Data Structure Algorithms & Applications (CT-159) Lab 02

Linked List

Objectives

The objective of the lab is to get students familiar with Linked List and its operation, that is, insertion and deletion.

Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Introduction

A **linked list** is a linear data structure in which elements (also called nodes) are stored in sequence but are not stored in contiguous memory locations. Instead, each element points to the next one in the sequence. This allows dynamic memory allocation, meaning that linked lists can easily grow or shrink during runtime.

Types of Linked Lists:

- **Singly Linked List**: Each node contains data and a pointer/reference to the next node.
- **Doubly Linked List**: Each node contains data, a pointer to the next node, and a pointer to the previous node.
- **Circular Linked List**: Similar to a singly or doubly linked list, but the last node points back to the first node, forming a loop.

Properties of Linked Lists:

- **Dynamic Size**: Linked lists do not have a fixed size, and their length can change dynamically as nodes are inserted or deleted.
- **Efficient Insertions and Deletions**: Adding or removing nodes from a linked list is efficient (especially at the head and tail) since no elements need to be shifted.
- **Non-contiguous Memory Allocation**: Nodes can be scattered in memory, unlike arrays, where elements are stored in contiguous blocks.
- **Pointers/References**: Each node contains at least one pointer that refers to another node, creating the link between nodes.
- **No Random Access**: Unlike arrays, linked lists do not allow direct access to an element by index. Traversal from the head is required to access specific nodes.

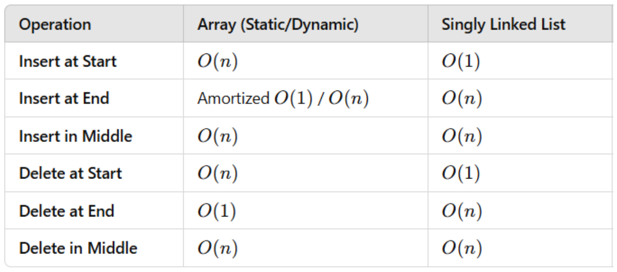
```
#include <iostream>
using namespace std;
class Node {   // Node class representing each element in the linked list
public:
   int data;  // Data of the node
   Node* next; // Pointer to the next node

   Node(int data) {    // Constructor to initialize the node with data and set next to nullptr
      this->data = data;
      this->next = nullptr;
   }
};
class LinkedList {              // LinkedList class representing the singly linked list
private:
   Node* head; // Pointer to the first node in the list
public:
   LinkedList() {                 // Constructor to initialize the linked list
      head = nullptr;
   }

   void insert(int data) {      // Insert a new node at the end of the list
      Node* newNode = new Node(data); // Create a new node
      if (head == nullptr) {
         head = newNode;  // If list is empty, set head to the new node
      } else {
         Node* temp = head;
         while (temp->next != nullptr) {
            temp = temp->next;  // Traverse to the last node
         }
         temp->next = newNode;  // Set the next of the last node to new node
      }
   }
```

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

        if (temp != nullptr && temp->data == key) {                  // If head node itself holds the key
            head = temp->next; // Move head to the next node
            delete temp;       // Free the old head
            return;
        }

        while (temp != nullptr && temp->data != key) {            // Search for the key to be deleted
            prev = temp;
            temp = temp->next;
        }
        if (temp == nullptr) return;    // If key was not present in the list
        prev->next = temp->next;     // Unlink the node from the linked list
        delete temp; // Free the memory of the node
    }

    bool search(int key) {   // Search for a node with a given value
        Node* temp = head;
        while (temp != nullptr) {
            if (temp->data == key) {
                return true;  // Key found
            }
            temp = temp->next;
        }
        return false;  // Key not found
    }

    void display() {                // Display the contents of the linked list
        Node* temp = head;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }
};
int main() {
    LinkedList list;

    list.insert(10); // Insert elements into the linked list
    list.insert(20);
    list.insert(30);
    list.insert(40);

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

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

    list.deleteNode(20); // Delete an element
    cout << "After deleting 20: ";
    list.display();
    return 0;
}
```

Computational Complexity:

- **Access Time**: O(n), since you need to traverse from the head to the desired node.
- **Search**: O(n), since traversal is required to search for a specific element.
- **Insertion**:
  At the Head: O(1), constant time since you just need to update the pointers.
  At the Tail: O(n), as traversal is required to reach the end.
  In the Middle: O(n), since traversal to the insertion point is required.
- **Deletion**:
  At the Head: O(1), constant time as it involves updating pointers.
  At the Tail: O(n), since traversal is needed to find the previous node.
  In the Middle: O(n), because you must first find the node to delete by traversal.

Summary of Complexity Comparison:



Operation	Array (Static/Dynamic)	Singly Linked List
Insert at Start	$O(n)$	$O(1)$
Insert at End	Amortized $O(1)$ / $O(n)$	$O(n)$
Insert in Middle	$O(n)$	$O(n)$
Delete at Start	$O(n)$	$O(1)$
Delete at End	$O(1)$	$O(n)$
Delete in Middle	$O(n)$	$O(n)$

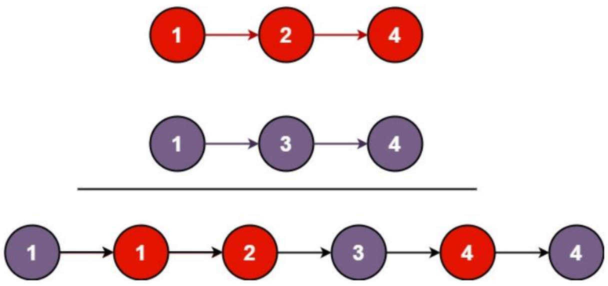
Key Use Cases:

- When dynamic memory allocation is needed.
- When frequent insertions and deletions are required, especially at the head or middle of the list.
- When the size of the dataset is unknown in advance.

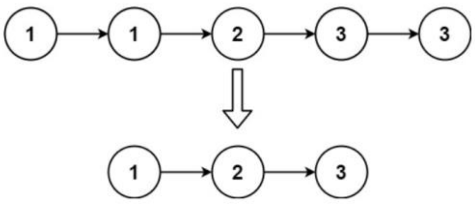
Linked lists are often used in scenarios like implementing stacks, queues, and adjacency lists in graphs.

Exercise

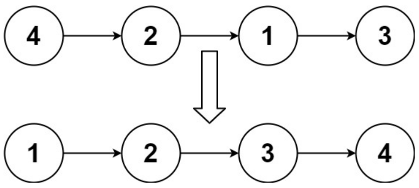
1. You are given the heads of two sorted linked lists list1 and list2. Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.
   Example: **Input:** list1 = [1,2,4], list2 = [1,3,4], **Output:** [1,1,2,3,4,4]



2. Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

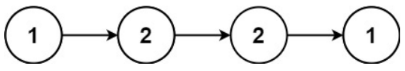


3. Given the head of a linked list, return the list after sorting it in ascending order. Use mergesort.



4. Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:



```
Input: head = [1,2,2,1]
Output: true
```

5. Implement class of a Circular Queue using a Linked List.
6. Implement class of a Stack using a Linked List.

Lab 06 Evaluation		
Student Name:	**Student ID:**	**Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		