Experiment-3

Student Name: Shubham **UID: 22BCS15853**
Branch: BE-CSE **Section/Group: KRG 2 B**
Semester: 6th **Date of Performance: 22/01/2025**
Subject Name: Advanced Programming Lab-2 **Subject Code: 22CSP-351**

Program 1:- Remove Duplicates from Sorted List II

1. **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.

2. **Objective:** The objective of this task is to implement a function that processes a sorted linked list to remove all nodes containing duplicate values, ensuring that only distinct values remain. The function should traverse the linked list, identify duplicates, and construct a new linked list that retains the sorted order of the original list. The solution should operate efficiently with a time complexity of O(n) and handle edge cases, such as empty lists or lists where all nodes are duplicates. Ultimately, the goal is to return a linked list that accurately reflects the unique values from the original input.

3. **Algorithm:**

   - **Create a Placeholder:** Initialize a `startNode` (or placeholder) to simplify handling edge cases and set `prev` to `startNode`.
   - **Set Current Pointer:** Point `current` to the head of the original linked list.
   - **Iterate Through the List:** Use a loop to traverse the linked list while `current` is not null.
   - **Detect Duplicates:** Check if `current` has duplicates by comparing `current.val` with `current.next.val`.
   - **Skip Duplicates:** If duplicates are found, move `current` forward until all duplicates are skipped.
   - **Update Previous Node:** If duplicates were found, connect `prev.next` to `current.next`. If no duplicates, move `prev` to `current`.
   - **Return the Result:** After the loop, return `startNode.next`, which points to the new linked list containing only distinct values.

4. Code:

```
import java.util.Arrays;
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) {
    val = x;     }
    public static ListNode deserialize(String data) {
        data = data.replaceAll("[\\[\\] ]", "");
        String[] values = data.split(",");
        if (values.length == 0 || (values.length == 1 && values[0].isEmpty())) {
        return null;       }
        ListNode head = new ListNode(Integer.parseInt(values[0]));
        ListNode current = head;
        for (int i = 1; i < values.length; i++) {
        if (!values[i].isEmpty()) {
        current.next = new ListNode(Integer.parseInt(values[i]));
        current = current.next;   }   }
        return head;  }   }
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
    ListNode startNode = new ListNode(0);
    startNode.next = head;
    ListNode prev = startNode;
    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; }
    else {
    prev = prev.next; }
    head = head.next;  }
    return startNode.next;  }
    public static void printLinkedList(ListNode head) {
    ListNode current = head;
    while (current != null) {
    System.out.print(current.val + " -> ");
    current = current.next;  }
    System.out.println("null");  }
    public static void main(String[] args) {
    String serializedList = "[1,2,3,3,4,4,5]";
    ListNode head = ListNode.deserialize(serializedList);
    System.out.println("Original linked list:");
    printLinkedList(head);
    Solution solution = new Solution();
    ListNode newHead = solution.deleteDuplicates(head);
    System.out.println("Linked list after removing duplicates:");
    printLinkedList(newHead);     }
}
```

5. Output:

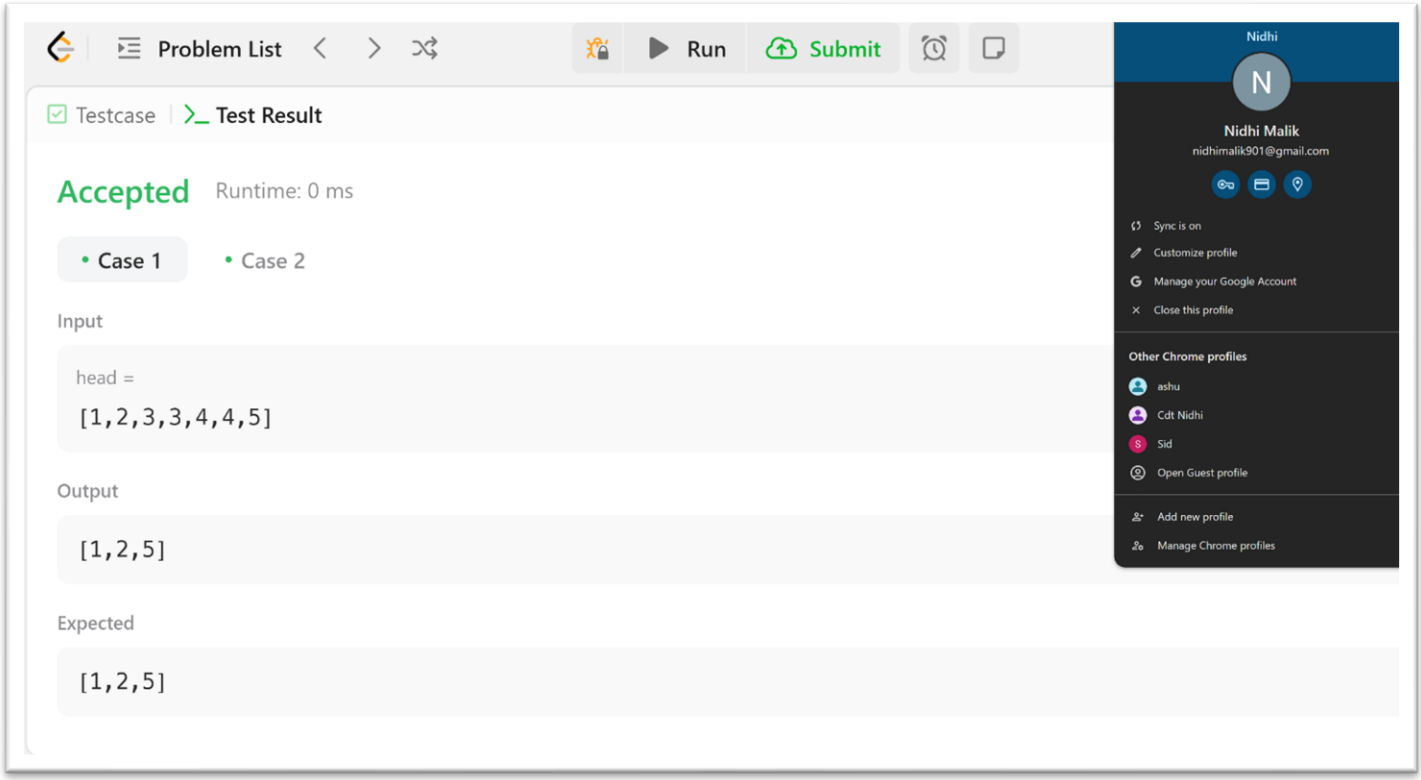

Figure.1

Program 2 :- LRU Cache

1. **Aim:** Design a data structure that follows the constraints of a **Least Recently Used (LRU) cache**. Implement the LRUCache class:
   - LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
   - int get(int key) Return the value of the key if the key exists, otherwise return -1.
   - void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

2. **Objective:** The objective of this task is to design and implement a Least Recently Used (LRU) cache that efficiently manages key-value pairs with a limited capacity. The cache should support two primary operations: retrieving a value by key (**get**) and adding or updating a key-value pair (**put**). Both operations must run in O(1) average time complexity. When the cache reaches its capacity, it should evict the least recently used item to make space for new entries. This implementation is crucial for optimizing memory usage in scenarios where data access patterns exhibit temporal locality.

3. **Algorithm:**

- **Initialize Structures:** Create a HashMap for key-node pairs and a doubly linked list for order.
- **Node Class:** Define a `Node` class with key, value, and pointers to previous and next nodes.
- **Constructor:** Set up the LRU cache with a specified capacity and dummy head/tail nodes.
- **Get Operation:** Check if the key exists; if yes, move the node to the head and return its value; if no, return -1.
- **Put Operation:** If the key exists, update its value and move it to the head; if not, create a new node and add it to the head**.**
- **Eviction Logic:** If adding a new node exceeds capacity, remove the tail node (least recently used) and delete it from the HashMap.
- **Maintain Order:** Implement methods to add nodes to the head, remove nodes, and move accessed nodestothehead.

4. Code:

```
import java.util.HashMap;
class LRUCache {
    private class Node {
        int key;
        int value;
        Node prev;
        Node next;
        Node(int key, int value) {
            this.key = key;
            this.value = value; }  }
    private final int capacity;
    private final HashMap<Integer, Node> cache;
    private Node head;
    private Node tail;
    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
        this.head = new Node(0, 0);
        this.tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;   }
    public int get(int key) {
        if (!cache.containsKey(key)) {
        return -1;   }
        Node node = cache.get(key);
        moveToHead(node);
        return node.value;   }
    public void put(int key, int value) {
        if (cache.containsKey(key)) {
            Node node = cache.get(key);
            node.value = value;
            moveToHead(node);
```

```
        } else {
            Node newNode = new Node(key, value);
            cache.put(key, newNode);
            addNode(newNode);
            if (cache.size() > capacity) {
                Node tailNode = popTail();
                cache.remove(tailNode.key);  }   }  }
    private void addNode(Node node) {
        node.prev = head;
        node.next = head.next;
        head.next.prev = node;
        head.next = node;  }
    private void removeNode(Node node) {
        Node prevNode = node.prev;
        Node nextNode = node.next;
        prevNode.next = nextNode;
        nextNode.prev = prevNode; }
    private void moveToHead(Node node) {
        removeNode(node);
        addNode(node);  }
     private Node popTail() {
       Node res = tail.prev;
        removeNode(res);
        return res;  }  }
public class Main {
    public static void main(String[] args) {
        LRUCache lruCache = new LRUCache(2);
        lruCache.put(1, 1);
        lruCache.put(2, 2);
        System.out.println(lruCache.get(1));
        lruCache.put(3, 3);
        System.out.println(lruCache.get(2));
        lruCache.put(4, 4);
        System.out.println(lruCache.get(1));
```

```
System.out.println(lruCache.get(3));
System.out.println(lruCache.get(4)); } }
```

5. Output:

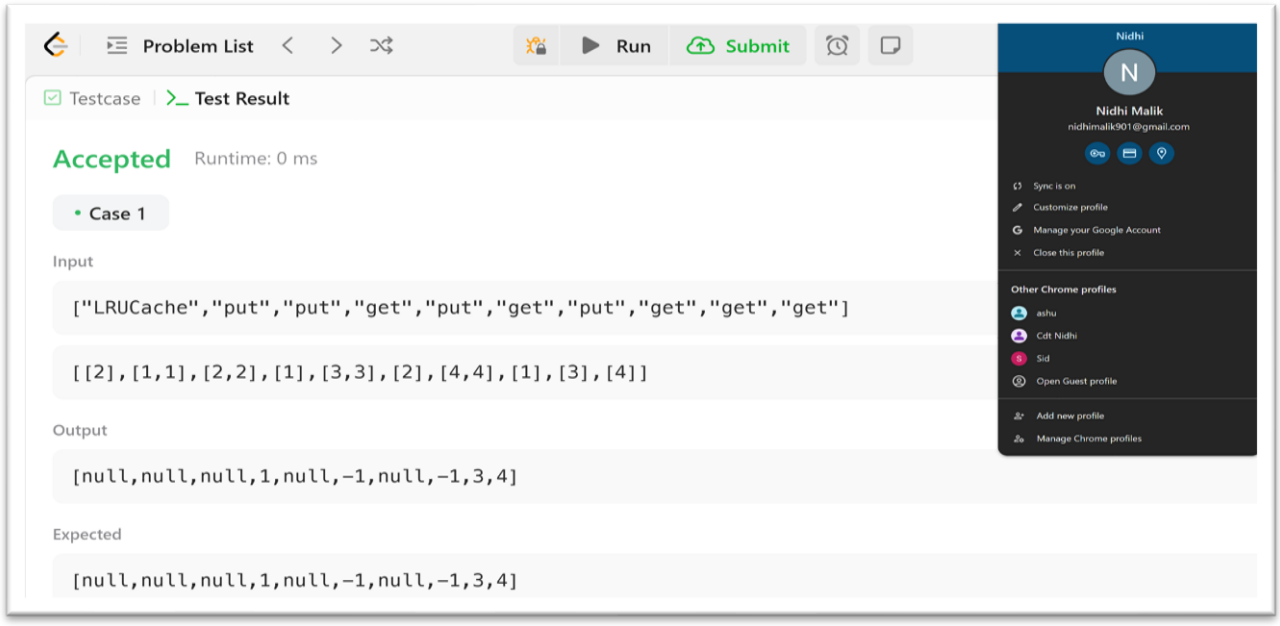

Figure.2

6. Learning Outcomes:

1. Understanding Cache Mechanisms: Gain insights into how caching works, including the concepts of hit and miss rates, and the importance of managing memory efficiently in applications.

2. Data Structure Proficiency: Develop proficiency in using advanced data structures, specifically the combination of HashMap and doubly linked lists, to achieve efficient data retrieval and management.

3. Algorithm Design Skills: Enhance skills in designing algorithms that meet specific performance criteria, such as O(1) time complexity for both retrieval and insertion operations.

4. Problem-Solving Techniques: Improve problem-solving abilities by tackling real-world scenarios where data needs to be stored and accessed efficiently, particularly in systems with limited resources.

5. Implementation of Object-Oriented Principles: Learn to apply object-oriented programming principles, such as encapsulation and abstraction, in designing a class that effectively models the behavior of an LRU cache.