Module 2 - Linked List

Theoretical basics of Linked Lists, their types, benefits, and example code.

Part 1 - Understanding Linked List

Definition of Linked List

A Linked List is a linear data structure consisting of elements called nodes. Each node has two main parts:

- **Data** – stores the value of the element.
- **Pointer/Reference** – points to the next node (or the previous node in a Doubly Linked List).

Differences Between Linked List and Array

Aspect	Array	Linked List
Storage	Elements are stored contiguously in memory.	Nodes are stored in non-contiguous memory and linked by pointers.
Element Access	Direct access using an index, complexity O(1).	Access requires traversal from the start, complexity O(n).
Size	Static, size determined at declaration.	Dynamic, can grow or shrink as needed.
Insert/Delete	Expensive due to element shifting, complexity O(n).	More efficient, only pointer adjustments needed, complexity O(1) if position is known.

Benefits of Linked List

Dynamic Size

- No need to define size in advance like an array.
- Can grow or shrink according to program needs.

Efficiency in Insert/Delete Operations

- Adding or removing elements does not require shifting data.
- Complexity can be O(1) if the pointer is known.

More Flexible Memory Usage

- Nodes can be stored in scattered memory locations.
- Useful in systems with fragmented memory.

Support for Complex Data Structures

- Forms the basis for other data structures such as Stack, Queue, Deque, and Graph.
- Enables creation of variants like Circular Linked List and Doubly Linked List.

Two-Way Traversal (in Doubly Linked List)

- Makes navigation forward and backward easier.

Applications of Linked Lists

Dynamic Memory Allocation

- Used to keep track of free and allocated memory blocks.

Undo/Redo Operations in Text Editors

- Stores changes and navigates through editing history.

Graph Representation

- Efficiently implements adjacency lists in graph structures.

Implementation of Other Data Structures

- Serves as a base for stacks, queues, and deques.

Advantages of Linked List

- Efficient insertion and deletion since shifting of elements is not required as in arrays.
- Dynamic size allows growth or shrinkage at runtime.
- Memory utilization is more efficient; no pre-allocation reduces wasted space.
- Suitable for applications with large or frequently changing datasets.
- Uses non-contiguous memory blocks, useful in memory-limited systems.

Disadvantages of Linked List

- Direct access to an element is not possible; traversal from the start is required.
- Each node requires extra memory to store a pointer.
- More complex to implement and manage compared to arrays.
- Pointer mismanagement can cause bugs, memory leaks, or segmentation faults.

Short Code Example

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

struct Node {
    int data;
```

```
    Node* next;
};

void pushFront(Node*& head, int value) {
    Node* newNode = new Node(); // Buat node baru
    newNode->data = value;      // Isi data
    newNode->next = head;       // Hubungkan ke head lama
    head = newNode;             // Jadikan node baru sebagai head
}

void printList(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        cout << current->data << " -> ";
        current = current->next;
    }
    cout << "NULL\n";
}

int main() {
    Node* head = nullptr;  // List kosong

    pushFront(head, 10);
    pushFront(head, 20);
    pushFront(head, 30);

    printList(head); // Output: 30 -> 20 -> 10 -> NULL

    return 0;
}
```

Code Explanation:

- `Node` is the basic structure of a Linked List.
- `pushFront` adds a new node at the beginning of the list.
- `printList` traverses the list and displays all elements.

Part 2 - Types of Linked List

Types of Linked Lists

Based on the structure of linked lists, they can be classified into several types:

- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**

1. Singly Linked List

The singly linked list is the simplest form of linked list, where each node contains two members: **data** and a **next pointer** that stores the address of the next node. Each node in a singly linked list is connected through the next pointer, and the next pointer of the last node points to `NULL`, denoting the end of the list.

Singly Linked List Representation

A singly linked list can be represented as a pointer to the first node, where each node contains:

- **Data:** Actual information stored in the node.
- **Next:** Pointer to the next node.

```
// Structure to represent the singly linked list
struct Node {
    int data;       // Data field
    struct Node* next; // Pointer to the next node
};
```

2. Doubly Linked List

The doubly linked list is a modified version of the singly linked list. Each node contains three members: **data**, **next**, and **prev**. The `prev` pointer stores the address of the previous node. Nodes are connected via both `next` and `prev` pointers. The `prev` pointer of the first node and the `next` pointer of the last node point to `NULL`.

Doubly Linked List Representation

Each node contains:

- **Data:** Actual information stored.
- **Next:** Pointer to the next node.
- **Prev:** Pointer to the previous node.

```
// Structure to represent the doubly linked list
struct Node {
    int data;       // Data field
    struct Node* next; // Pointer to the next node
    struct Node* prev; // Pointer to the previous node
};
```

3. Circular Linked List

A circular linked list is similar to a singly linked list, except the next pointer of the last node points to the **first node** instead of `NULL`. This circular nature is useful in applications like media players.

Circular Linked List Representation

Each node contains:

- **Data:** Actual information stored.
- **Next:** Pointer to the next node; the last node points to the first node.

```
// Structure to represent the circular linked list
struct Node {
    int data;       // Data field
    struct Node* next; // Pointer to the next node
};
```

Part 3 - Searching

1. Searching in a Custom Singly Linked List

```
#include <bits/stdc++.h>
using namespace std;

// Linked list node
class Node
{
public:
    int key;
    Node* next;
};

// Add a new node at the front
void push(Node** head_ref, int new_key)
{
    Node* new_node = new Node();
    new_node->key = new_key;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Search for a value in the linked list
bool search(Node* head, int x)
{
    Node* current = head;
    while (current != NULL)
    {
        if (current->key == x)
            return true;
        current = current->next;
    }
    return false;
}

int main()
```

```
{
    Node* head = NULL;
    int x = 21;

    // Construct list: 14->21->11->30->10
    push(&head, 10);
    push(&head, 30);
    push(&head, 11);
    push(&head, 21);
    push(&head, 14);

    search(head, 21) ? cout << "Yes" : cout << "No";
    return 0;
}
```

Explanation:

- `Node` class represents each node in the linked list.
- `push` adds a new node at the beginning.
- `search` traverses the list iteratively to find the key.

2. Searching in STL `std::list`

C++ Standard Template Library (STL) provides the `list` container which can be used to implement a linked list. Searching can be done using the `std::find` algorithm.

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

int main() {
    list<int> l = {14, 21, 11, 30, 10};
    int x = 21;

    auto it = find(l.begin(), l.end(), x);

    if (it != l.end())
        cout << "Yes";
    else
        cout << "No";
```

```
    return 0;
}
```

Explanation:

- `std::list` is a doubly linked list implementation.
- `std::find` iterates through the list to locate the element.
- Returns an iterator to the element if found, or `l.end()` if not found.

Part 4 - Manual VS STL List

A **doubly linked list** is a data structure where each node contains a pointer to the **next** and **previous** nodes, allowing traversal in both directions. In C++, you can implement it manually or use the built-in STL `std::list`.

Differences between Manual Doubly Linked List and STL `std::list`:

Feature	Manual Doubly Linked List	STL `std::list`
Implementation	You define the `Node` structure and pointers manually.	Already implemented as a doubly linked list in the STL.
Memory Management	Manual allocation and deallocation using `new` and `delete`.	Automatic memory management.
Operations	Insert, delete, traversal, and search must be implemented manually.	Provides built-in functions like `push_back`, `push_front`, `erase`, and `find`.
Complexity	More control but more prone to errors like memory leaks.	Safer and easier to use, less prone to bugs.

1. Manual Doubly Linked List Example

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

struct Node {
    int data;
    Node* next;
    Node* prev;
};

// Add a node at the end
void pushBack(Node*& head, int value) {
    Node* newNode = new Node{value, nullptr, nullptr};
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) temp = temp->next;
```

```
    temp->next = newNode;
    newNode->prev = temp;
}

// Print the list
void printList(Node* head) {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " ";
        temp = temp->next;
    }
    cout << endl;
}

int main() {
    Node* head = nullptr;
    pushBack(head, 10);
    pushBack(head, 20);
    pushBack(head, 30);

    printList(head); // Output: 10 20 30
}
```

2. STL `std::list` Example

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

int main() {
    list<int> l;

    l.push_back(10);
    l.push_back(20);
    l.push_back(30);

    for (int x : l)
        cout << x << " "; // Output: 10 20 30
    cout << endl;
}
```

Explanation:

- Manual doubly linked list requires defining nodes and managing pointers yourself.
- `std::list` simplifies everything: memory management and operations like insertion, deletion, and traversal are built-in.