Linked Lists

1 Overview

A **linked list** is a linear data structure where each element (called a *node*) points to the next, forming a chain. Each node contains:

- The actual data (here of type `int`)
- A pointer to the next node

The **head pointer** refers to the first node of the list. It is essential for accessing the list. If the head is `nullptr`, the list is empty.

1.1 Advantages over Arrays

- Dynamic size: no need to define capacity in advance.
- Efficient insertion/deletion: especially at the head or middle (no shifting required).

1.2 Disadvantages

- Random access is not possible; traversal is sequential.
- Extra memory is used for storing pointers.
- Poorer cache performance compared to arrays.

2 Visual Representation

A simple linked list storing 3, 7, and 10 would look like:

$$\texttt{head} \rightarrow \boxed{3 \mid *} \rightarrow \boxed{7 \mid *} \rightarrow \boxed{10 \mid \texttt{NULL}}$$

Each box represents a node storing an integer and a pointer to the next node.

3 Node Structure and Linked List Class

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

struct Node {
    int data;
    Node* next;
    Node(int val);
};

class LinkedList {
private:
```

```cpp
    Node* head;

public:
    LinkedList();
    ~LinkedList();

    void insertAtHead(int val);
    void insertAtEnd(int val);
    void deleteValue(int val);
    void print() const;
};

#endif

#include <iostream>
#include "LinkedList.h"

Node::Node(int val) : data(val), next(nullptr) {}

LinkedList::LinkedList() : head(nullptr) {}

LinkedList::~LinkedList() {
    Node* curr = head;
    while (curr) { // Until current node is null ptr
        Node* temp = curr;
        curr = curr->next;
        delete temp;
    }
}

void LinkedList::insertAtHead(int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    head = newNode;
}

void LinkedList::insertAtEnd(int val) {
    Node* newNode = new Node(val);
    if (!head) { // List empty
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) // Until next node is nullptr
        temp = temp->next;
    temp->next = newNode;
}

void LinkedList::deleteValue(int val) {
    // If the list is empty, nothing to delete
    if (!head) return;

    // Check if the head node is the one to delete
    if (head->data == val) {
        Node* temp = head;        // Save the current head
        head = head->next;        // Move head to the next node
        delete temp;              // Delete the old head
        return;
    }

    // Traverse the list to find the node before the one to delete
    Node* curr = head;
    while (curr->next && curr->next->data != val) {
        curr = curr->next;
    }

    // If the node was found, delete it
    if (curr->next) {
```

```
        Node* temp = curr->next;                // Node to delete
        curr->next = curr->next->next;          // Bypass the node
        delete temp;                            // Free memory
    }
}

void LinkedList::print() const {
    Node* temp = head;
    while (temp) {
        std::cout << temp->data << " -> ";
        temp = temp->next;
    }
    std::cout << "NULL\n";
}
```

3.1 Doubly Linked Lists

A **doubly linked list** allows traversal in both directions. Each node has:

- A data field
- A pointer to the next node
- A pointer to the previous node

```
struct DNode {
    int data;
    DNode* prev;
    DNode* next;
    DNode(int val) : data(val), prev(nullptr), next(nullptr) {}
};
```

3.2 Templated Linked List

A generic linked list using templates:

```
template <typename T>
struct Node {
    T data;
    Node* next;
    Node(T val) : data(val), next(nullptr) {}
};
```

This allows storing any data type: `int`, `double`, `std::string`, or user-defined objects, with the same logic as above reused across types.

4 Time Complexity (Big-O)

Operation	Time Complexity
Read	$O(n)$
Insert	$O(1)$
Delete	$O(1)$
Update	$O(n)$

- Read and Update are linear-time operations: $O(n)$
- Insert and Delete are constant-time if a pointer to the location is available (head/tail): $O(1)$