Singly circular Linked list

📘 Circular Linked List (CLL) – Complete Guide

🔷 1. Introduction

A **Circular Linked List (CLL)** is a variation of a linked list in which **the last node points back to the first node** instead of pointing to `NULL`.

Thus, the list forms a **circle**.

🧠 Key Properties:

- There is **no NULL** at the end.
- Traversal can start from **any node**.
- Useful for applications like **round-robin scheduling**, **music playlists**, and **memory buffers**.

🔷 2. Types of Circular Linked Lists

1. **Singly Circular Linked List** – Each node has a pointer to the next node, and the last node points to the first node.
2. **Doubly Circular Linked List** – Each node has pointers to both previous and next nodes, forming a closed loop in both directions.

🔷 3. Node Structure

👉 In C

```
struct Node {
    int data;
```

```
    struct Node* next;
};
```

👉 In C++

```
class Node {
public:
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
};
```

🔷 4. Creating a Circular Linked List

👉 In C

```
#include <stdio.h>
#include <stdlib.h>

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

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = newNode; // Point to itself (important)
```

```
    return newNode;
}
```

👉 In C++

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

class Node {
public:
    int data;
    Node* next;
    Node(int val) {
        data = val;
        next = this; // Point to itself
    }
};
```

🔷 5. Traversal of Circular Linked List

🔶 Algorithm

1. Start from the `head` node.
2. Print data.
3. Move to the next node.
4. Stop when you reach back to `head`.

👉 C Code

```
void display(struct Node* head) {
    if (head == NULL) return;
    struct Node* temp = head;
    do {
```

```
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
}
```

👉 C++ Code

```
void display(Node* head) {
    if (head == NULL) return;
    Node* temp = head;
    do {
        cout << temp->data << " ";
        temp = temp->next;
    } while (temp != head);
}
```

🔷 6. Insertion Operations

6.1 Insert at Beginning

🔶 Algorithm

1. Create a new node.
2. Traverse to the last node.
3. Point new node's next to head.
4. Point last node's next to new node.
5. Update head to new node.

👉 C Code

```
void insertAtBegin(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
```

```
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp→next != *head)
        temp = temp→next;
    temp→next = newNode;
    newNode→next = *head;
    *head = newNode;
}
```

👉 C++ Code

```
void insertAtBegin(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp→next != head)
        temp = temp→next;
    temp→next = newNode;
    newNode→next = head;
    head = newNode;
}
```

6.2 Insert at End

🔶 Algorithm

1. Create a new node.
2. Traverse to the last node.
3. Point last node's next to new node.

4. Point new node’s next to head.

👉 C Code

```
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != *head)
        temp = temp->next;
    temp->next = newNode;
    newNode->next = *head;
}
```

👉 C++ Code

```
void insertAtEnd(Node*& head, int data) {
    Node* newNode = new Node(data);
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != head)
        temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}
```

6.3 Insert at Given Position

🔶 Algorithm

1. If position is 1 → call insertAtBegin.
2. Else traverse to (pos–1)th node.
3. Insert new node between (pos–1)th and pos-th nodes.

👉 C Code

```
void insertAtPos(struct Node** head, int data, int pos) {
    struct Node* newNode = createNode(data);
    if (pos == 1) {
        insertAtBegin(head, data);
        return;
    }
    struct Node* temp = *head;
    for (int i = 1; i < pos - 1 && temp->next != *head; i++)
        temp = temp->next;
    newNode->next = temp->next;
    temp->next = newNode;
}
```

👉 C++ Code

```
void insertAtPos(Node*& head, int data, int pos) {
    if (pos == 1) {
        insertAtBegin(head, data);
        return;
    }
    Node* newNode = new Node(data);
    Node* temp = head;
    for (int i = 1; i < pos - 1 && temp->next != head; i++)
        temp = temp->next;
    newNode->next = temp->next;
```

```
    temp→next = newNode;
}
```

🔷 7. Deletion Operations

7.1 Delete from Beginning

🔶 Algorithm

1. If list empty → return.
2. Traverse to the last node.
3. Point last node's next to head→next.
4. Delete head and update head pointer.

👉 C Code

```
void deleteBegin(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    struct Node* last = *head;
    while (last→next != *head)
        last = last→next;
    if (temp == last) {
        free(*head);
        *head = NULL;
        return;
    }
    last→next = temp→next;
    *head = temp→next;
    free(temp);
}
```

👉 C++ Code

```
void deleteBegin(Node*& head) {
    if (head == NULL) return;
    Node* temp = head;
    Node* last = head;
    while (last→next != head)
        last = last→next;
    if (temp == last) {
        delete head;
        head = NULL;
        return;
    }
    last→next = head→next;
    head = head→next;
    delete temp;
}
```

7.2 Delete from End

🔶 Algorithm

1. If list empty → return.
2. Traverse till the second last node.
3. Update its next to head.
4. Delete the last node.

👉 C Code

```
void deleteEnd(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    struct Node* prev = NULL;
    while (temp→next != *head) {
        prev = temp;
        temp = temp→next;
```

```
    }
    if (temp == *head) {
        free(*head);
        *head = NULL;
        return;
    }
    prev→next = *head;
    free(temp);
}
```

👉 C++ Code

```
void deleteEnd(Node*& head) {
    if (head == NULL) return;
    Node* temp = head;
    Node* prev = NULL;
    while (temp→next != head) {
        prev = temp;
        temp = temp→next;
    }
    if (temp == head) {
        delete head;
        head = NULL;
        return;
    }
    prev→next = head;
    delete temp;
}
```

7.3 Delete at Given Position

🔶 Algorithm

1. If position = 1 → call deleteBegin.

2. Traverse till (pos–1)th node.
3. Skip pos-th node and adjust links.
4. Free deleted node.

👉 C Code

```
void deleteAtPos(struct Node** head, int pos) {
    if (*head == NULL) return;
    if (pos == 1) {
        deleteBegin(head);
        return;
    }
    struct Node* temp = *head;
    struct Node* prev = NULL;
    for (int i = 1; i < pos && temp->next != *head; i++) {
        prev = temp;
        temp = temp->next;
    }
    prev->next = temp->next;
    free(temp);
}
```

👉 C++ Code

```
void deleteAtPos(Node*& head, int pos) {
    if (head == NULL) return;
    if (pos == 1) {
        deleteBegin(head);
        return;
    }
    Node* temp = head;
    Node* prev = NULL;
    for (int i = 1; i < pos && temp->next != head; i++) {
        prev = temp;
```

```
        temp = temp→next;
    }
    prev→next = temp→next;
    delete temp;
}
```

🔷 8. Time Complexity Analysis

Operation	Best Case	Average Case	Worst Case	Explanation
Traversal	O(1)	O(n)	O(n)	Traverses n nodes
Insert at Beginning	O(1)	O(n)	O(n)	Need to find last node
Insert at End	O(n)	O(n)	O(n)	Traverse to last node
Insert at Position	O(n)	O(n)	O(n)	Traverse up to position
Delete from Beginning	O(n)	O(n)	O(n)	Traverse to last node
Delete from End	O(n)	O(n)	O(n)	Traverse to second last node
Delete at Position	O(n)	O(n)	O(n)	Traverse up to position

✅ **Space Complexity:** O(1) (no extra space apart from node pointers)

🔷 9. Advantages of Circular Linked List

- Efficient **round traversal** (no need to check for `NULL`).
- Can **access all nodes from any point**.
- Useful in **circular queues**, **buffer management**, and **task scheduling**.

🔷 10. Disadvantages

- Slightly more complex to implement.
- Risk of **infinite loops** if pointers are not handled properly.

🔷 11. Summary

Feature	Singly Circular Linked List
Last node points to	First node
Traversal direction	Only forward
NULL pointer	Not present
Memory usage	Less
Typical Use	Round-robin tasks, queues
