM:

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:



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

    return dummy.next;
  }
}
```

Accepted Runtime: 0 ms

Input

head =

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

Output

```
[1,2,5]
```

Expected

```
[1,2,5]
```

PROBLEM 5

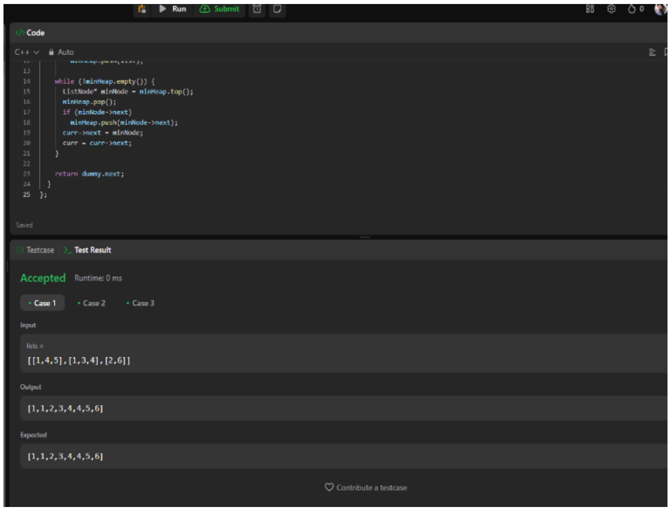
1. AIM:

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:



```
      minHeap.push(list);

    while (!minHeap.empty()) {
      ListNode* minNode = minHeap.top();
      minHeap.pop();
      if (minNode->next)
        minHeap.push(minNode->next);
      curr->next = minNode;
      curr = curr->next;
    }

    return dummy.next;
  }
};
```

Accepted Runtime: 0 ms

Input

```
lists =
[[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]
```

PROBLEM 6

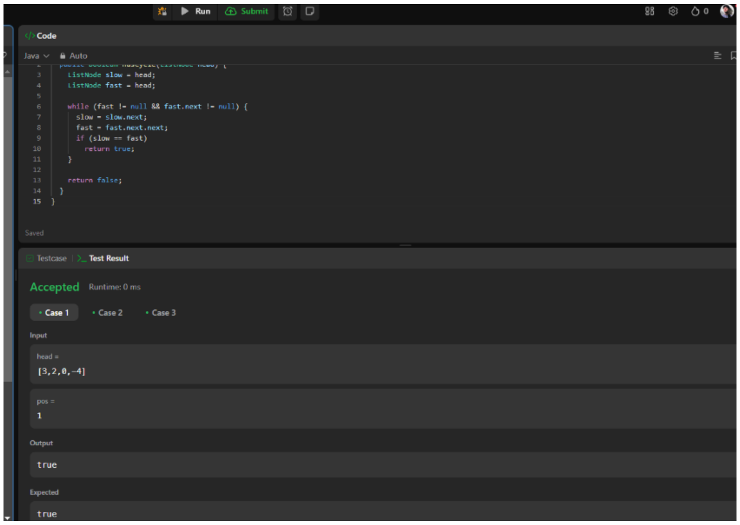
1. AIM:

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**:



```
  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;
  }
}
```

Accepted Runtime: 0 ms

Input

head =
[3,2,0,-4]

pos =
1

Output

true

Expected

true