**📘 Circular Linked List (CLL)**

**🔹 Definition:**

A **Circular Linked List** is a linear data structure where the **last node points back to the first node**, forming a **circular loop**.

It can be of two types:

1. **Singly Circular Linked List** – each node has a next pointer only
2. **Doubly Circular Linked List** – each node has both prev and next pointers

**🔹 Structure (Singly Circular)**

[Data | Next] → [Data | Next] → [Data | Next] → (points back to head)

**🔹 Key Characteristics:**

* No None in the next of the last node.
* Can start from any node and traverse the entire list.
* The traversal continues until the current node reaches the **head again**.

**🔹 Operations:**

**✅ 1. Traversal**

* Traverse until you reach the starting node again.

**✅ 2. Insertion**

* At beginning
* At end

**✅ 3. Deletion**

* From beginning
* From end

**🔹 Python Code Example (Singly Circular)**

class CNode:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

class CircularLinkedList:

def \_\_init\_\_(self):

self.head = None

# Append at the end

def append(self, data):

new\_node = CNode(data)

if not self.head:

self.head = new\_node

new\_node.next = self.head # point to itself

return

curr = self.head

while curr.next != self.head:

curr = curr.next

curr.next = new\_node

new\_node.next = self.head # link back to head

# Display the circular list

def display(self):

if not self.head:

print("List is empty")

return

curr = self.head

while True:

print(curr.data, end=" → ")

curr = curr.next

if curr == self.head:

break

print("(back to head)")

**🔹 Example Usage**

cll = CircularLinkedList()

cll.append(1)

cll.append(2)

cll.append(3)

cll.display()

**🔹 Output**

1 → 2 → 3 → (back to head)

**🔹 Advantages:**

* Circular nature makes it useful in applications where the list needs to be **cycled repeatedly** (e.g., round-robin scheduling)
* No need to reset pointer to head after reaching the end

**🔹 Disadvantages:**

* Slightly more complex to implement than singly/doubly linked lists
* Care is needed to avoid infinite loops during traversal

**📘 Circular Doubly Linked List (CDLL)**

**🔹 Definition:**

A **Circular Doubly Linked List** is a type of linked list where:

* Each node contains three parts:
  1. Prev – pointer to the previous node
  2. Data – the actual value
  3. Next – pointer to the next node
* The **last node's next** points to the **first node**, and the **first node's prev** points to the **last node**, forming a **circular structure** in both directions.

**🔹 Node Structure:**

↖ ↘ ↖ ↘ ↖ ↘

[Prev | Data | Next] ⇄ [Prev | Data | Next] ⇄ [Prev | Data | Next]

↑ ↓

└────────────(circular link)◄──────────────┘

**🔹 Characteristics:**

* No None in next or prev of any node.
* You can move forward and backward from any node.
* The traversal is circular in both directions.

**🔹 Operations:**

**✅ 1. Traversal**

* Forward: From head until you reach head again.
* Backward: From tail until you reach tail again.

**✅ 2. Insertion**

* At beginning
* At end

**✅ 3. Deletion**

* From beginning
* From end

**🔹 Python Code Example**

class CDNode:

def \_\_init\_\_(self, data):

self.data = data

self.prev = None

self.next = None

class CircularDoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

# Append at the end

def append(self, data):

new\_node = CDNode(data)

if not self.head:

self.head = new\_node

new\_node.next = new\_node

new\_node.prev = new\_node

return

tail = self.head.prev

tail.next = new\_node

new\_node.prev = tail

new\_node.next = self.head

self.head.prev = new\_node

# Display the list forward

def display(self):

if not self.head:

print("List is empty")

return

curr = self.head

while True:

print(curr.data, end=" ⇄ ")

curr = curr.next

if curr == self.head:

break

print("(back to head)")

**🔹 Example Usage**

cdll = CircularDoublyLinkedList()

cdll.append(11)

cdll.append(22)

cdll.append(33)

cdll.display()

**🔹 Output**

11 ⇄ 22 ⇄ 33 ⇄ (back to head)

**🔹 Advantages:**

* Can traverse in both directions.
* Circular nature is useful for **cyclic navigation** (e.g., music playlist, tab switcher).
* Efficient insertions and deletions at both ends.

**🔹 Disadvantages:**

* More complex due to both prev and next links.
* Uses more memory per node.