DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
Discover. Learn. Empower.

Experiment - 3

Student Name **Rajat Pandey** UID **22BCS14428**

Branch: **BE - CSE** Section/Group **641 - A**

Semester: **6** Sub Code: **22CSP-351**

Subject Name: **Advanced Programming Lab - 2** Date: **14/01/2025**

Problem - 1

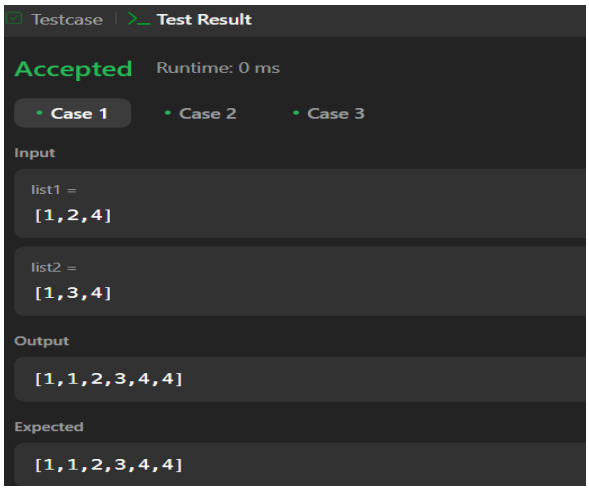
Aim - You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

Objective - To merge two sorted linked lists into a single sorted linked list by splicing together their nodes.

Implementation/Code -

```
ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
    if (!list1) return list2;
    if (!list2) return list1;
    if (list1->val <= list2->val) {
        list1->next = mergeTwoLists(list1->next, list2);
        return list1;
    } else {
        list2->next = mergeTwoLists(list1, list2->next);
        return list2;
    }
}
```

Output -



Time Complexity - O(n + m)

Space Complexity - O(n + m)

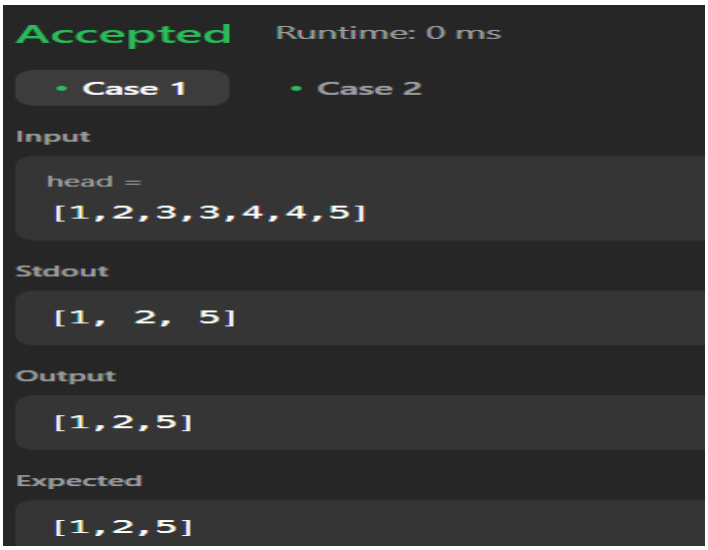
Aim - Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

Objective - To remove all nodes with duplicate numbers from a sorted linked list, leaving only distinct numbers, and return the sorted linked list.

Implementation/Code -

```
ListNode* deleteDuplicates(ListNode* head) {
    ListNode* dummy = new ListNode(0, head);
    ListNode* prev = dummy;
    while (head) {
        if (head->next && head->val == head->next->val) {
            while (head->next && head->val == head->next->val) {
                head = head->next;
            }
            prev->next = head->next;
        } else {
            prev = prev->next;
        }
        head = head->next;
    }
    return dummy->next;
}
```

Output -



Time Complexity - O(n) **Space Complexity -** O(1)

Aim - Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

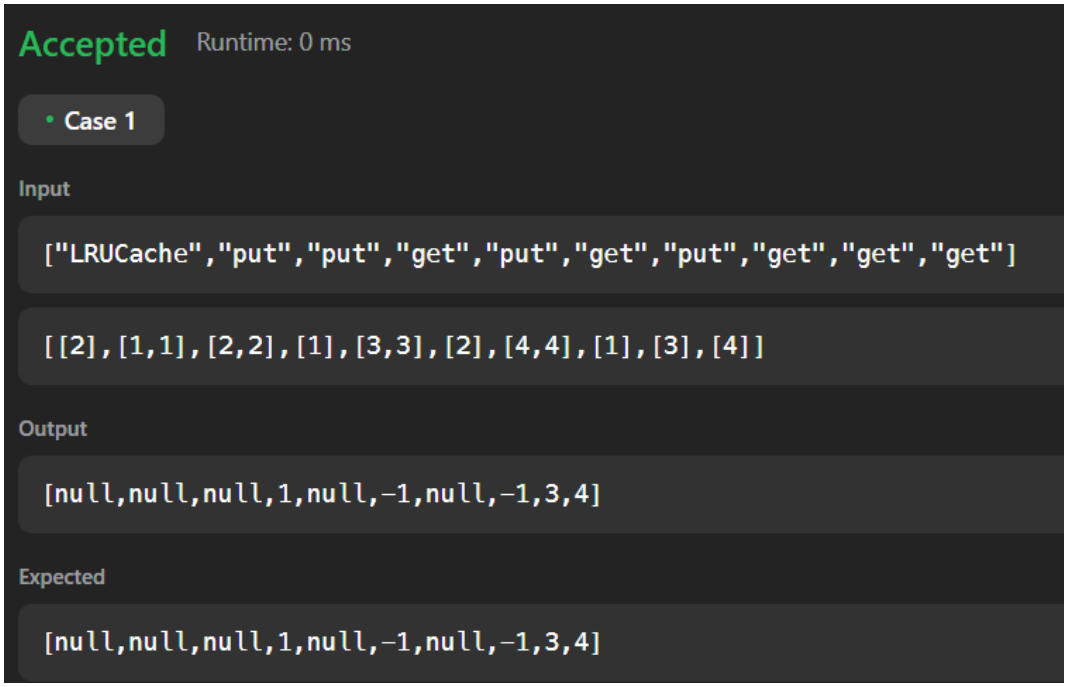
Objective - To design a data structure that implements the functionality of an LRU (Least Recently Used) cache, supporting efficient insertion, retrieval, and removal of least recently used elements.

Implementation/Code -

```
class LRUCache {
    int capacity;
    list<pair<int, int>> cache;
    unordered_map<int, list<pair<int, int>>::iterator> map;

public:
    LRUCache(int capacity) : capacity(capacity) {}

    int get(int key) {
        if (map.find(key) == map.end()) return -1;
        cache.splice(cache.begin(), cache, map[key]);
        return map[key]->second;
```

```
    }

    void put(int key, int value) {
        if (map.find(key) != map.end()) {
            cache.splice(cache.begin(), cache, map[key]);
            map[key]->second = value;
        } else {
            if (cache.size() == capacity) {
                int lru = cache.back().first;
                cache.pop_back();
                map.erase(lru);
            }
            cache.emplace_front(key, value);
            map[key] = cache.begin();
        }
    }
};
```

Output -



Time Complexity - O(1)

Space Complexity - O(n)

Learning Outcomes -

1. Understand how to merge two sorted linked lists into one sorted list.
2. Learn to remove duplicate nodes from a sorted linked list while maintaining order.
3. Gain knowledge of designing an efficient LRU cache using data structures like hash maps and linked lists.
4. Develop skills in implementing algorithms with optimal time and space complexities.
5. Strengthen problem-solving abilities by manipulating linked lists and managing memory efficiently.