Data Structures & Applications (CT-159) - Lab 02 Manual

Linked List Operations: A Lab Guide

1. Introduction & Lab Objective

This manual is designed to guide you through the practical material for **Singly Linked Lists**. The objective is to build a strong understanding of their core operations: insertion, deletion, traversal, and search.

By the end of this lab, you should be able to:

1. Implement a singly linked list from scratch.
2. Perform various operations on linked lists.
3. Apply linked lists to solve common programming problems.
4. Analyze the time complexity of different operations.

2. Core Concept Review

A **Linked List** is a linear data structure where elements, called **nodes**, are stored in sequence using pointers, not in contiguous memory locations like an array.

Key Components of a Singly Linked List:

- **Node:** The fundamental building block. It contains a data value and a next pointer/reference to the next node.
- **Head:** A special pointer that points to the first node of the list. If the head is nullptr, the list is empty.

Why Use a Linked List?

- **Dynamic Size:** It can grow and shrink at runtime without the need for contiguous memory blocks.
- **Efficient Insertions/Deletions:** Especially at the head of the list, which can be done in O(1) time, unlike arrays which require shifting elements in O(n) time.

Time Complexity Summary

Operation	Singly Linked List	Note
Access (by index)	O(n)	Must traverse from the head.
Search	O(n)	Must traverse from the

		head.
Insertion at Head	O(1)	Just update the head pointer.
Insertion at Tail	O(n)	Must traverse to the last node.
Deletion at Head	O(1)	Just update the head pointer.
Deletion at Tail	O(n)	Must traverse to the second-to-last node.

3. Starter Code

Use this code as the foundation for all your exercises. It provides the basic Node and LinkedList classes with the essential insert, deleteNode, search, and display methods.

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

// The Node class represents a single element in the linked list.
class Node {
public:
    int data;
    Node* next;

    // Constructor to create a new node.
    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

// The LinkedList class manages the nodes.
class LinkedList {
private:
    Node* head;

public:
    // Constructor to initialize an empty list.
    LinkedList() {
```

```
    head = nullptr;
}

// Insert a new node at the end of the list.
void insert(int data) {
    Node* newNode = new Node(data);
    if (head == nullptr) {
        head = newNode;
    } else {
        Node* temp = head;
        while (temp->next != nullptr) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

// Delete the first node with the given key.
void deleteNode(int key) {
    Node* temp = head;
    Node* prev = nullptr;

    // Case 1: Head node is the key.
    if (temp != nullptr && temp->data == key) {
        head = temp->next;
        delete temp;
        return;
    }

    // Case 2: Find the key to be deleted.
    while (temp != nullptr && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // Case 3: Key was not found.
    if (temp == nullptr) return;

    // Unlink the node and free memory.
    prev->next = temp->next;
    delete temp;
}
```

```
// Search for a value in the list.
bool search(int key) {
    Node* temp = head;
    while (temp != nullptr) {
        if (temp->data == key) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

// Display all elements of the list.
    void display() {
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }
};

int main() {
    LinkedList list;
    list.insert(10);
    list.insert(20);
    list.insert(30);
    list.insert(40);

cout << "Initial Linked List: ";
list.display();

int key = 20;
if (list.search(key)) {
    cout << key << " found in the list." << endl;
} else {
    cout << key << " not found in the list." << endl;
}

list.deleteNode(20);
cout << "After deleting 20: ";
list.display();
```

```
    return 0;
}
```

4. Lab Exercises: Breakdown & Guidance

Here is a detailed breakdown of each exercise. Try to solve them on your own first before referring to the hints.

Exercise 1: Merge Two Sorted Lists

Task: Given the heads of two sorted linked lists, list1 and list2, merge them into one sorted list.

Example:
Input: list1 = 1 -> 2 -> 4 -> NULL, list2 = 1 -> 3 -> 4 -> NULL
Output: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> NULL
Hints & Pseudocode:

- Create a **dummy node** to simplify handling the head of the new merged list.
- Use a tail pointer to build the new list, always appending to tail->next.
- Compare nodes from both lists and append the smaller one to the merged list.
- Remember to handle the remaining nodes after one list is exhausted.

```
function mergeTwoLists(list1, list2):
    dummy = new Node(0)
    tail = dummy

    while list1 is not null AND list2 is not null:
        if list1.data <= list2.data:
            tail.next = list1
            list1 = list1.next
        else:
            tail.next = list2
            list2 = list2.next
        tail = tail.next

if list1 is not null:
    tail.next = list1
else if list2 is not null:
    tail.next = list2

return dummy.next
```

Exercise 2: Remove Duplicates from Sorted List

Task: Delete all duplicates from a sorted linked list, leaving only distinct numbers.

Example:
Input: 1 -> 1 -> 2 -> 3 -> 3 -> NULL
Output: 1 -> 2 -> 3 -> NULL
Hints & Pseudocode:

- Since the list is sorted, all duplicates are adjacent.
- Traverse the list and compare the current node with the next node.
- If current->data is equal to current->next->data, bypass the duplicate node. Don't forget to free the memory of the deleted node.

```
function deleteDuplicates(head):
    current = head

    while current is not null AND current.next is not null:
        if current.data == current.next.data:
            duplicate = current.next
            current.next = current.next.next
            delete duplicate
        else:
            current = current.next

    return head
```

Exercise 3: Sort List (MergeSort)

Task: Sort a linked list in ascending order using MergeSort.

Hints & Pseudocode:

- **Base Case:** If the list is empty or has one node, return the head.
- **Split:** Find the middle of the list using the slow-fast pointer technique.
- **Divide:** Recursively sort the two halves.
- **Merge:** Use the mergeTwoLists logic from Exercise 1 to combine the two sorted halves.

```
function sortList(head):
    if head is null OR head.next is null:
        return head

    slow = head
    fast = head.next
    while fast is not null AND fast.next is not null:
        slow = slow.next
```

```
    fast = fast.next.next

rightHalf = slow.next
slow.next = nullptr
leftHalf = head

leftSorted = sortList(leftHalf)
rightSorted = sortList(rightHalf)

return mergeTwoLists(leftSorted, rightSorted)
```

Exercise 4: Check for Palindrome Linked List

Task: Return true if a linked list is a palindrome (reads the same forwards and backwards).

Example:
Input: 1 -> 2 -> 2 -> 1 -> NULL
Output: true
Hints & Pseudocode (Efficient O(n) **time,** O(1) **space):**

- **Find Middle:** Use the slow-fast pointer technique.
- **Reverse Second Half:** Reverse the linked list starting from the middle node.
- **Compare:** Traverse the first half and the reversed second half simultaneously, comparing their values.
- **Restore (Optional):** As a good practice, reverse the second half again to restore the original list structure.

```
function isPalindrome(head):
    if head is null: return true

    // 1. Find the middle
    slow = head
    fast = head
    while fast.next is not null AND fast.next.next is not null:
        slow = slow.next
        fast = fast.next.next

    // 2. Reverse the second half
    secondHalfStart = reverseList(slow.next)

    // 3. Compare for palindrome
    firstPointer = head
    secondPointer = secondHalfStart
    result = true
```

```
while secondPointer is not null:
    if firstPointer.data != secondPointer.data:
        result = false
        break
    firstPointer = firstPointer.next
    secondPointer = secondPointer.next

// 4. Restore the list
slow.next = reverseList(secondHalfStart)

return result
```

Exercise 5 & 6: Implement Circular Queue and Stack using Linked List

These are **Application Exercises**. You will use a linked list to implement another abstract data structure.

For Stack (LIFO - Last In, First Out):

- The head of your linked list will be the top of the stack.
- **Push:** Insert a new node at the **head** (O(1)).
- **Pop:** Remove and return the value from the **head** (O(1)).
- **Peek:** Return the value from the **head** without removing it (O(1)).

For Circular Queue (FIFO - First In, First Out):

- You will need **two pointers**: front and rear.
- **Enqueue:** Insert a new node at the **rear**. Update the rear pointer and link the new node to the front to make it circular.
- **Dequeue:** Remove the node from the **front**. Update the front pointer and ensure the rear's next pointer still points to the new front.

5. Grading Rubric (25 Marks)

- **R1 (5 marks):** Correctness and efficiency of the solution.
- **R2 (5 marks):** Code clarity, organization, and comments.
- **R3 (5 marks):** Proper memory management (no leaks).
- **R4 (5 marks):** Ability to test with multiple input cases.
- **R5 (5 marks):** Overall completion and submission quality.

Good luck!