**Experiment 3**

**Student Name: Navkirat Singh UID: 22BET10232**

**Branch: IT Section/Group: 22BET-701/A**

**Semester: 6th Date of Performance:05.02.25**

**Subject Name: AP Lab - 2 Subject Code: 22ITP-351**

1. **Aim:**

To implement the concept of Linked List by solving the problems on LeetCode.

* 1. Print Linked list
  2. Remove duplicates from a sorted list
  3. Reverse a linked list
  4. Delete middle node of a list
  5. Merge two sorted linked list
  6. Remove duplicates from sorted linked lists
  7. Detect a cycle in a linked list
  8. Reverse linked list 2
  9. Rotate a list
  10. Merge k sorted lists
  11. Sort List

1. **Objective:**

* Traverse and display all elements of the linked list.
* Remove consecutive duplicate values to retain unique elements.
* Reverse the order of nodes by modifying pointers.
* Identify and remove the middle node efficiently.
* Combine two sorted lists into a single sorted list.
* Ensure only distinct elements remain in the list.
* Detect if a cycle exists using slow and fast pointer techniques.
* Reverse a specific section of the list between given positions.
* Shift nodes to the right by a given number of positions.
* Efficiently merge multiple sorted lists into one.
* Sort the linked list with an optimal time complexity.

1. **Code:**

# Problem 1: Print linked list

class Solution {

public:

// Function to display the elements of a linked list in same line void printList(Node \*head) {

Node\*temp=head; while(temp!=NULL){

cout<<temp->data<<" "; temp=temp->next;

}

}

};

# Problem 2: [Remove Duplicates from Sorted List](https://leetcode.com/problems/remove-duplicates-from-sorted-list/)

class Solution { public:

ListNode\* deleteDuplicates(ListNode\* head) { if(!head) return nullptr; ListNode\*current=head;

while(current->next){

if(current->val==current->next->val){ ListNode\*temp=current->next; current->next=current->next->next; delete temp;

}else{

current=current->next;

}

}

return head;

}

};

# Problem 3: Reverse Linked List

class Solution { public:

ListNode\* reverseList(ListNode\* head) { ListNode\*prev=nullptr; ListNode\*current=head; ListNode\*next=nullptr; while(current!=nullptr){

next=current->next; current->next=prev;

prev=current; current=next;

}

return prev;

}

};

# Problem 4: [Delete the Middle Node of a Linked List](https://leetcode.com/problems/delete-the-middle-node-of-a-linked-list/)

class Solution { public:

ListNode\* deleteMiddle(ListNode\* head) {

if (!head || !head->next) // If list is empty or has only one node return nullptr;

ListNode \*slow = head, \*fast = head, \*prev = nullptr;

// Move fast pointer twice as fast as slow pointer while (fast && fast->next) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

// Remove the middle node if (prev)

prev->next = slow->next; delete slow; // Free memory return head;

}

};

# Problem 5: Merge two sorted linked list

class Solution { public:

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) { if (!list1) return list2; // If list1 is empty, return list2

if (!list2) return list1; // If list2 is empty, return list1

ListNode\* dummy = new ListNode(-1); // Dummy node for ease of merging ListNode\* current = dummy;

while (list1 && list2) {

if (list1->val <= list2->val) {

current->next = list1; list1 = list1->next;

} else {

current->next = list2; list2 = list2->next;

}

current = current->next;

}

// Attach the remaining nodes of the non-empty list current->next = list1 ? list1 : list2;

return dummy->next; // Return merged list (skip dummy node)

}

};

# Problem 6: Remove duplicates from sorted list II

class Solution { public:

ListNode\* deleteDuplicates(ListNode\* head) { if (!head) return nullptr;

ListNode\* dummy = new ListNode(0, head); // Dummy node before head ListNode\* prev = dummy; // Pointer to track nodes before duplicate sequence

while (head) {

if (head->next && head->val == head->next->val) {

// Skip all nodes with the same value

while (head->next && head->val == head->next->val) { head = head->next;

}

prev->next = head->next; // Remove all duplicates

} else {

prev = prev->next; // Move prev if no duplicate

}

head = head->next; // Move head forward

}

return dummy->next; // Return new head (skip dummy node)

}

};

# Problem 7: Linked List cycle

class Solution { public:

bool hasCycle(ListNode \*head) {

if (!head || !head->next) return false; // No cycle if empty or single node ListNode \*slow = head, \*fast = head;

while (fast && fast->next) {

slow = slow->next; // Move slow by 1 step fast = fast->next->next; // Move fast by 2 steps

if (slow == fast) return true; // Cycle detected

}

return false; // No cycle

}

};

# Problem 8: Reverse Linked list II

class Solution { public:

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

if (!head || left == right) return head; // No need to reverse if empty or one node

ListNode\* dummy = new ListNode(0); // Dummy node before head dummy->next = head;

ListNode\* prev = dummy;

// Move prev to the node just before "left" for (int i = 1; i < left; i++) {

prev = prev->next;

}

ListNode\* current = prev->next; // First node to be reversed ListNode\* next = nullptr;

// Reverse the sublist from left to right

for (int i = 0; i < right - left; i++) { next = current->next;

current->next = next->next; next->next = prev->next; prev->next = next;

}

return dummy->next; // Return modified list (skip dummy node)

}

};

**Problem 9: Rotate list** class Solution { public:

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

if (!head || !head->next || k == 0) return head; // Edge case

// Step 1: Find the length of the linked list int length = 1;

ListNode\* tail = head; while (tail->next) {

tail = tail->next; length++;

}

// Step 2: Optimize k to prevent unnecessary full rotations k = k % length;

if (k == 0) return head; // No change if k is a multiple of length

// Step 3: Find the new tail (length - k - 1) and new head (length - k) ListNode\* newTail = head;

for (int i = 1; i < length - k; i++) { newTail = newTail->next;

}

// Step 4: Perform rotation

ListNode\* newHead = newTail->next; newTail->next = nullptr; // Break the list

tail->next = head; // Connect old tail to old head

return newHead; // Return new head

}

};

# Problem 10: Merge k sorted lists

class Solution { public:

struct Compare {

bool operator()(ListNode\* a, ListNode\* b) {

return a->val > b->val; // Min-Heap based on node values

}

};

ListNode\* mergeKLists(vector<ListNode\*>& lists) { priority\_queue<ListNode\*, vector<ListNode\*>, Compare> minHeap;

// Push the first node of each list into the heap for (auto list : lists) {

if (list) minHeap.push(list);

}

ListNode\* dummy = new ListNode(0); // Dummy node to simplify result list ListNode\* tail = dummy; // Tail to keep track of merged list

while (!minHeap.empty()) {

ListNode\* smallest = minHeap.top(); minHeap.pop();

tail->next = smallest; // Attach to merged list tail = tail->next; // Move tail forward

if (smallest->next) {

minHeap.push(smallest->next); // Push next node of extracted list

}

}

return dummy->next; // Return merged list

}

};

**Problem 11: Sort List** class Solution { public:

// Function to merge two sorted linked lists ListNode\* merge(ListNode\* l1, ListNode\* l2) {

if (!l1) return l2; if (!l2) return l1;

ListNode\* dummy = new ListNode(0); ListNode\* tail = dummy;

while (l1 && l2) {

if (l1->val < l2->val) { tail->next = l1;

l1 = l1->next;

} else {

tail->next = l2; l2 = l2->next;

}

tail = tail->next;

}

if (l1) tail->next = l1; if (l2) tail->next = l2;

return dummy->next;

}

// Function to find the middle node and split the list ListNode\* sortList(ListNode\* head) {

if (!head || !head->next) return head; // Base case

// Find the middle using slow & fast pointer ListNode\* slow = head, \*fast = head, \*prev = nullptr; while (fast && fast->next) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

prev->next = nullptr; // Split the list into two halve

// Recursively sort both halves ListNode\* left = sortList(head); ListNode\* right = sortList(slow);

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

}

};

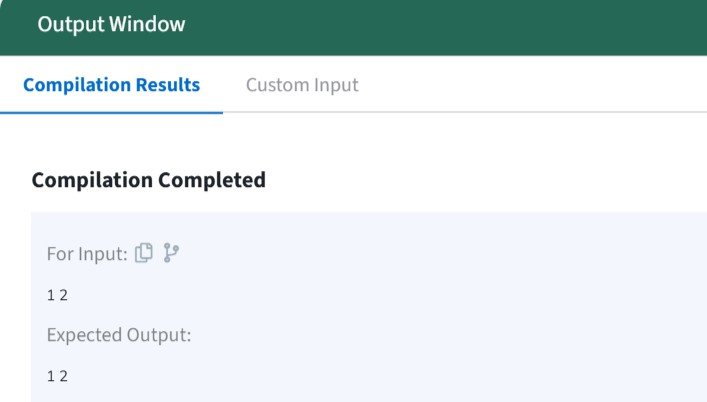
1. **Output:**

Fig 1. Print linked list

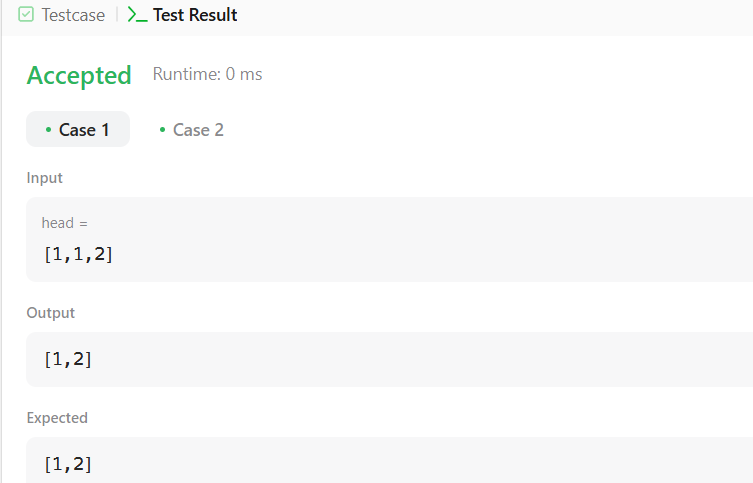


Fig 2. [Remove Duplicates from Sorted List](https://leetcode.com/problems/remove-duplicates-from-sorted-list/)



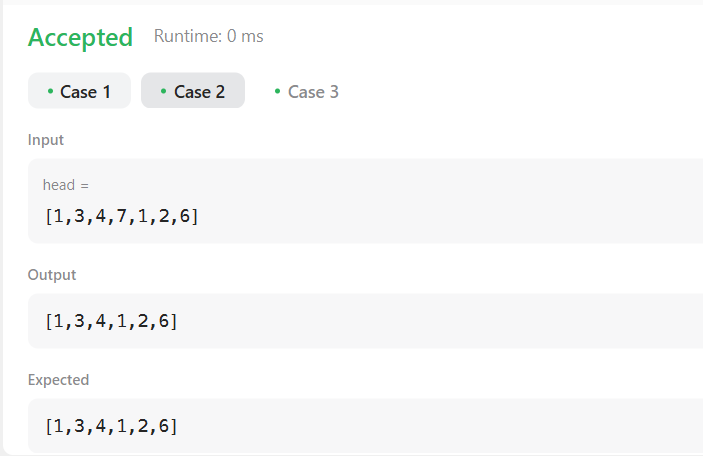
Fig 3. Reverse linked list

Fig 4. Delete the middle node of a list



Fig 5. Merge two sorted list

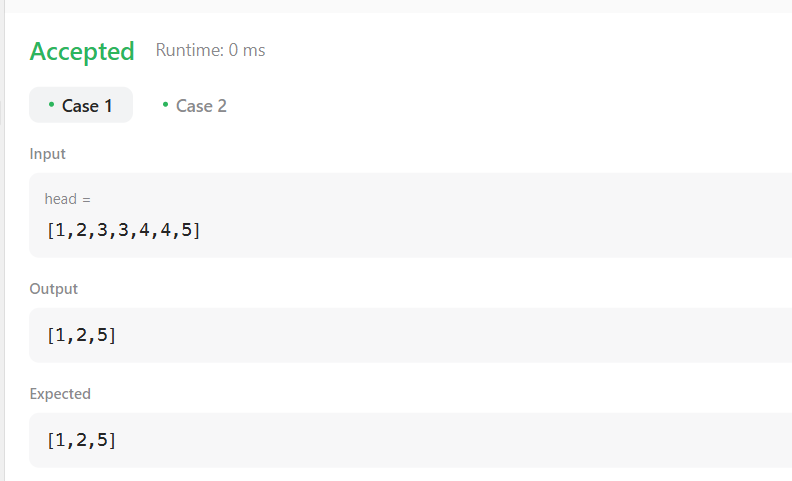


Fig 6. Remove duplicates from a list

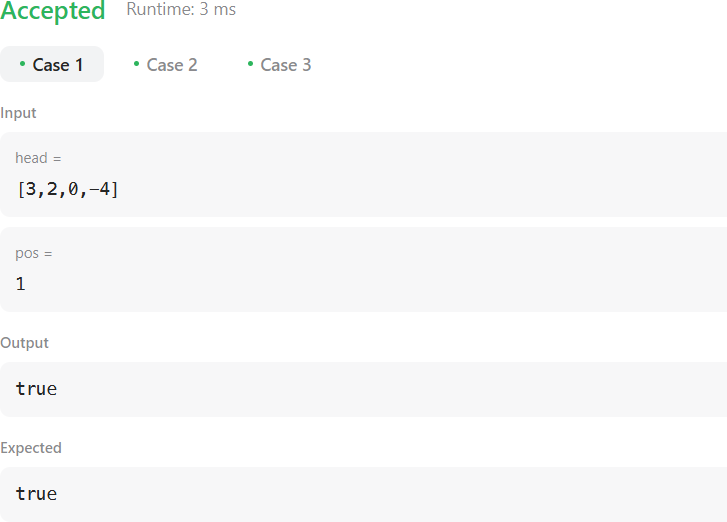


Fig 7. Linked list cycle

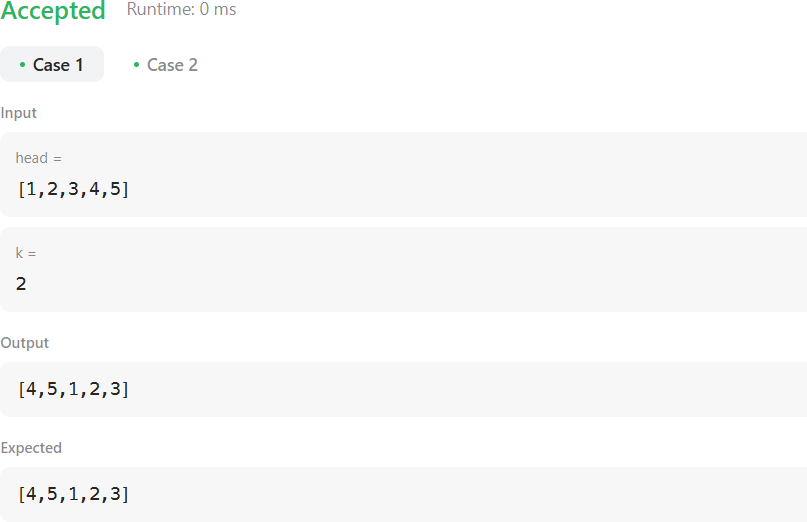


Fig 9. Rotate list



Fig 10. Merge k sorted list

1. **Learning Outcomes:**

* Understand and implement fundamental linked list operations, including traversal, insertion, deletion, and modification.
* Develop efficient algorithms to detect and remove duplicates, reverse a list, and merge multiple sorted lists.
* Apply advanced techniques such as Floyd’s cycle detection and merge sort for optimized linked list processing.
* Enhance problem-solving skills by working with pointer manipulation and recursion in linked list-based algorithms.
* Gain hands-on experience in optimizing linked list operations for real-world applications, improving time and space complexity.