Experiment-3

Name: Riya Thakur
UID: 22BET10171
Branch: BE-IT
Section/Group: 22BET_IOT-702/A
Semester: 6th
Date of Performance: 31/01/25
Subject Name: AP LAB-II
Subject Code: 22ITP-351

Problem-1

1.Aim:

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true if there is a cycle in the linked list. Otherwise, return false.

2.Objective:

- There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.
- Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.
- Return true if there is a cycle in the linked list. Otherwise, return false

3.Code:

```
class Solution {

 public:

  bool hasCycle(ListNode* head) {

    ListNode* slow = head;

    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {

      slow = slow->next;

      fast = fast->next->next;

      if (slow == fast)

        return true;
```

```
    }
    return false;
  }
    };
```

4.Output:

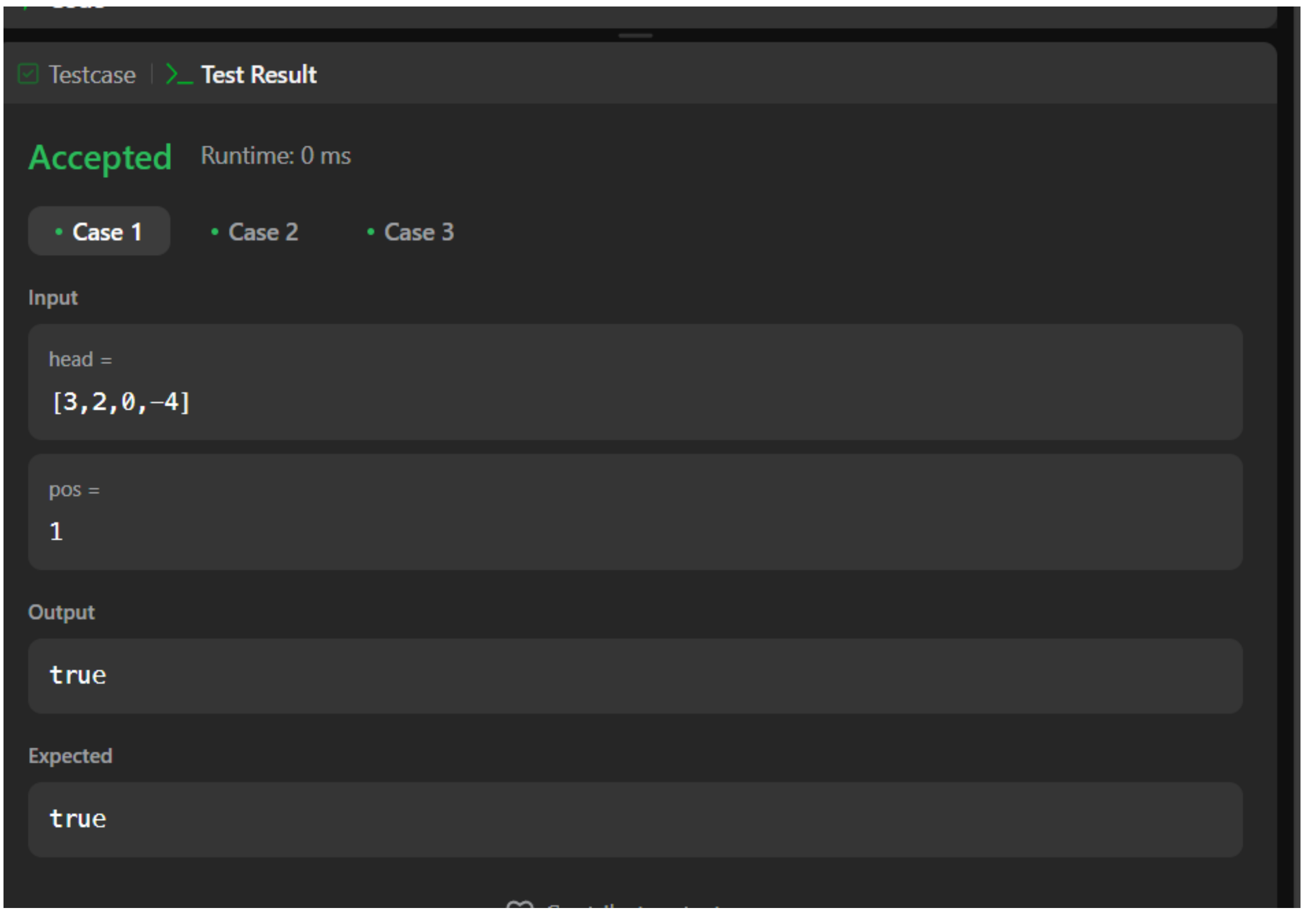

Fig.1.Detect a cycle in a linkedlist

Problem-2

1.Aim:

Reverse a linked List- the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list.

2.Objective:

- Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right,
- return the reversed list.

3.Code:

```
class Solution {

    public:

 ListNode* reverseBetween(ListNode* head, int left, int right) {

    if (!head || left == right) return head;



  ListNode* dummy = new ListNode(0);

  dummy->next = head;

  ListNode* prev = dummy;



  for (int i = 1; i < left; ++i) {

      prev = prev->next;

  }



  ListNode* start = prev->next;

  ListNode* then = start->next;
```

```
        for (int i = 0; i < right - left; ++i) {
            start->next = then->next;
            then->next = prev->next;
            prev->next = then;
            then = start->next;
        }

        return dummy->next;
    }
};
```

4.Output:

Fig.2:Reverse LinkedList

Problem-3

1.Aim: Rotate List-Given the head of a linked list, rotate the list to the right by k places.

2.Objective:

- Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right,
- return the reversed list.

3.Code:

```
class Solution {

public:

ListNode* rotateRight(ListNode* head, int k) {

    if(head==NULL ||head->next==NULL)return head;

    int n=0;//n is length

    ListNode* temp=head;

    ListNode* tail=NULL;

    while(temp!=NULL){

     if(temp->next==NULL)tail=temp;

     temp=temp->next;

     n++;

     }

     k=k%n;

     if(k==0)return head;

     //I have to place temp at (n-k) position

     temp=head;

     for(int i=1;i<n-k;i++)

     {
```

```
    temp=temp->next;
 }
 tail->next=head;
 head=temp->next;
 temp->next=NULL;
 return head;
   }
};
```

4.Output:

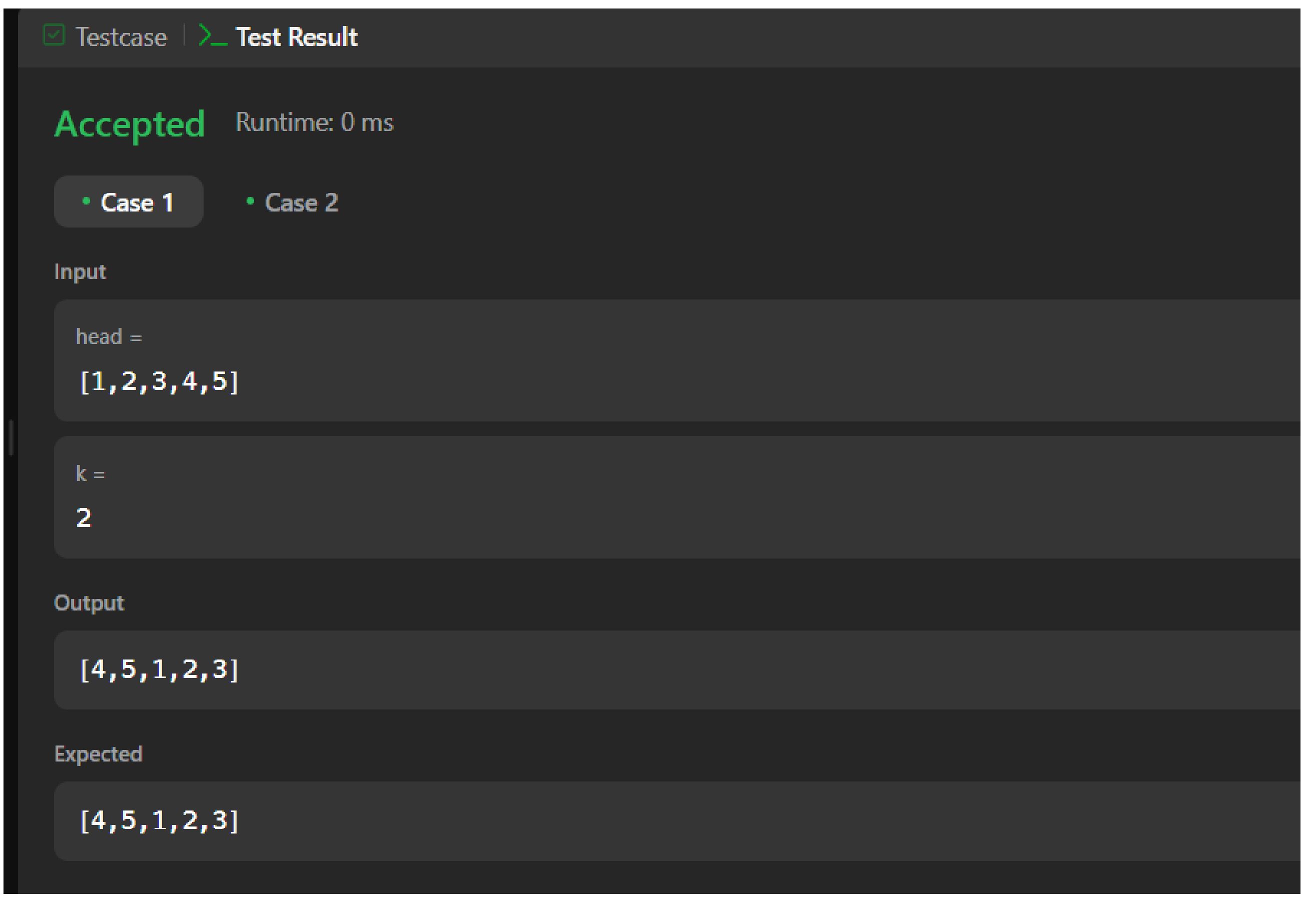

Fig.3:Rotate List

Problem-4

1.Aim: The aim of this code is to merge k sorted linked lists into a single sorted linked list efficiently.

2.Objective:

- Efficient Merging: Use a min-heap (priority queue) to merge k sorted linked lists while maintaining the overall sorted order.
- Optimize Time Complexity: Achieve an optimal time complexity of O(N log k), where N is the total number of elements and k is the number of lists.

3.Code:

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, Compare> minHeap;

        // Push the head of each linked list into the min-heap
        for (ListNode* list : lists) {
            if (list) minHeap.push(list);
        }

        ListNode dummy(0); // Dummy node to simplify list construction
        ListNode* tail = &dummy;

        // Extract the smallest element and insert the next element from that list
        while (!minHeap.empty()) {
```

```
            ListNode* node = minHeap.top();

            minHeap.pop();

            tail->next = node;

            tail = tail->next;



            if (node->next) {

                minHeap.push(node->next);

            }

        }



        return dummy.next;

    }

};
```

4.Output:

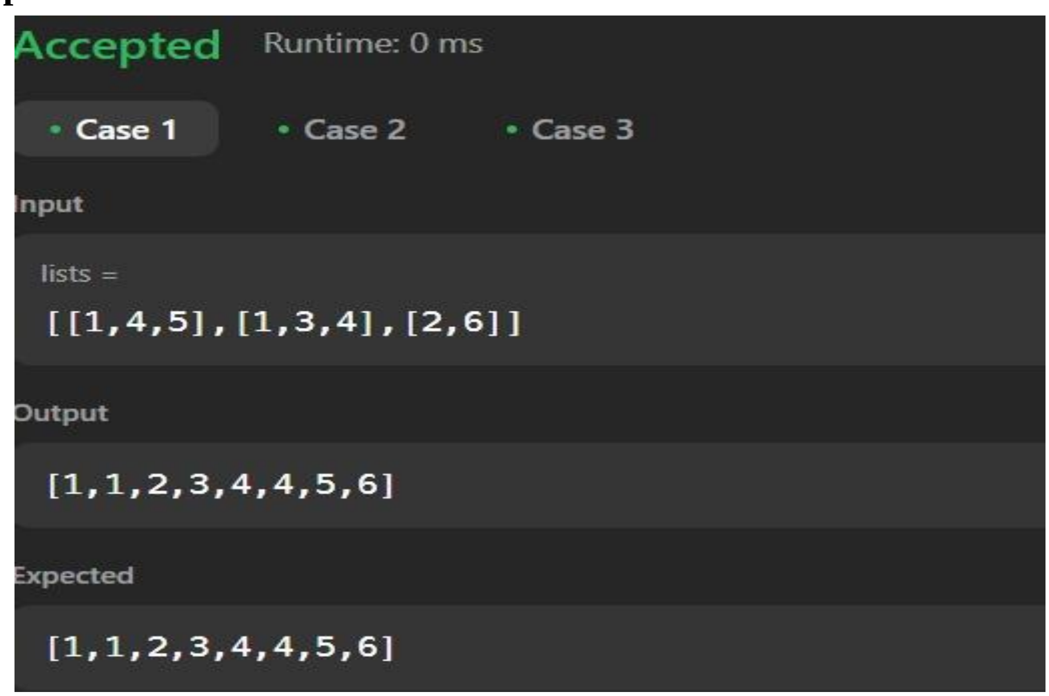

Fig.4:Single Sorted LinkedList

Problem-5

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

2.Objective:

- Implement an Efficient Sorting Algorithm.
- Implement an efficient algorithm like Merge Sort to achieve O(N log N) time complexity.

3.Code:

```
class Solution {
public:
// Function to find the middle node of the linked list
  ListNode* findMiddle(ListNode* head) {
  ListNode* slow = head;
  ListNode* fast = head;
  ListNode* prev = nullptr;

  while (fast && fast->next) {
     prev = slow;
     slow = slow->next;
     fast = fast->next->next;
  }

  if (prev) prev->next = nullptr; // Split the list into two halves
  return slow;
}
```

```
// Function to merge two sorted linked lists
ListNode* merge(ListNode* l1, ListNode* l2) {
    if (!l1) return l2;
    if (!l2) return l1;

    if (l1->val < l2->val) {
        l1->next = merge(l1->next, l2);
        return l1;
    } else {
        l2->next = merge(l1, l2->next);
        return l2;
    }
}
// Function to sort the linked list using merge sort
ListNode* sortList(ListNode* head) {
    if (!head || !head->next) return head; // Base case: 0 or 1 node

    ListNode* mid = findMiddle(head); // Find middle
    ListNode* left = sortList(head);  // Sort first half
    ListNode* right = sortList(mid);  // Sort second half

    return merge(left, right); // Merge both sorted halves
}
    };
```

4.Output:

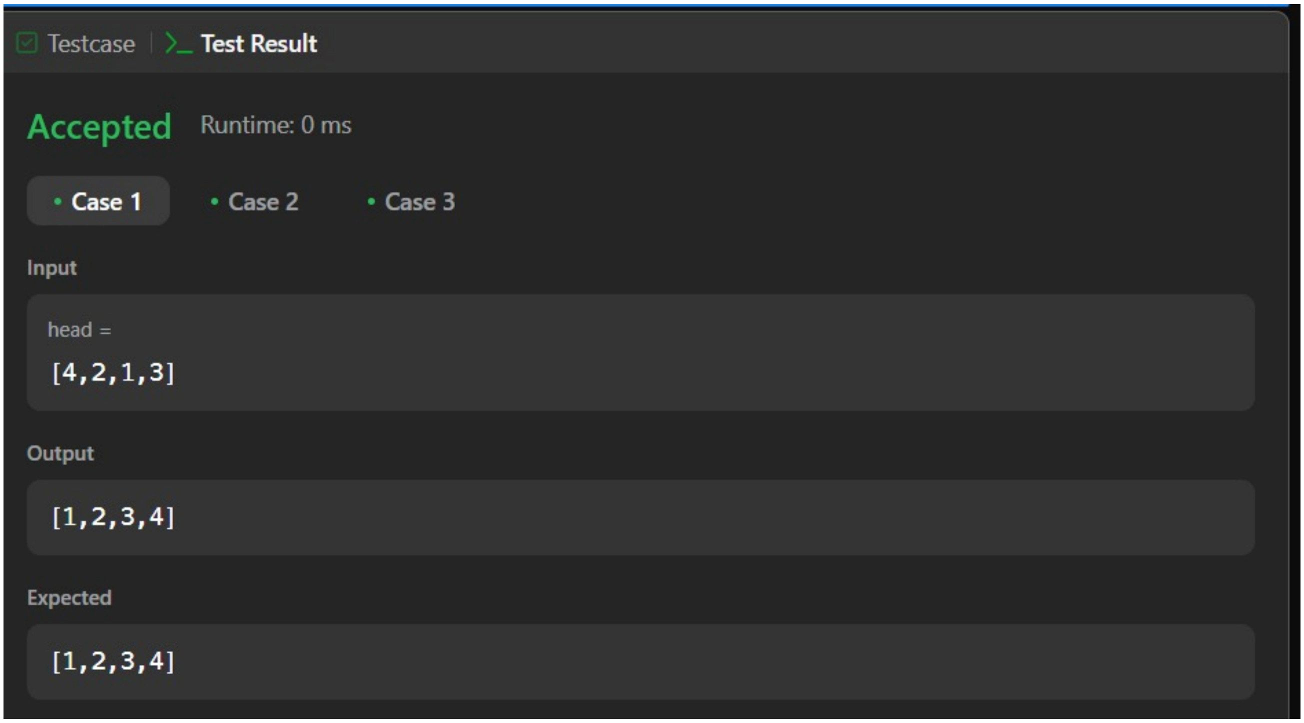

Fig.5:Sort List HW