Lab Experiment No. 2

Objective

To implement and analyze Linked Lists and their operations.

Tasks:

T-2.1	**Singly Linked List**
2.1.1	Insertion a new node
2.1.2	Deletion of a node
2.1.3	Traversing
T-2.2	**Doubly Linked List**
2.2.1	Insertion a new node
2.2.2	Deletion of a node
2.2.3	Traversing
T-2.3	**Circular Linked List**
2.3.1	Insertion a new node
2.3.2	Deletion of a node
2.3.3	Traversing

Theory

A **linked list** is a linear data structure where each element (called a **node**) contains two parts:

1. **Data**: The value stored in the node.
2. **Pointer**: A reference to the next (or previous) node in the sequence.

Unlike arrays, linked lists do not require contiguous memory allocation, making them dynamic and flexible in size.

Types of Linked Lists

1. **Singly Linked List**:
   - Each node points only to the next node.
   - The last node points to NULL.
2. **Doubly Linked List**:
   - Each node has two pointers: one to the next node and one to the previous node.
   - The first node's prev pointer and the last node's next pointer point to NULL.
3. **Circular Linked List**:
   - Similar to a singly linked list, but the last node points back to the first node, forming a circle.

Algorithm & Implementation

T-2.1: Singly Linked List

2.1.1 Insertion of a New Node

- **Algorithm**:
    - Create a new node.
    - Assign data to the new node.
    - Point the new node's next to the current head.
    - Update the head to point to the new node.
- **Time Complexity**:
    - **Best Case**: O(1) (insertion at the head).
    - **Worst Case**: O(n) (insertion at the end).

2.1.2 Deletion of a Node

- **Algorithm**:
    - Traverse the list to find the node to delete.
    - Update the next pointer of the previous node to skip the node to be deleted.
    - Free the memory of the deleted node.
- **Time Complexity**:
    - **Best Case**: O(1) (deletion at the head).
    - **Worst Case**: O(n) (deletion at the end).

2.1.3 Traversing

- **Algorithm**:
    1. Start from the head.
    2. Move to the next node until NULL is reached.
- **Time Complexity**: O(n).

T-2.2: Doubly Linked List

2.2.1 Insertion of a New Node

- **Algorithm**:
    - Create a new node.
    - Assign data to the new node.
    - Point the new node's next to the current head and prev to NULL.
    - Update the prev of the current head to point to the new node.
    - Update the head to point to the new node.
- **Time Complexity**:
    - **Best Case**: O(1) (insertion at the head).
    - **Worst Case**: O(n) (insertion at the end).

2.2.2 Deletion of a Node

- **Algorithm**:
    - Traverse the list to find the node to delete.
    - Update the next pointer of the previous node and the prev pointer of the next node.
    - Free the memory of the deleted node.
- **Time Complexity**:
    - **Best Case**: O(1) (deletion at the head).
    - **Worst Case**: O(n) (deletion at the end).

2.2.3 Traversing

- **Algorithm**:
    1. Start from the head.
    2. Move to the next node until NULL is reached.
- **Time Complexity**: O(n).

T-2.3: Circular Linked List

2.3.1 Insertion of a New Node

- **Algorithm**:
    - Create a new node.
    - Assign data to the new node.
    - Point the new node's next to the current head.
    - Update the next pointer of the last node to point to the new node.
    - Update the head to point to the new node (if inserting at the head).
- **Time Complexity**:
    - **Best Case**: O(1) (insertion at the head).
    - **Worst Case**: O(n) (insertion at the end).

2.3.2 Deletion of a Node

- **Algorithm**:
    - Traverse the list to find the node to delete.
    - Update the next pointer of the previous node to skip the node to be deleted.
    - Free the memory of the deleted node.
- **Time Complexity**:
    - **Best Case**: O(1) (deletion at the head).
    - **Worst Case**: O(n) (deletion at the end).

2.3.3 Traversing

- **Algorithm**:
    1. Start from the head.

2. Move to the next node until the head is reached again.

- **Time Complexity**: O(n).

Code

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

// Node structure for Singly Linked List
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

// Singly Linked List Class
class SinglyLinkedList {
public:
    Node* head;
    SinglyLinkedList() : head(nullptr) {}

    void insert(int val) {
        Node* newNode = new Node(val);
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
    }

    void remove(int val) {
        if (!head) return;
        if (head->data == val) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }
        Node* temp = head;
        while (temp->next && temp->next->data !=
val) temp = temp->next;
        if (temp->next) {
            Node* delNode = temp->next;
            temp->next = temp->next->next;
```

```
    void traverse() {
        DNode* temp = head;
        while (temp) {
            cout << temp->data << " <-> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};

// Node structure for Circular Linked List
struct CNode {
    int data;
    CNode* next;
    CNode(int val) : data(val), next(nullptr) {}
};

// Circular Linked List Class
class CircularLinkedList {
public:
    CNode* head;
    CircularLinkedList() : head(nullptr) {}

    void insert(int val) {
        CNode* newNode = new CNode(val);
        if (!head) {
            head = newNode;
            head->next = head;
            return;
        }
        CNode* temp = head;
        while (temp->next != head) temp =
temp->next;
        temp->next = newNode;
        newNode->next = head;
    }

    void remove(int val) {
        if (!head) return;
        CNode* temp = head, *prev = nullptr;
```

```
            delete delNode;
        }
    }

    void traverse() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};

// Node structure for Doubly Linked List
struct DNode {
    int data;
    DNode* next;
    DNode* prev;
    DNode(int val) : data(val), next(nullptr),
prev(nullptr) {}
};

// Doubly Linked List Class
class DoublyLinkedList {
public:
    DNode* head;
    DoublyLinkedList() : head(nullptr) {}

    void insert(int val) {
        DNode* newNode = new DNode(val);
        if (!head) {
            head = newNode;
            return;
        }
        DNode* temp = head;
        while (temp->next) temp = temp->next;
        temp->next = newNode;
        newNode->prev = temp;
    }

    void remove(int val) {
        if (!head) return;
        DNode* temp = head;
        while (temp && temp->data != val) temp =
temp->next;
        if (!temp) return;
```

```
        if (head->data == val && head->next ==
head) {
            delete head;
            head = nullptr;
            return;
        }
        while (temp->next != head && temp->data !=
val) {
            prev = temp;
            temp = temp->next;
        }
        if (temp->data == val) {
            if (temp == head) {
                prev = head;
                while (prev->next != head) prev =
prev->next;
                head = head->next;
                prev->next = head;
            } else {
                prev->next = temp->next;
            }
            delete temp;
        }
    }

    void traverse() {
        if (!head) return;
        CNode* temp = head;
        do {
            cout << temp->data << " -> ";
            temp = temp->next;
        } while (temp != head);
        cout << "(HEAD)\n";
    }
};

// Main function for testing
int main() {
    cout << "Singly Linked List:\n";
    SinglyLinkedList sll;
    sll.insert(1); sll.insert(2); sll.insert(3);
    sll.traverse();
    sll.remove(2);
    sll.traverse();

    cout << "\nDoubly Linked List:\n";
    DoublyLinkedList dll;
    dll.insert(10); dll.insert(20); dll.insert(30);
```

```
        if (temp->prev) temp->prev->next =
temp->next;
        if (temp->next) temp->next->prev =
temp->prev;
        if (temp == head) head = temp->next;
        delete temp;
    }
```

```
    dll.traverse();
    dll.remove(20);
    dll.traverse();

    cout << "\nCircular Linked List:\n";
    CircularLinkedList cll;
    cll.insert(100); cll.insert(200); cll.insert(300);
    cll.traverse();
    cll.remove(200);
    cll.traverse();

    return 0;
}
```

Sample Output

```
Singly Linked List:
1 -> 2 -> 3 -> NULL
1 -> 3 -> NULL

Doubly Linked List:
10 <-> 20 <-> 30 <-> NULL
10 <-> 30 <-> NULL

Circular Linked List:
100 -> 200 -> 300 -> (HEAD)
100 -> 300 -> (HEAD)
```

Complexity Analysis

Operation	**Singly Linked List**	**Doubly Linked List**	**Circular Linked List**
Insertion at Head	O(1)	O(1)	O(1)
Insertion at Tail	O(n)	O(n)	O(n)
Insertion at Middle	O(n)	O(n)	O(n)
Deletion at Head	O(1)	O(1)	O(1)
Deletion at Tail	O(n)	O(n)	O(n)
Deletion at Middle	O(n)	O(n)	O(n)

Traversal	O(n)	O(n)	O(n)
Searching	O(n)	O(n)	O(n)

Conclusion

In this lab, we implemented and analyzed the operations of singly, doubly, and circular linked lists. We observed that:

- Insertion and deletion at the head are efficient (O(1)) for all types of linked lists.
- Traversal always takes O(n) time.
- Doubly linked lists provide additional flexibility with backward traversal but require more memory due to the extra prev pointer.

Linked lists are ideal for dynamic data storage where frequent insertions and deletions are required.