ASSIGNMENT

Student Name: Archi Bansal UID: 22BCS15264

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

Semester: 6th Subject Name: AP LAB

1. Print Linked List

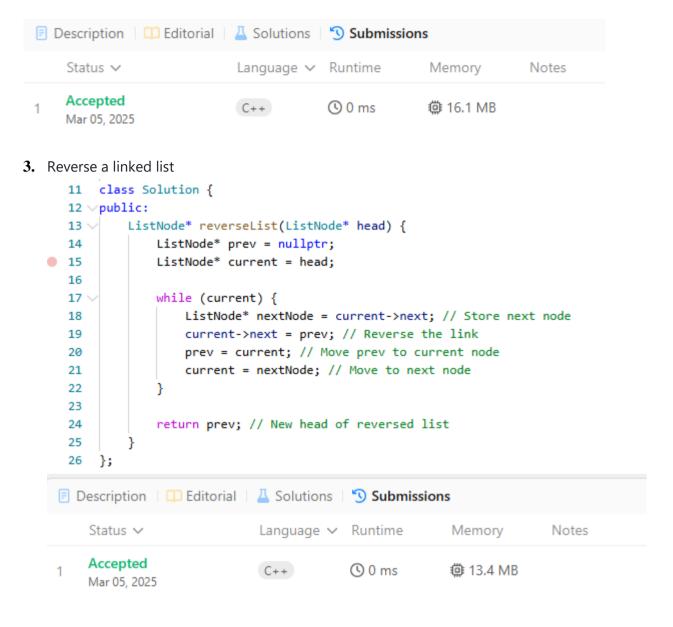
```
class Solution {
  public:
    // Function to display the elements of a linked list in same line
    void printList(Node *head) {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " ";
            temp = temp->next;
        }
    }
}
```

My Submissions All Submissions



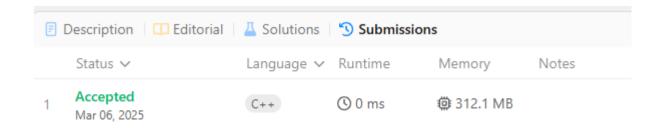
2. Remove duplicates from a sorted list

```
class Solution {
11
    public:
12
        ListNode* deleteDuplicates(ListNode* head) {
13
            ListNode* current = head;
14
15
16
            while (current && current->next) {
17
                if (current->val == current->next->val) {
18
                    ListNode* duplicate = current->next;
19
                    current->next = current->next->next; // Remove duplicate node
                    delete duplicate; // Free memory
20
21
                 } else {
22
                     current = current->next; // Move to next node
23
24
25
26
             return head;
27
28
    };
29
```



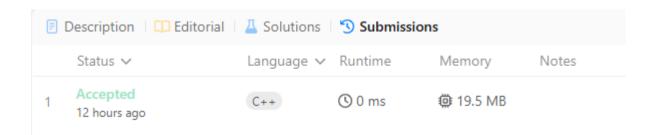
4. Delete middle node of a list

```
class Solution {
12
13
    public:
        ListNode* deleteMiddle(ListNode* head) {
14
15
             if (!head | | !head->next) return nullptr;
16
             ListNode* slow = head;
             ListNode* fast = head;
17
             ListNode* prev = nullptr;
18
             while (fast && fast->next) {
19
                 prev = slow;
20
21
                 slow = slow->next;
                 fast = fast->next->next;
22
23
24
             prev->next = slow->next;
25
             delete slow;
26
             return head;
27
28
    };
```



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 = 11;
            l1 = l1->next;
        } else {
            head = 12;
            12 = 12->next;
        // Use a pointer to track the current position in the merged list
        ListNode* current = head;
        // Merge the two lists
        while (11 && 12) {
            if (l1->val <= l2->val) {
                current->next = 11;
                l1 = l1 \rightarrow next;
            } else {
                current->next = 12;
                12 = 12->next;
            current = current->next; // Move the current pointer forward
        // Append the remaining part of whichever list is not exhausted
        if (11) {
            current->next = 11;
        } else {
            current->next = 12;
        return head; // Return the merged list
};
```



6. Detect a cycle in a linked list

```
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 V
                          Language ∨ Runtime
                                                    Memory
                                                                 Notes
   Accepted
                          C++
                                      (12 ms
                                                    @ 11.8 MB
    12 hours ago
```

7. Rotate a list

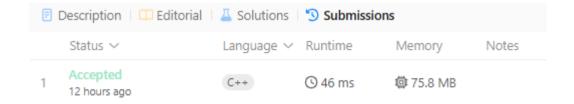
```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        // Edge case: if the list is empty or has only one node
        if (!head || !head->next || k == 0) {
            return head:
        // Step 1: Find the length of the list
        ListNode* current = head;
        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 | ⑤ Submissions

    Status V
                               Language ∨ Runtime
                                                             Memory
                                                                             Notes
    Accepted
                                             ( 0 ms
                                                             @ 16.5 MB
                               C++
    12 hours ago
```

8. Sort List

```
class Solution (
public:
    // Merge two sorted linked lists
    ListNode+ merge(ListNode+ 11, ListNode+ 12) {
       ListNode* dummy = new ListNode(0); // Dummy node to simplify merge logic
       ListNode* current = dummy;
       // Merge the two sorted lists
       while (11 && 12) {
           if (ll->val < l2->val) {
               current->next = 11;
               11 = 11-\text{next};
           } else {
               current->next = 12;
               12 = 12->next;
            current = current->next;
       // Append any remaining nodes
       if (ll) current->next = ll;
       if (12) current->next = 12;
       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);
1:
```



9. Merge k Sorted Lists

```
class Solution {
 12
 13
     public:
         ListNode* mergeKLists(vector<ListNode*>& lists) {
 14
             // Lambda function for min-heap (priority queue)
 15
 16
             auto compare = [](ListNode* a, ListNode* b) {
 17
                 return a->val > b->val; // Min-heap (smallest element first)
 18
             };
 19
             priority_queue<ListNode*, vector<ListNode*>, decltype(compare)> minHeap(compare);
 20
 21
 22
             // Insert all head nodes into the heap
             for (auto list : lists) {
 23
 24
                 if (list) minHeap.push(list);
 25
 26
             ListNode dummy(0); // Dummy node to simplify list construction
 27
 28
             ListNode* tail = &dummy;
 29
 30
             // Process the min-heap
 31
             while (!minHeap.empty()) {
 32
                 ListNode* node = minHeap.top();
 33
                 minHeap.pop();
 34
                 tail->next = node;
                 tail = tail->next;
 35
 36
 37
                 if (node->next) {
 38
                     minHeap.push(node->next);
 39
 40
 41
 42
             return dummy.next;
 43
 44
     };

■ Description | ⑤ Accepted × | □ Editorial | △ Solutions | ⑤ Submissions
    Status ~
                              Language V Runtime
                                                            Memory
                                                                            Notes
    Accepted
                              C++
                                            (§ 1 ms
                                                            @ 18.6 MB
    a few seconds ago
```