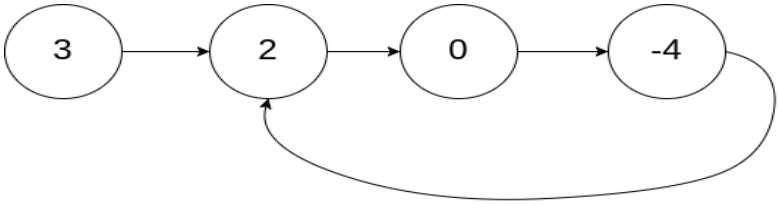
Problem 142 Linked List Cycle-2

Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Example 1:

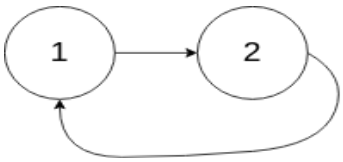


Input: head = [3,2,0,-4], pos = 1

Output: tail connects to node index 1

Explanation: There is a cycle in the linked list, where tail connects to the second node.

Example 2:

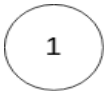


Input: head = [1,2], pos = 0

Output: tail connects to node index 0

Explanation: There is a cycle in the linked list, where tail connects to the first node.

Example 3:



Input: head = [1], pos = -1

Output: no cycle

Explanation: There is no cycle in the linked list.

Constraints:

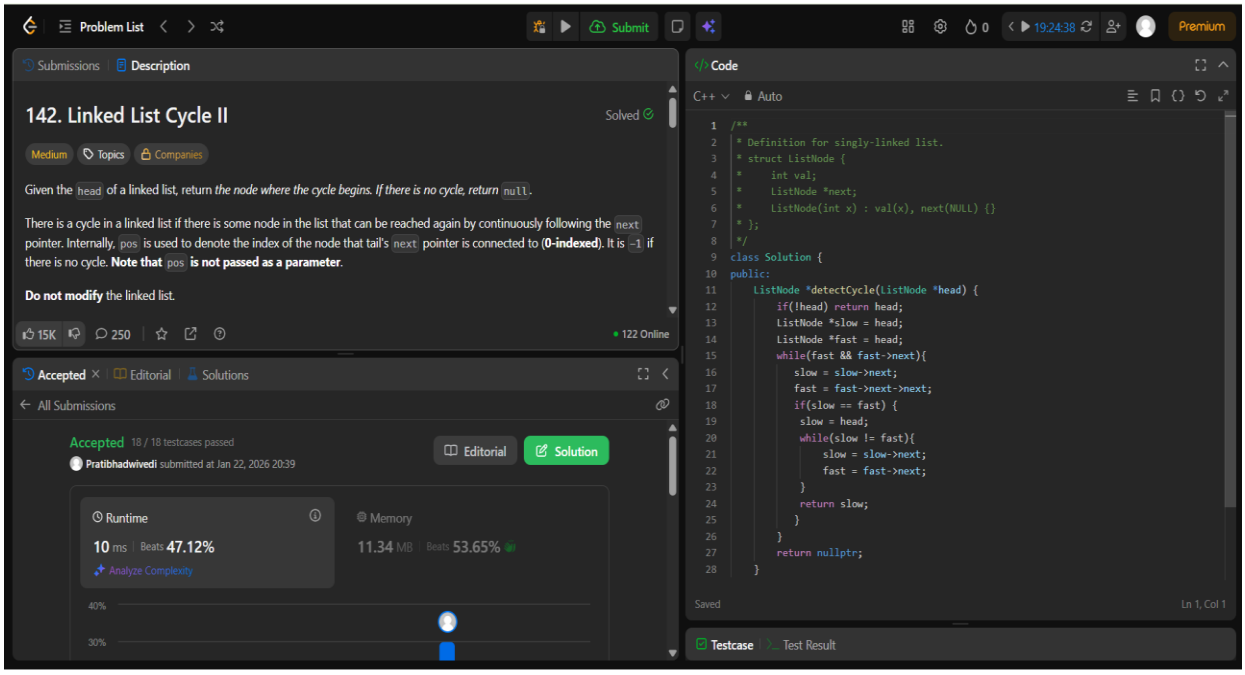
- The number of the nodes in the list is in the range [0, 10^4].

- -10^5 <= Node.val <= 10^5
- pos is -1 or a valid index in the linked-list.

Solution

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if(!head) return head;
        ListNode *slow = head;
        ListNode *fast = head;
        while(fast && fast->next){
          slow = slow->next;
          fast = fast->next->next;
          if(slow == fast) {
           slow = head;
           while(slow != fast){
              slow = slow->next;
              fast = fast->next;
           }
           return slow;
          }
        }
        return nullptr;
    }
```

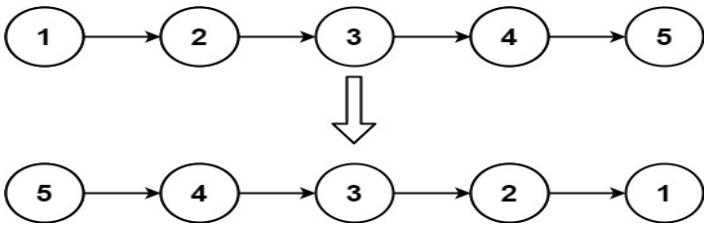
};



Problem 206. Reverse Linked List

Given the head of a singly linked list, reverse the list, and return *the reversed list*.

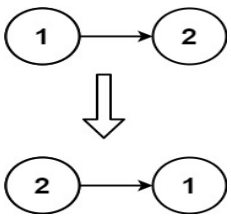
Example 1:



Input: head = [1,2,3,4,5]

Output: [5,4,3,2,1]

Example 2:



Input: head = [1,2]

Output: [2,1]

Example 3:

Input: head = []

Output: []

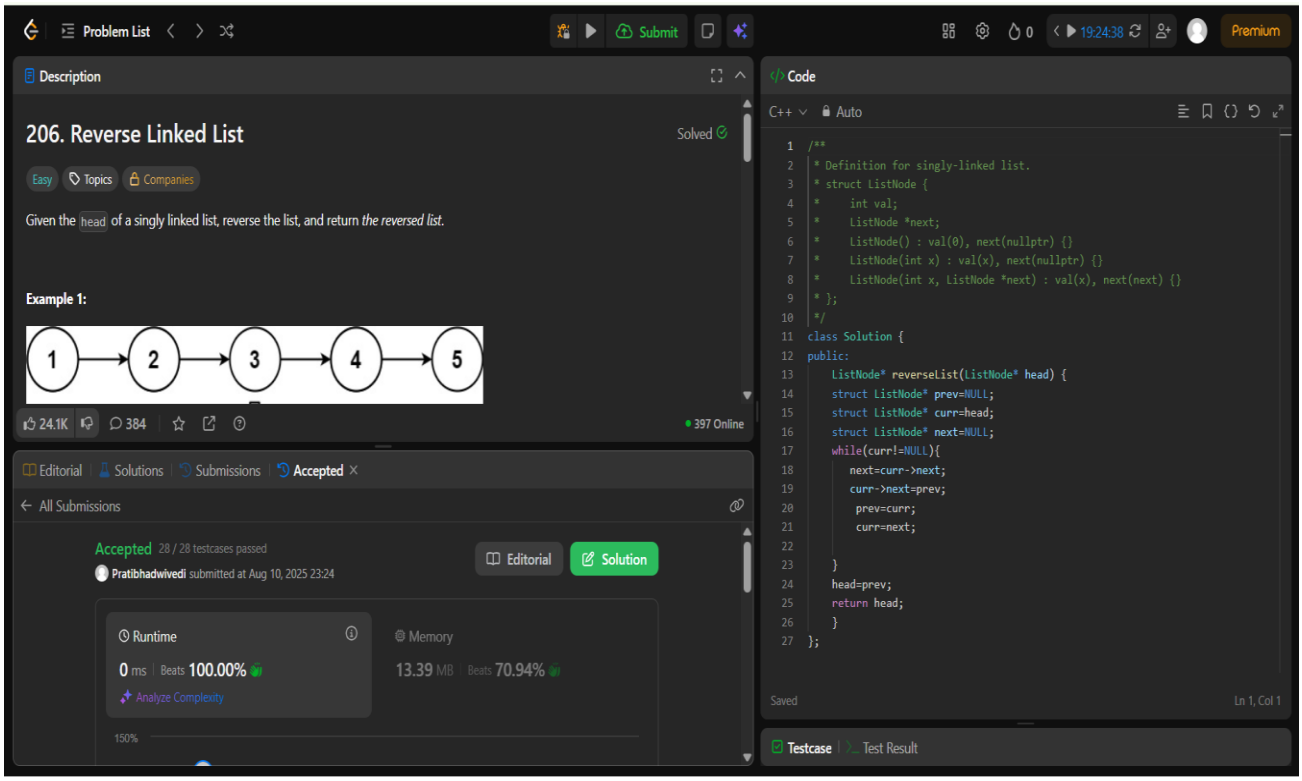
Constraints:

- The number of nodes in the list is the range [0, 5000].
- -5000 <= Node.val <= 5000

Solution:

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
    struct ListNode* prev=NULL;
    struct ListNode* curr=head;
    struct ListNode* next=NULL;
    while(curr!=NULL){
      next=curr->next;
      curr->next=prev;
       prev=curr;
       curr=next;
```

```
    }
    head=prev;
    return head;
    }
};
```



Implementation of Doubly Linked List:

```
#include <iostream>
using namespace std;

// Node structure
struct Node {
    int data;
    Node* prev;
    Node* next;

    // Constructor
    Node(int val) {
        data = val;
        prev = NULL;
```

```
        next = NULL;
    }
};

// Head pointer
Node* head = NULL;

// Insert a node at the beginning
void insertatbeginning(int value) {
    Node* newNode = new Node(value);
    newNode->next = head;

    if (head != NULL) {
        head->prev = newNode;
    }

    head = newNode;
}

// Insert a node at the end
void insertatend(int value) {
    Node* newNode = new Node(value);

    if (head == NULL) {
        head = newNode;
        return;
    }

    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = newNode;
    newNode->prev = temp;
}

// Insert a node at a given position (1-based)
void insertatposition(int value, int pos) {
    if (pos <= 0) return;

    if (pos == 1) {
        insertatbeginning(value);
        return;
    }

    Node* temp = head;
    int count = 1;
```

```
    while (temp != NULL && count < pos - 1) {
        temp = temp->next;
        count++;
    }

    if (temp == NULL) return;

    Node* newNode = new Node(value);
    newNode->prev = temp;
    newNode->next = temp->next;

    if (temp->next != NULL) {
        temp->next->prev = newNode;
    }

    temp->next = newNode;
}

// Delete the first node
void deletefrombeginning() {
    if (head == NULL) return;

    Node* temp = head;
    head = head->next;

    if (head != NULL) {
        head->prev = NULL;
    }

    delete temp;
}

// Delete the last node
void deletefromend() {
    if (head == NULL) return;

    Node* temp = head;

    if (temp->next == NULL) {
        delete temp;
        head = NULL;
        return;
    }

    while (temp->next != NULL) {
        temp = temp->next;
    }
```

```
    temp->prev->next = NULL;
    delete temp;
}

// Delete a node by its value
void deletebyvalue(int value) {
    if (head == NULL) return;

    Node* temp = head;

    if (temp->data == value) {
        deletefrombeginning();
        return;
    }

    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }

    if (temp == NULL) return;

    temp->prev->next = temp->next;

    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }

    delete temp;
}

// Display the list
void display() {
    Node* temp = head;

    while (temp != NULL) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }

    cout << "NULL\n";
}

// Main function (simple testing)
int main() {
    insertatbeginning(10);
    insertatend(20);
    insertatend(30);
```

```
    insertatposition(15, 2);

    display();   // 10 <-> 15 <-> 20 <-> 30 <-> NULL

    deletefrombeginning();
    display();   // 15 <-> 20 <-> 30 <-> NULL

    deletefromend();
    display();   // 15 <-> 20 <-> NULL

    deletebyvalue(20);
    display();   // 15 <-> NULL

    return 0;
}
```