



Experiment-3

Student Name: Garv Kumar

UID: 22BET10103

Branch: BE-IT

Section/Group: 22BET_IOT-702/A

Semester: 6th

Date of Performance: 30/01/2025

Subject Name: Advanced Programming Lab-2

Subject Code: 22ITP-351

Problem-1

1. Aim:

To develop and comprehend a program that identifies cycles in a linked list using Floyd's Cycle Detection Algorithm, also known as the Tortoise and Hare Algorithm.

2. Objective:

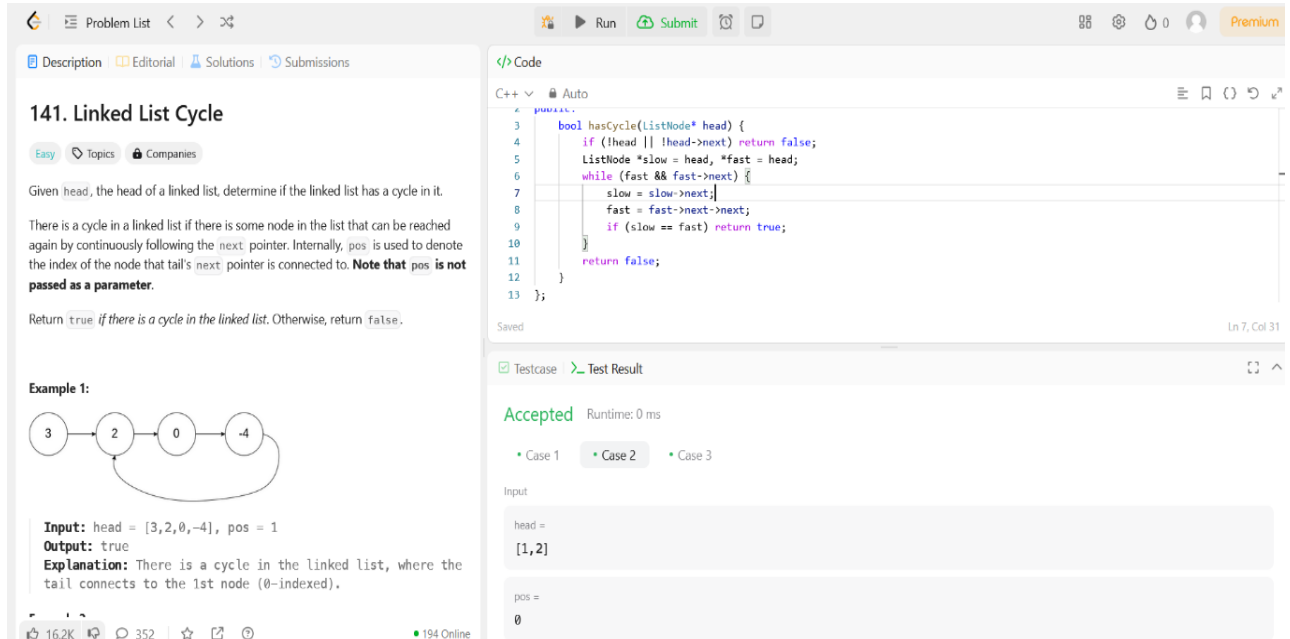
- To enhance the ability to detect and resolve issues related to cyclic linked lists.
- To strengthen expertise in implementing data structure algorithms using C++.

3. Implementation:

```
class Solution {  
public:  
    bool hasCycle(ListNode* head) {  
        if (!head || !head->next) return false;  
        ListNode *slow = head, *fast = head;  
        while (fast && fast->next) {  
            slow = slow->next;  
            fast = fast->next->next;  
            if (slow == fast) return true;  
        }  
        return false;  
    }  
}
```

```
};
```

4. Output:



141. Linked List Cycle

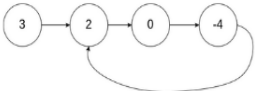
Easy Topics Companies

Given head, the head of a linked list, determine if the linked list has a cycle in it.

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. **Note that pos is not passed as a parameter.**

Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:



Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

16.2K 352 194 Online

```
bool hasCycle(ListNode* head) {
    if (!head || !head->next) return false;
    ListNode *slow = head, *fast = head;
    while (fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) return true;
    }
    return false;
}
```

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

head =
[1,2]

pos =
0

Fig: Linked List Cycle.

Problem-2

1. Aim:

To develop a program that reverses a designated segment of a singly linked list between specified positions left and right.

2. Objective:

- 1 To gain proficiency in traversing and modifying a singly linked list using pointers efficiently.
- 2 To develop a clear understanding of in-place reversal of a sublist within a linked list.

3. Implementation:

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int left, int right) {
        if (!head || left == right) return head;
```

Discover. Learn. Empower.

```
ListNode dummy(0);
dummy.next = head;
ListNode* prev = &dummy;
```

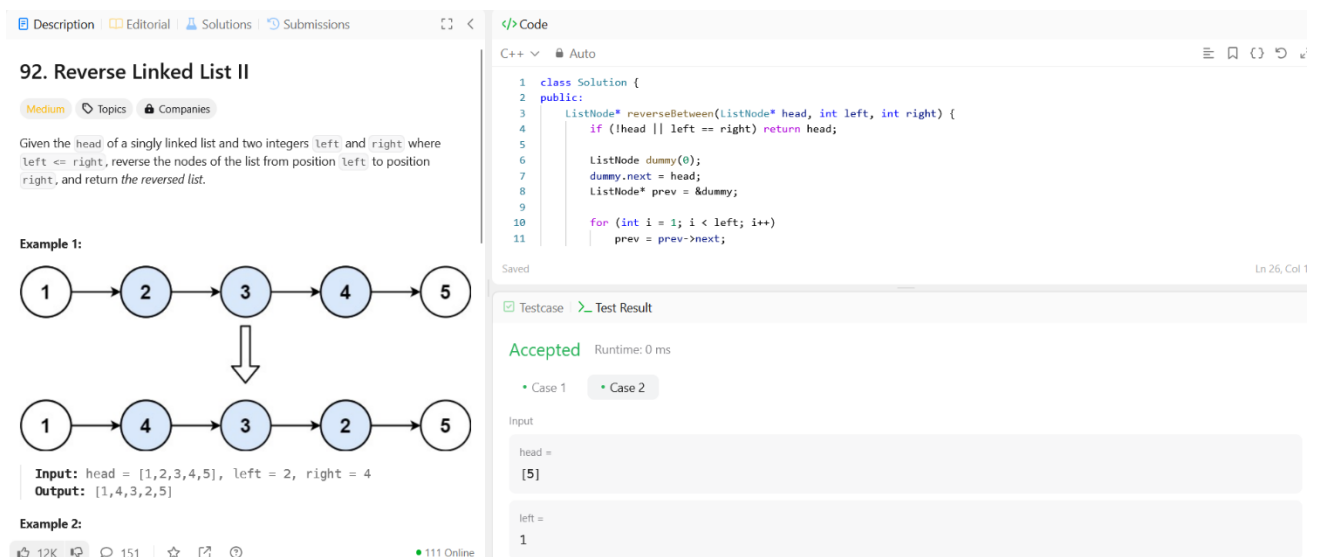
```
for (int i = 1; i < left; i++)
    prev = prev->next;
```

```
ListNode* curr = prev->next;
ListNode* nextNode = nullptr;
```

```
for (int i = 0; i < right - left; i++) {
    nextNode = curr->next;
    curr->next = nextNode->next;
    nextNode->next = prev->next;
    prev->next = nextNode;
}
```

```
return dummy.next;
}
};
```

4. Output:



The screenshot displays a coding problem titled "92. Reverse Linked List II" on a platform. The problem description states: "Given the head of a singly linked list and two integers left and right where left <= right, reverse the nodes of the list from position left to position right, and return the reversed list." Example 1 shows a linked list [1, 2, 3, 4, 5] being reversed from index 2 to 4, resulting in [1, 4, 3, 2, 5]. The input is head = [1, 2, 3, 4, 5], left = 2, right = 4, and the output is [1, 4, 3, 2, 5]. Example 2 is also shown. On the right, the C++ code is displayed, which implements the solution using a dummy node and a loop to reverse the specified range. The code is accepted, with a runtime of 0 ms. The input fields show head = [5] and left = 1.

Fig: Reverse Linked List II.

Problem-3**1. Aim:**

To develop a program that efficiently rotates a singly linked list to the right by a specified number of positions, while effectively managing edge cases and large inputs.

2. Objective:

- 1 To grasp the concept of rotating a singly linked list and its implementation.
- 2 To gain experience in modifying linked lists in-place, including creating and breaking circular connections.

3. Implementation:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if (!head || !head->next || k == 0) return head;

        ListNode* temp = head;
        int len = 1;

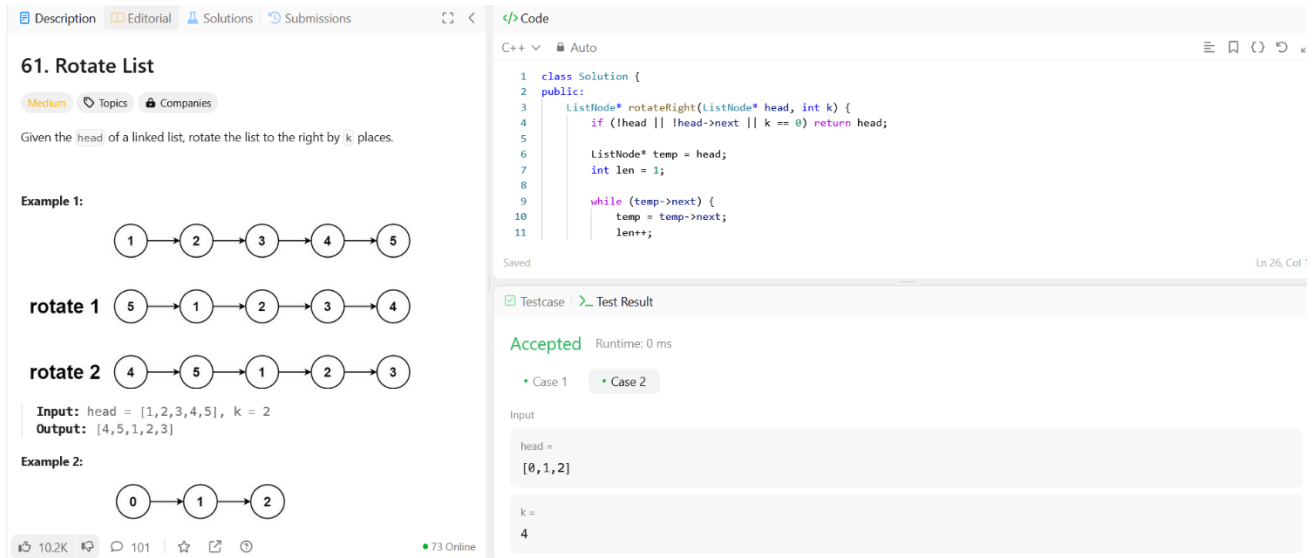
        while (temp->next) {
            temp = temp->next;
            len++;
        }

        temp->next = head;
        k %= len;
        int steps = len - k;

        while (steps--) temp = temp->next;

        head = temp->next;
        temp->next = nullptr;
        return head;
    }
};
```

4. Output:



61. Rotate List

Medium Topics Companies

Given the `head` of a linked list, rotate the list to the right by `k` places.

Example 1:

rotate 1

rotate 2

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

Example 2:

Input: `head = [0, 1, 2], k = 4`
Output: `[1, 2, 0, 1, 2]`

```

1 class Solution {
2 public:
3     ListNode* rotateRight(ListNode* head, int k) {
4         if (!head || !head->next || k == 0) return head;
5         ListNode* temp = head;
6         int len = 1;
7         while (temp->next) {
8             temp = temp->next;
9             len++;
10        }
11    }
12 }

```

Accepted Runtime: 0 ms

Case 1 Case 2

Input

head =
[0, 1, 2]

k =
4

Fig: Rotate List.

Problem-4

1. Aim:

To design an efficient algorithm that merges k sorted linked lists into one sorted list, utilizing priority queues or divide-and-conquer methods.

2. Objective:

- 1 To understand the process of merging multiple sorted linked lists into a single sorted list.
- 2 To implement efficient merging techniques, such as using a min-heap or a divide-and-conquer approach.

3. Implementation:

```

#include <queue>
#include <vector>
class Solution {
public:
    struct Compare {
        bool operator()(ListNode* a, ListNode* b) {
            return a->val > b->val;
        }
    }

```

};

```
ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    std::priority_queue<ListNode*, std::vector<ListNode*>, Compare> pq;

    for (auto list : lists) {
        if (list) pq.push(list);
    }
    ListNode dummy(0);
    ListNode* tail = &dummy;

    while (!pq.empty()) {
        ListNode* node = pq.top();
        pq.pop();
        tail->next = node;
        tail = node;
        if (node->next) pq.push(node->next);
    }
    return dummy.next;
};
```

4. Output:

Description Editorial Solutions Submissions

23. Merge k Sorted Lists

Hard Topics Companies

You are given an array of `k` linked-lists `lists`, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

Input: `lists = [[1,4,5],[1,3,4],[2,6]]`
Output: `[1,1,2,3,4,4,5,6]`
Explanation: The linked-lists are:

```
[
  1->4->5,
  1->3->4,
  2->6
]
```

merging them into one sorted list:
`1->1->2->3->4->4->5->6`

Example 2:

Input: `lists = []`
Output: `[]`

20.1K 243 277 Online

Code

C++ Auto

```
1 #include <queue>
2 #include <vector>
3
4 class Solution {
5 public:
6     struct Compare {
7         bool operator()(ListNode* a, ListNode* b) {
8             return a->val > b->val;
9         }
10    };
11
```

Saved Ln 33, C

Testcase Test Result

Accepted Runtime: 0 ms

Case 1 Case 2 Case 3

Input

`lists =`
`[[1,4,5],[1,3,4],[2,6]]`

Output

`[1,1,2,3,4,4,5,6]`

Fig: Rotate List.

Problem-5

1. Aim:

To develop an algorithm that efficiently sorts a singly linked list in ascending order using techniques such as merge sort or quick sort.

2. Objective:

- 1 To explore the challenges and strategies involved in sorting linked lists compared to arrays.
- 2 To learn and implement efficient sorting algorithms for linked lists, such as merge sort.

3. Implementation:

```
class Solution {
public:
    ListNode* merge(ListNode* left, ListNode* right) {
        ListNode dummy(0);
        ListNode* tail = &dummy;

        while (left && right) {
            if (left->val < right->val) {
                tail->next = left;
                left = left->next;
            } else {
                tail->next = right;
                right = right->next;
            }
            tail = tail->next;
        }

        tail->next = left ? left : right;
        return dummy.next;
    }

    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) return head;
```

Discover. Learn. Empower.

```

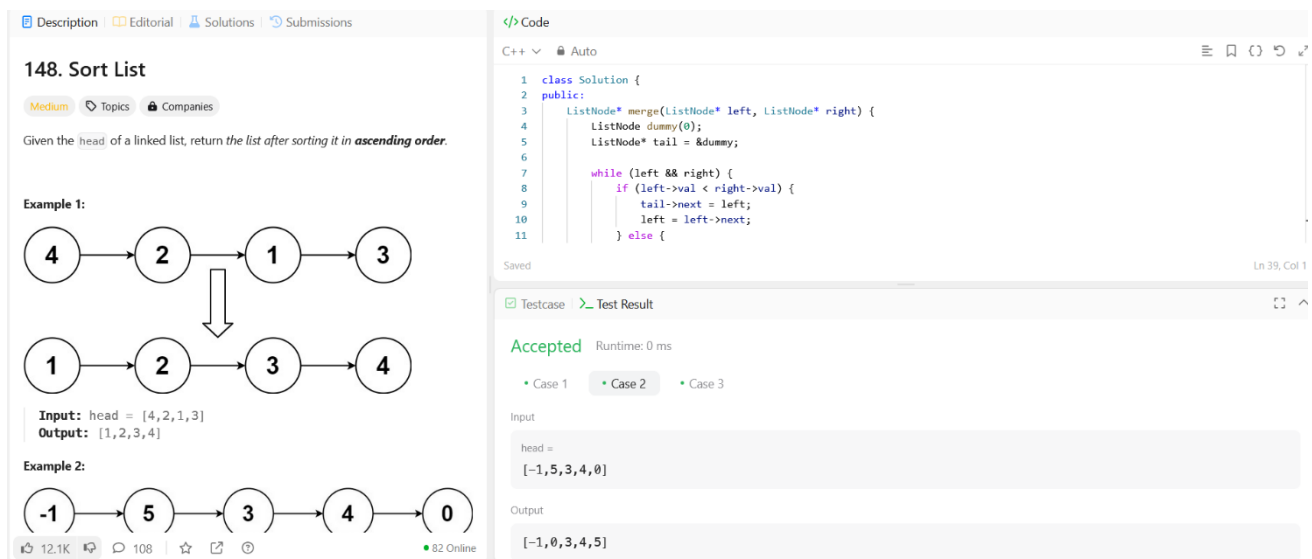
ListNode *slow = head, *fast = head, *prev = nullptr;
while (fast && fast->next) {
    prev = slow;
    slow = slow->next;
    fast = fast->next->next;
}

prev->next = nullptr;
ListNode* left = sortList(head);
ListNode* right = sortList(slow);

return merge(left, right);
};

```

4. Output:



The screenshot displays a coding problem titled "148. Sort List" on a platform. The problem description asks to sort a linked list in ascending order. Example 1 shows a linked list with nodes 4, 2, 1, 3 being transformed into 1, 2, 3, 4. Example 2 shows a linked list with nodes -1, 5, 3, 4, 0 being transformed into -1, 0, 3, 4, 5. The solution code is written in C++ and implements a merge sort algorithm for linked lists. The code defines a `merge` function that recursively splits the list and merges it back in sorted order. The test results show the solution is "Accepted" with a runtime of 0 ms for three test cases.

Fig: Sort List.

5. Learning Outcomes:

1. Detecting a Cycle in a Linked List (Floyd's Cycle Detection Algorithm)

- Understand how to efficiently detect cycles in a linked list using Floyd's Tortoise and Hare algorithm.
- Gain experience in implementing two-pointer techniques to optimize linked list operations.

2. Reversing a Sublist in a Singly Linked List

- Develop skills in modifying specific sections of a linked list in-place without using extra space.
- Learn to manipulate linked list pointers dynamically to achieve partial reversals efficiently.

3. Rotating a Singly Linked List to the Right

- Understand how to efficiently rotate a linked list while handling edge cases such as large rotations.
- Learn to optimize linked list operations by converting it into a circular list and breaking it at the correct position.

4. Merging k Sorted Linked Lists

- Gain proficiency in using priority queues (min-heaps) to merge multiple sorted linked lists efficiently.
- Explore the divide-and-conquer approach to recursively merge sorted lists with reduced time complexity.

5. Sorting a Singly Linked List (Merge Sort)

- Learn to implement merge sort on a linked list, leveraging its natural properties for efficient sorting.
- Understand the challenges of sorting linked lists compared to arrays and how to optimize pointer manipulation.