**Experiment-3**

**Student Name: Anushka UID: 22BET10126**

**Branch: BE-IT Section/Group: 22BET\_IOT-702/B**

**Semester: 6th Date of Performance: 31stJan, 2025**

**Subject Name: Advanced Programming Lab-2 Subject Code: 22ITP-351**

**Problem-1**

1. **Aim:** To determine whether a given linked list contains a cycle
2. **Objective:**

 To understand the concept of linked lists and cycle detection.

 To implement efficient algorithms for detecting cycles in a linked list.

 To optimize space complexity while checking for cycles in a linked list

1. **Code:**

#include <iostream>

using namespace std;

class Solution {

public:

bool hasCycle(ListNode \*head) {

if (head == nullptr || head->next == nullptr) return false;

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

while (fast != nullptr && fast->next != nullptr) {

slow = slow->next; // Move slow pointer one step

fast = fast->next->next; // Move fast pointer two steps

if (slow == fast) { // If they meet, cycle is detected

return true;

}

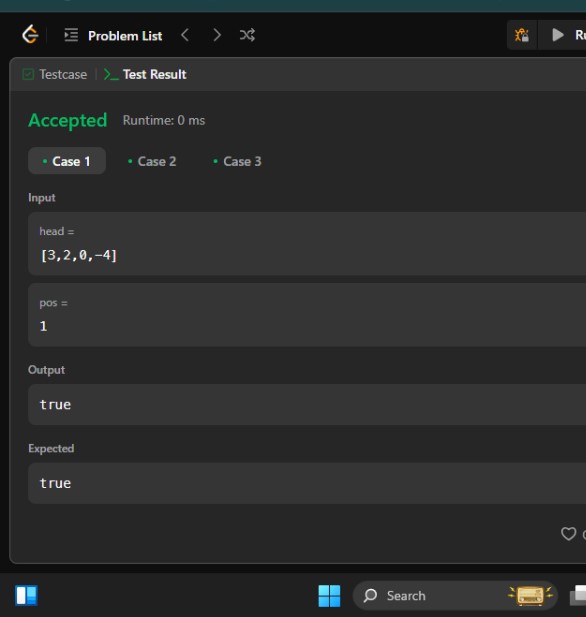
}

return false; // No cycle detected

}

};

1. **Output:**

****

**Fig.1. Cycle Detection**

1. **Learning Outcomes:**

* Ability to traverse a linked list and detect cycles using Floyd’s Algorithm.
* Understanding the working of slow and fast pointer techniques.
* Implementing an efficient algorithm with O(n) time complexity and O(1) space complexity.

**Problem-2**

1. **Aim:** To reverse a portion of a given singly linked list between positions left and right in **one pass.**
2. **Objective:**

 To understand and implement linked list operations.

 To reverse a specific section of a singly linked list efficiently.

 To solve the problem in **O(n) time complexity** and **O(1) space complexity**.

1. **Code:**

#include <iostream>

using namespace std;

class Solution {

public:

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

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

ListNode dummy(0); // Dummy node to simplify edge cases

dummy.next = head;

ListNode \*prev = &dummy;

// Move prev to the node just before the left position

for (int i = 1; i < left; i++) {

prev = prev->next;

}

ListNode\* current = prev->next;

ListNode\* nextNode = nullptr;

// Reverse the sublist from left to right

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

nextNode = current->next;

current->next = nextNode->next;

nextNode->next = prev->next;

prev->next = nextNode;

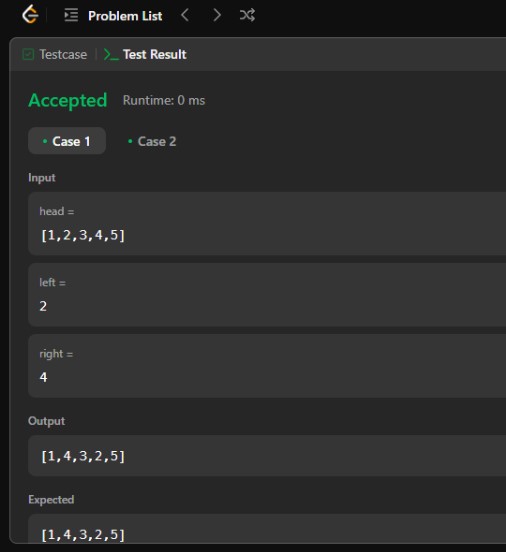
}

return dummy.next;

}

};

1. **Output:**

****

**Fig.2. Reversal**

1. **Learning Outcomes:**

 Understanding the linked list structure and pointer manipulation.

 Implementing in-place reversal of a portion of the list.

 Efficiently handling edge cases, such as when left == right.

**Problem-3**

1. **Aim:** To rotate a given singly linked list to the right by k places efficiently.
2. **Objective:**

 To understand linked list manipulations involving rotations.

 To solve the problem in **O(n) time complexity** using an optimized approach.

 To handle cases where k is larger than the length of the list.

1. **Code:**

#include <iostream>

using namespace std;

class Solution {

public:

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

if (!head || !head->next || k == 0) return head; // Edge cases: empty list, single node, no rotation

// Step 1: Compute the length of the list

ListNode\* temp = head;

int length = 1;

while (temp->next) {

temp = temp->next;

length++;

}

// Step 2: Make the list circular

temp->next = head; // Connect the tail to head

// Step 3: Find the new head position

k = k % length; // Reduce k if it's greater than length

int stepsToNewHead = length - k;

temp = head; // Reset temp to head

for (int i = 1; i < stepsToNewHead; i++) {

temp = temp->next; // Move temp to the new tail

}

// Step 4: Break the circular list and update head

head = temp->next;

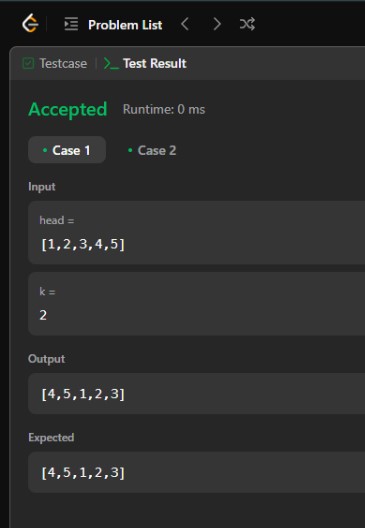
temp->next = nullptr;

return head;

}

};

1. **Output:**

****

**Fig.2. Rotation**

1. **Learning Outcomes:**

 Understanding **circular linked lists** and breaking loops.

 Implementing an **efficient** rotation method instead of shifting nodes one by one.

 Handling **edge cases** like k > length of list or empty list.

**Problem-4**

**1. Aim:** To merge k sorted linked lists into a single sorted linked list.

1. **Objective:**

 To use a **min-heap (priority queue)** for efficiently merging multiple sorted linked lists.

 To maintain **optimal time complexity** while merging the lists in **ascending order**.

1. **Code:**

#include <stdio.h>

#include <stdlib.h>

// Definition for singly-linked list (LeetCode provides this)

struct ListNode {

int val;

struct ListNode \*next;

};

// Comparison function for the min-heap (used with qsort)

int compare(const void \*a, const void \*b) {

return (\*(int \*)a - \*(int \*)b); // Ascending order

}

struct ListNode\* mergeKLists(struct ListNode\*\* lists, int listsSize) {

if (lists == NULL || listsSize == 0) {

return NULL;

}

// Array to store all node values for sorting

int\* values = (int\*)malloc(sizeof(int) \* 10000); // Adjust size as needed

int valuesSize = 0;

// Collect all values from the linked lists

for (int i = 0; i < listsSize; i++) {

struct ListNode\* current = lists[i];

while (current != NULL) {

values[valuesSize++] = current->val;

current = current->next;

}

}

// Sort the collected values

qsort(values, valuesSize, sizeof(int), compare);

// Create the merged linked list

struct ListNode\* dummy = (struct ListNode\*)malloc(sizeof(struct ListNode)); // Dummy head

dummy->next = NULL;

struct ListNode\* current = dummy;

for (int i = 0; i < valuesSize; i++) {

struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

newNode->val = values[i];

newNode->next = NULL;

current->next = newNode;

current = newNode;

}

free(values); // Free the allocated memory

return dummy->next;

}

// \*\*\*FOR TESTING ONLY - DO NOT INCLUDE ON LEETCODE SUBMISSION\*\*\*

struct ListNode\* createLinkedList() {

int val;

struct ListNode\* head = NULL;

struct ListNode\* tail = NULL;

printf("Enter list elements (enter -1 to end): ");

while (scanf("%d", &val) == 1 && val != -1) {

struct ListNode\* newNode = (struct ListNode\*)malloc(sizeof(struct ListNode));

newNode->val = val;

newNode->next = NULL;

if (head == NULL) {

head = newNode;

tail = newNode;

} else {

tail->next = newNode;

tail = newNode;

}

}

return head;

}

// \*\*\*FOR TESTING ONLY - DO NOT INCLUDE ON LEETCODE SUBMISSION\*\*\*

int main() {

int k;

printf("Enter the number of linked lists (k): ");

scanf("%d", &k);

struct ListNode\*\* lists = (struct ListNode\*\*)malloc(sizeof(struct ListNode\*) \* k);

for (int i = 0; i < k; ++i) {

printf("Enter elements for list %d:\n", i + 1);

lists[i] = createLinkedList();

}

struct ListNode\* mergedList = mergeKLists(lists, k);

printf("Merged List: ");

while (mergedList != NULL) {

printf("%d ", mergedList->val);

mergedList = mergedList->next;

}

printf("\n");

// Clean up memory (Important!)

for(int i = 0; i < k; i++) {

struct ListNode\* head = lists[i];

while (head) {

struct ListNode\* temp = head;

head = head->next;

free(temp);

}

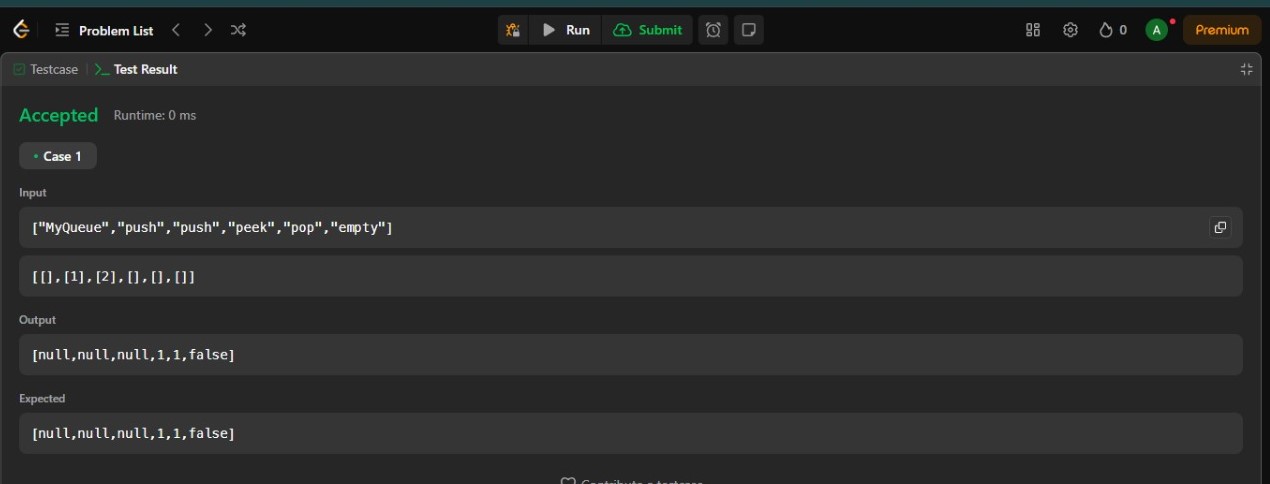
}

free(lists);

return 0;

}

1. **Output:**

****

**Fig.2. merge and sort**

1. **Learning Outcomes:**

 Efficient use of data structures like **priority queues** to handle sorting.

 Understanding the merging process of multiple sorted lists.

 Optimizing the solution to handle large inputs efficiently.

**Problem-5**

**1. Aim:** To implement an efficient algorithm for sorting a singly linked list in **ascending order**.

1. **Objective:**

 **Understand Linked List Sorting** – Learn how to sort a linked list using **Merge Sort** instead of array-based sorting algorithms.

 **Implement Divide and Conquer Approach** – Use recursion to break down the list and merge it efficiently.

 **Optimize Space Complexity** – Avoid extra memory allocation for sorting while maintaining efficiency.

1. **Code:**

class Solution {

public:

// Function to find the middle of the linked list

ListNode\* findMiddle(ListNode\* head) {

ListNode\* slow = head;

ListNode\* fast = head->next; // Start fast one step ahead to split properly

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

// Function to merge two sorted linked lists

ListNode\* merge(ListNode\* left, ListNode\* right) {

if (!left) return right;

if (!right) return left;

ListNode\* dummy = new ListNode(0);

ListNode\* current = dummy;

while (left && right) {

if (left->val <= right->val) {

current->next = left;

left = left->next;

} else {

current->next = right;

right = right->next;

}

current = current->next;

}

if (left) current->next = left;

if (right) current->next = right;

return dummy->next;

}

// Merge Sort for linked list

ListNode\* sortList(ListNode\* head) {

if (!head || !head->next) return head;

// Step 1: Split the list into two halves

ListNode\* mid = findMiddle(head);

ListNode\* rightHalf = mid->next;

mid->next = nullptr; // Break the list into two parts

// Step 2: Recursively sort both halves

ListNode\* leftSorted = sortList(head);

ListNode\* rightSorted = sortList(rightHalf);

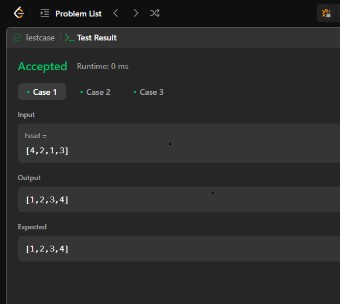
// Step 3: Merge the sorted halves

return merge(leftSorted, rightSorted);

}

};

1. **Output:**

****

**Fig.2. sorting**

1. **Learning Outcomes:**

* Gain hands-on experience with **Merge Sort on linked lists** and understand why it is the best approach for sorting them.
* Learn the **slow and fast pointer technique** to find the middle node of a linked list.
* Develop an understanding of **recursive sorting algorithms** and their impact on **time complexity**.
* Be able to implement **efficient merging of two sorted lists** using pointers, avoiding extra space usage.