Doubly Circular LInked List

📘 Doubly Circular Linked List (DCLL) – Complete Guide

🔷 1. Introduction

A **Doubly Circular Linked List (DCLL)** is an advanced version of the doubly linked list in which:

- The **last node's next pointer** points to the **first node**.
- The **first node's prev pointer** points to the **last node**.

Thus, it forms a **circular structure in both directions**.

🔷 2. Characteristics

- Each node has **two pointers**: `next` and `prev`.
- You can **traverse in both directions**.
- There is **no NULL pointer** at either end.
- Commonly used in applications like **navigation menus**, **undo/redo systems**, and **multiplayer games**.

🔷 3. Node Structure

👉 In C

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

```
    struct Node* prev;
};
```

👉 In C++

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

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

🔷 4. Creating a Doubly Circular Linked List

👉 In C

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

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

struct Node* createNode(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = val;
    newNode->next = newNode;
```

```
    newNode->prev = newNode;
    return newNode;
}
```

👉 In C++

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

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

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

🔷 5. Traversal of DCLL

🔶 Algorithm (Forward Traversal)

1. Start from `head`.
2. Print data.
3. Move to `next` until reaching `head` again.

🔶 Algorithm (Backward Traversal)

1. Start from `head->prev` (last node).
2. Print data.
3. Move to `prev` until reaching `head->prev` again.

👉 C Code

```
void displayForward(struct Node* head) {
    if (head == NULL) return;
    struct Node* temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);
}

void displayBackward(struct Node* head) {
    if (head == NULL) return;
    struct Node* temp = head->prev;
    do {
        printf("%d ", temp->data);
        temp = temp->prev;
    } while (temp != head->prev);
}
```

👉 C++ Code

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

void displayBackward(Node* head) {
    if (head == NULL) return;
    Node* temp = head->prev;
    do {
```

```
        cout << temp→data << " ";
        temp = temp→prev;
    } while (temp != head→prev);
}
```

🔷 6. Insertion Operations

6.1 Insert at Beginning

🔶 Algorithm

1. Create new node.
2. If list empty → new node points to itself.
3. Otherwise:
    - Set `newNode→next = head`
    - Set `newNode→prev = head→prev`
    - Update `head→prev→next = newNode`
    - Update `head→prev = newNode`
    - Update `head = newNode`

👉 C Code

```
void insertAtBegin(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* last = (*head)→prev;
    newNode→next = *head;
    newNode→prev = last;
    last→next = newNode;
    (*head)→prev = newNode;
```

```
    *head = newNode;
}
```

👉 C++ Code

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

6.2 Insert at End

🔶 Algorithm

1. Create a new node.
2. If list empty → new node points to itself.
3. Otherwise:
    - Let `last = head→prev`
    - Update `newNode→next = head`
    - Update `newNode→prev = last`
    - Update `last→next = newNode`
    - Update `head→prev = newNode`

👉 C Code

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

👉 C++ Code

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

6.3 Insert at Given Position

🔶 Algorithm

1. If position = 1 → call `insertAtBegin`.
2. Traverse till (pos–1)th node.

3. Insert new node between nodes.

👉 C Code

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

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

```
    temp→next = newNode;
}
```

🔷 7. Deletion Operations

7.1 Delete from Beginning

🔶 Algorithm

1. If list empty → return.
2. If only one node → delete it.
3. Else:
    - Let `last = head→prev`
    - Update `head = head→next`
    - Update `head→prev = last`
    - Update `last→next = head`
    - Delete old head.

👉 C Code

```
void deleteBegin(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    struct Node* last = (*head)→prev;

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

```
    free(temp);
}
```

👉 C++ Code

```
void deleteBegin(Node*& head) {
    if (head == NULL) return;
    Node* temp = head;
    Node* last = head->prev;

    if (temp->next == head) {
        delete temp;
        head = NULL;
        return;
    }
    head = temp->next;
    head->prev = last;
    last->next = head;
    delete temp;
}
```

7.2 Delete from End

🔶 Algorithm

1. If list empty → return.
2. If only one node → delete it.
3. Else:
    - Let `last = head->prev`
    - Update `secondLast = last->prev`
    - Update `secondLast->next = head`
    - Update `head->prev = secondLast`
    - Delete last.

👉 C Code

```
void deleteEnd(struct Node** head) {
    if (*head == NULL) return;
    struct Node* last = (*head)→prev;

    if (last == *head) {
        free(last);
        *head = NULL;
        return;
    }
    struct Node* secondLast = last→prev;
    secondLast→next = *head;
    (*head)→prev = secondLast;
    free(last);
}
```

👉 C++ Code

```
void deleteEnd(Node*& head) {
    if (head == NULL) return;
    Node* last = head→prev;

    if (last == head) {
        delete head;
        head = NULL;
        return;
    }
    Node* secondLast = last→prev;
    secondLast→next = head;
    head→prev = secondLast;
    delete last;
}
```

7.3 Delete at Given Position

🔶 Algorithm

1. If position = 1 → call `deleteBegin`.
2. Traverse to (pos)th node.
3. Adjust previous and next links.
4. Delete target node.

👉 C Code

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

    temp->prev->next = temp->next;
    temp->next->prev = temp->prev;
    free(temp);
}
```

👉 C++ Code

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

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

🔷 8. Time Complexity Analysis

Operation	Best Case	Average Case	Worst Case	Explanation
Traversal (Forward/Backward)	O(1)	O(n)	O(n)	Must visit all nodes
Insert at Beginning	O(1)	O(1)	O(1)	Constant-time update
Insert at End	O(1)	O(1)	O(1)	Direct access via `prev` pointer
Insert at Position	O(n)	O(n)	O(n)	Traverse to position
Delete from Beginning	O(1)	O(1)	O(1)	Direct head removal
Delete from End	O(1)	O(1)	O(1)	Access last node directly
Delete at Position	O(n)	O(n)	O(n)	Traverse up to position

✅ **Space Complexity:** O(1)

🔷 9. Advantages of DCLL

- **Bidirectional traversal** (forward & backward).
- **No NULL checks** needed at ends.
- **Easy insertion/deletion** at both ends.
- Useful for **real-time circular buffers**, **playlist systems**, etc.

🔷 10. Disadvantages

- Slightly **higher memory usage** (two pointers per node).
- **More complex** insertion/deletion logic.
- Requires careful pointer management to avoid infinite loops.

🔷 11. Summary

Feature	Doubly Circular Linked List
Links	Both next & prev
Last node points to	First node
First node’s prev points to	Last node
Traversal	Both directions
End Condition	None (circular)
Applications	Music playlist, buffer management, OS scheduling

---