Introduction

A circular doubly linked list is a type of **linked list** where each node is connected to both its previous and next nodes, and the last node links back to the first node. This structure allows for efficient bidirectional traversal and looping through the list. In this guide, we will learn the basics, operations, and applications of circular doubly linked lists.

What is Circular Doubly Linked List?

A circular doubly linked list is a special kind of list used in **DSA** to store data. In this list, each piece of data is stored in a node. Every node is connected to two other nodes: one before it and one after it. This makes it easy to move forwards and backwards through the list.

What's unique about a circular doubly linked list is that the last node connects back to the first node, forming a circle. So, if you start at any node and keep moving forwards or backwards, you will eventually come back to the starting node.

This circular connection makes it different from regular linked lists, where the last node simply points to nothing (or "null").

Example:

Imagine a circular doubly linked list like a Ferris wheel. Each node is like a seat on the Ferris wheel. You can move to the seat next to you (forward) or the seat behind you (backward). And when you reach the last seat, instead of stopping, you move to the first seat again, just like how the Ferris wheel goes round and round.

This structure is useful in many computer applications where you need to cycle through data repeatedly, such as in task scheduling or managing playlists in a media player.

Structure of Circular Doubly Linked List

A circular doubly linked list is made up of nodes, where each node has three main components: data, a next pointer, and a previous pointer.

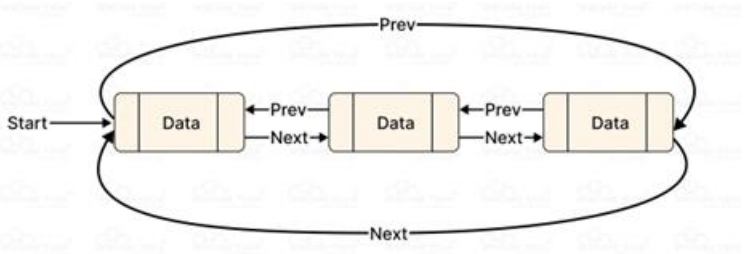
Nodes and Their Components

- **Data:** This is the actual information or value stored in the node.
- **Next Pointer:** This pointer points to the next node in the list.
- **Previous Pointer:** This pointer points to the previous node in the list.

Circular Nature

- In a circular doubly linked list, the next pointer of the last node points to the first node, creating a circular loop.
- Similarly, the previous pointer of the first node points to the last node, allowing bidirectional traversal in a circular manner.

Visual Representation



Circular Doubly Linked List in Java

```
class Node {
   int data;
   Node next;
   Node prev;

   public Node(int data) {
      this.data = data;
      next=null;
       prev=null;
   }
}

class CircularDoublyLinkedList {
   private Node head = null;
```

```
public void append(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
        newNode.next = newNode;
        newNode.prev = newNode;
    } else {
        Node tail = head.prev;
        tail.next = newNode;
        newNode.prev = tail;
        newNode.next = head;
        head.prev = newNode;
    }
}

public void display() {
    if (head == null) return;
    Node temp = head;
    do {
        System.out.print(temp.data + " ");
        temp = temp.next;
    } while (temp != head);
    System.out.println();
}

public static void main(String[] args) {
    CircularDoublyLinkedList cdll = new CircularDoublyLinkedList();
    cdll.append(1);
    cdll.append(2);
    cdll.append(3);
```

```
        cdll.display();  // Output: 1 2 3
    }
}
```

Basic Operations on Circular Doubly Linked List

1. Insertion

Inserting at the Beginning:

- Create a new node.
- Adjust the next pointer of the new node to point to the head.
- Adjust the previous pointer of the head to point to the new node.
- Set the new node as the new head.
- Update the previous pointer of the new head to point to the last node, and the next pointer of the last node to point to the new head.

Inserting at the End:

- Create a new node.
- Adjust the previous pointer of the new node to point to the last node.
- Adjust the next pointer of the last node to point to the new node.
- Set the next pointer of the new node to point to the head.
- Update the previous pointer of the head to point to the new node.

Inserting at a Given Position:

- Traverse the list to find the position.
- Adjust the pointers of the new node and the nodes at the position to insert the new node.

2. Deletion

Deleting from the Beginning:

- Adjust the next pointer of the last node to point to the second node.
- Set the second node as the new head.
- Adjust the previous pointer of the new head to point to the last node.

Deleting from the End:

- Adjust the next pointer of the second-to-last node to point to the head.
- Adjust the previous pointer of the head to point to the second-to-last node.

Deleting a Specific Node:

- Traverse the list to find the node.

- Adjust the pointers of the previous and next nodes to bypass the node to be deleted.

3. Traversal

Forward Traversal:

- Start from the head and move forward using the next pointer.
- Continue until you reach the head again.

Backward Traversal:

- Start from the head and move backward using the previous pointer.
- Continue until you reach the head again.

4. Searching

- Traverse the list using either the next or previous pointers.
- Compare each node's data with the target value until the value is found or the list is fully traversed.