ASSIGNMENT

Student Name: Archi Bansal **UID: 22BCS15264**

Branch: BE-CSE **Section/Group: 608/B**

Semester: 6th **Subject Name: AP LAB**

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) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    }
```

My Submissions All Submissions

Refresh

Time (IST)	Status	Marks	Lang	Test Cases	Code
2025-02-21 11:02:18	Correct	1	cpp	1112 / 1112	View

2. **Remove duplicates from a sorted list**

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        ListNode* current = head;

        while (current && current->next) {
            if (current->val == current->next->val) {
                ListNode* duplicate = current->next;
                current->next = current->next->next; // Remove duplicate node
                delete duplicate; // Free memory
            } else {
                current = current->next; // Move to next node
            }
        }

        return head;
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted Mar 05, 2025	C++	0 ms	16.1 MB	

3. Reverse a linked list

```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* current = head;

        while (current) {
            ListNode* nextNode = current->next; // Store next node
            current->next = prev; // Reverse the link
            prev = current; // Move prev to current node
            current = nextNode; // Move to next node
        }

        return prev; // New head of reversed list
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted Mar 05, 2025	C++	0 ms	13.4 MB	

4. Delete middle node of a list

```cpp
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if (!head || !head->next) return nullptr;
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }
        prev->next = slow->next;
        delete slow;
        return head;
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted Mar 06, 2025	C++	0 ms	312.1 MB	

5. Merge two sorted linked lists

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        // Handle empty cases early
        if (!l1) return l2;
        if (!l2) return l1;

        // Initialize the head of the merged list
        ListNode* head = nullptr;

        // Ensure the head points to the smaller of the two list heads
        if (l1->val <= l2->val) {
            head = l1;
            l1 = l1->next;
        } else {
            head = l2;
            l2 = l2->next;
        }

        // Use a pointer to track the current position in the merged list
        ListNode* current = head;

        // Merge the two lists
        while (l1 && l2) {
            if (l1->val <= l2->val) {
                current->next = l1;
                l1 = l1->next;
            } else {
                current->next = l2;
                l2 = l2->next;
            }
            current = current->next;  // Move the current pointer forward
        }

        // Append the remaining part of whichever list is not exhausted
        if (l1) {
            current->next = l1;
        } else {
            current->next = l2;
        }

        return head;  // Return the merged list
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted 12 hours ago	C++	0 ms	19.5 MB	

6. Detect a cycle in a linked list

```cpp
class Solution {
public:
    bool hasCycle(ListNode* head) {
        // If the list is empty or has only one node, it cannot have a cycle
        if (!head || !head->next) {
            return false;
        }

        ListNode* slow = head;
        ListNode* fast = head;

        // Move slow by 1 step and fast by 2 steps
        while (fast && fast->next) {
            slow = slow->next;          // Move slow by 1 step
            fast = fast->next->next;    // Move fast by 2 steps

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

        return false;  // No cycle detected
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted 12 hours ago	C++	12 ms	11.8 MB	

7. Rotate a list

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        // Edge case: if the list is empty or has only one node
        if (!head || !head->next || k == 0) {
            return head;
        }

        // Step 1: Find the length of the list
        ListNode* current = head;
        int length = 1;  // start from 1 to count the head
        while (current->next) {
            current = current->next;
            length++;
        }

        // Step 2: Reduce k to a smaller number (since rotating by k == n is the
        k = k % length;
        if (k == 0) {
            return head;  // no need to rotate if k % length == 0
        }

        // Step 3: Make the list circular by connecting the last node to the head
        current->next = head;

        // Step 4: Find the new tail node (which is at position n - k % n - 1)
        for (int i = 0; i < length - k - 1; i++) {
            head = head->next;
        }

        // Step 5: The new head is the next node of the current head
        ListNode* newHead = head->next;
        head->next = nullptr;  // Break the circular list

        return newHead;
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted 12 hours ago	C++	0 ms	16.5 MB	

8. Sort List

```cpp
class Solution {
public:
    // Merge two sorted linked lists
    ListNode* merge(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode(0); // Dummy node to simplify merge logic
        ListNode* current = dummy;

        // Merge the two sorted lists
        while (l1 && l2) {
            if (l1->val < l2->val) {
                current->next = l1;
                l1 = l1->next;
            } else {
                current->next = l2;
                l2 = l2->next;
            }
            current = current->next;
        }

        // Append any remaining nodes
        if (l1) current->next = l1;
        if (l2) current->next = l2;

        return dummy->next;  // Return the merged list starting from the first va
    }

    // Sort the linked list using Merge Sort
    ListNode* sortList(ListNode* head) {
        // Base case: if the list is empty or has one node, it's already sorted
        if (!head || !head->next) return head;

        // Step 1: Find the middle of the list using the fast and slow pointer ap
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* prev = nullptr;

        // Move fast pointer two steps and slow pointer one step at a time
        while (fast && fast->next) {
            prev = slow;
            slow = slow->next;
            fast = fast->next->next;
        }

        // Split the list into two halves
        prev->next = nullptr;  // Break the list into two parts

        // Step 2: Recursively sort both halves
        ListNode* left = sortList(head);
        ListNode* right = sortList(slow);

        // Step 3: Merge the two sorted halves
        return merge(left, right);
    }
};
```

Description | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted 12 hours ago	C++	46 ms	75.8 MB	

9. Merge k Sorted Lists

```cpp
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        // Lambda function for min-heap (priority queue)
        auto compare = [](ListNode* a, ListNode* b) {
            return a->val > b->val;  // Min-heap (smallest element first)
        };

        priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> minHeap(compare);

        // Insert all head nodes into the heap
        for (auto list : lists) {
            if (list) minHeap.push(list);
        }

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

        // Process the min-heap
        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;
    }
};
```

Description | Accepted × | Editorial | Solutions | **Submissions**

	Status	Language	Runtime	Memory	Notes
1	Accepted a few seconds ago	C++	1 ms	18.6 MB	