**### Limitations of Singly Linked Lists**

1. Unidirectional Traversal: In a singly linked list, traversal is only possible in one direction (from the head to the end of the list). This makes certain operations, such as reverse traversal, inefficient or impossible without additional modifications.

2. Inefficient Access to Nodes: Singly linked lists do not allow random access to elements (unlike arrays), so accessing a specific element requires a traversal from the head, which takes \(O(n)\) time in the worst case.

3. Inefficient for Reverse Operations: Reversing a singly linked list requires re-traversal of the list from the beginning or the use of an auxiliary data structure to keep track of previous nodes, making the operation inefficient.

4. Higher Memory Overhead for Pointers: Each node contains a pointer to the next node, leading to additional memory usage compared to arrays.

5. Difficulties with Deletion: Deleting a node in the middle requires the previous node's reference, so tracking the previous node is necessary for efficient deletion. This can make deletion operations more complex.

**### Limitations of Doubly Linked Lists**

1. Higher Memory Usage: Each node in a doubly linked list contains two pointers (one for the next node and one for the previous node), which doubles the memory overhead compared to singly linked lists.

2. Additional Complexity in Maintenance: Insertion and deletion require careful updating of two pointers (previous and next), which makes coding operations more complex and increases the chances of errors (e.g., memory leaks).

3. Slower Operations Compared to Arrays: As with singly linked lists, doubly linked lists lack random access, so accessing elements still requires traversal and takes \(O(n)\) time in the worst case.

4. Requires More Operations for Deletion: Although doubly linked lists allow bidirectional traversal, deletion is still complex because it requires updates to both the previous and next pointers, and special handling for edge cases like head and tail nodes.

5. Garbage Collection Complexity: In languages without automatic garbage collection, doubly linked lists can create memory leaks if nodes aren’t properly freed, as both forward and backward links need to be carefully managed.

A circular linked list is a variation of the linked list in which the last node points back to the first node, creating a circular structure. Unlike a standard linear linked list where the last node’s `next` pointer is `null` (or `None` in Python), in a circular linked list, the last node’s `next` pointer references the first node.

**### Types of Circular Linked Lists**

1. Circular Singly Linked List: Each node has a single `next` pointer that points to the next node in the sequence, with the last node’s `next` pointer referencing the head node.

2. Circular Doubly Linked List: Each node has two pointers: `next` (pointing to the next node) and `prev` (pointing to the previous node), with the `next` pointer of the last node pointing to the head, and the `prev` pointer of the head pointing to the last node.

**### Characteristics of a Circular Linked List**

- No Null End: Unlike linear lists, circular linked lists do not have a `null` or `None` at the end; instead, they form a continuous loop.

- Traversal: You can start from any node and keep traversing until you return to the starting node. This property is useful in applications where the end and start of the list need to be connected, like in scheduling applications.

- Bidirectional (for circular doubly linked lists): In a circular doubly linked list, you can traverse both forwards and backwards without encountering a `null` pointer.

**### Advantages of Circular Linked Lists**

- Efficient Memory Usage in Circular Structures: They are useful for implementing circular structures like round-robin scheduling, where processes cycle repeatedly.

- Quick Navigation from End to Start: Accessing the head from the tail (and vice versa in doubly linked lists) is efficient, making them useful in applications where both ends need to be connected.

**### Disadvantages of Circular Linked Lists**

- Complex Traversal: Special care is needed to detect the end of the list during traversal, as there's no `null` reference to signify the end.

- More Complex Deletion and Insertion Logic: Since the structure wraps around, inserting or deleting nodes requires extra handling, especially for the head and tail nodes.

**### Applications of Circular Linked Lists**

- Round-robin scheduling: Useful in process scheduling in operating systems, where each process is given equal CPU time in a circular fashion.

- Buffer Management: Often used in buffer management for cyclic buffer implementations.

- Games: Frequently used in player rotations for turn-based games where players are arranged in a cycle.

Circular linked lists offer a unique structure that is ideal for specific applications needing continuous cycling without reaching a traditional end.

In a circular linked list, the last node is defined as the node whose `next` pointer references the head node, creating the circular link back to the start. Here are some ways to check for the last node in a circular linked list, depending on whether it's a circular singly linked list or a circular doubly linked list.

**### 1. Checking the Last Node in a Circular Singly Linked List**

In a circular singly linked list, the last node is the one whose `next` pointer points to the head node. To find the last node, you can:

- Traverse from the head node.

- At each step, check if the `next` pointer of the current node points back to the head.

- If it does, you've found the last node.

Here’s an example in Python:

class Node:

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

self.data = data

self.next = None

class CircularLinkedList:

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

self.head = None

def is\_last\_node(self, node):

# Check if the node is the last node

return node.next == self.head

def find\_last\_node(self):

if self.head is None:

return None # Empty list

current = self.head

while current.next != self.head:

current = current.next

return current # This is the last node

**# Example usage:**

clist = CircularLinkedList()

# (Assume nodes are added to the list here)

last\_node = clist.find\_last\_node()

if last\_node:

print("Last node data:", last\_node.data)

**### 2. Checking the Last Node in a Circular Doubly Linked List**

In a circular doubly linked list, you can identify the last node similarly:

- The last node's `next` pointer points to the head.

- Additionally, the head’s `prev` pointer points to the last node.

Here’s an example in Python:

class DNode:

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

self.data = data

self.next = None

self.prev = None

class CircularDoublyLinkedList:

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

self.head = None

def is\_last\_node(self, node):

# Check if the node is the last node

return node.next == self.head

def find\_last\_node(self):

if self.head is None:

return None # Empty list

current = self.head

while current.next != self.head:

current = current.next

return current # This is the last node

# Example usage:

cdlist = CircularDoublyLinkedList()

# (Assume nodes are added to the list here)

last\_node = cdlist.find\_last\_node()

if last\_node:

print("Last node data:", last\_node.data)

### Summary

To check if a node is the last node:

- In both singly and doubly circular linked lists, the last node’s `next` pointer points to the head node.

- For doubly circular linked lists, the head’s `prev` pointer also points to the last node, giving an additional way to verify it’s the last node.

In a circular linked list, inserting a node at the beginning or the end requires some special handling because the list wraps around to the head node, creating a continuous loop. Let's go through how to handle these insertions in both a circular singly linked list and a circular doubly linked list.

**### Circular Singly Linked List**

**#### Insertion at the Beginning**

1. Create a new node.

2. If the list is empty, set the new node's `next` pointer to itself, and set it as the `head`.

3. If the list is not empty:

- Traverse the list until you find the last node (the node whose `next` points to the current head).

- Set the new node’s `next` to the current head.

- Update the last node’s `next` to point to the new node.

- Set the `head` to the new node.

Here's an example:

```python

class Node:

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

self.data = data

self.next = None

class CircularSinglyLinkedList:

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

self.head = None

def insert\_at\_beginning(self, data):

new\_node = Node(data)

if self.head is None: # List is empty

new\_node.next = new\_node

self.head = new\_node

else:

# Find the last node

last = self.head

while last.next != self.head:

last = last.next

new\_node.next = self.head

last.next = new\_node

self.head = new\_node

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.next

if current == self.head:

break

print()

# Example usage:

clist = CircularSinglyLinkedList()

clist.insert\_at\_beginning(10)

clist.insert\_at\_beginning(20)

clist.display() # Output: 20 -> 10 ->

**#### Insertion at the End**

1. Create a new node.

2. If the list is empty, set the new node’s `next` to itself, and set it as the `head`.

3. If the list is not empty:

- Traverse to the last node (the node whose `next` points to the head).

- Set the last node’s `next` to the new node.

- Set the new node’s `next` to the head.

Example:

def insert\_at\_end(self, data):

new\_node = Node(data)

if self.head is None: # List is empty

new\_node.next = new\_node

self.head = new\_node

else:

last = self.head

while last.next != self.head:

last = last.next

last.next = new\_node

new\_node.next = self.head

# Example usage:

clist.insert\_at\_end(30)

clist.display() # Output: 20 -> 10 -> 30 ->

**### Circular Doubly Linked List**

**#### Insertion at the Beginning**

1. Create a new node.

2. If the list is empty, set the new node’s `next` and `prev` to itself, and set it as the `head`.

3. If the list is not empty:

- Set the new node’s `next` to the current head and its `prev` to the last node (the current head's `prev`).

- Update the last node’s `next` to point to the new node.

- Set the head’s `prev` to the new node.

- Set the `head` to the new node.

**Example:**

class DNode:

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

self.data = data

self.next = None

self.prev = None

class CircularDoublyLinkedList:

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

self.head = None

def insert\_at\_beginning(self, data):

new\_node = DNode(data)

if self.head is None: # List is empty

new\_node.next = new\_node

new\_node.prev = new\_node

self.head = new\_node

else:

last = self.head.prev

new\_node.next = self.head

new\_node.prev = last

last.next = new\_node

self.head.prev = new\_node

self.head = new\_node

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" <-> ")

current = current.next

if current == self.head:

break

print()

**# Example usage:**

cdlist = CircularDoublyLinkedList()

cdlist.insert\_at\_beginning(10)

cdlist.insert\_at\_beginning(20)

cdlist.display() # Output: 20 <-> 10 <->

**#### Insertion at the End**

1. Create a new node.

2. If the list is empty, set the new node’s `next` and `prev` to itself, and set it as the `head`.

3. If the list is not empty:

- Set the new node’s `next` to the head and its `prev` to the last node (head’s `prev`).

- Update the head’s `prev` to point to the new node.

- Update the last node’s `next` to point to the new node.

Example:

def insert\_at\_end(self, data):

new\_node = DNode(data)

if self.head is None: # List is empty

new\_node.next = new\_node

new\_node.prev = new\_node

self.head = new\_node

else:

last = self.head.prev

new\_node.next = self.head

new\_node.prev = last

last.next = new\_node

self.head.prev = new\_node

# Example usage:

cdlist.insert\_at\_end(30)

cdlist.display() # Output: 20 <-> 10 <-> 30 <->

These methods demonstrate how to perform insertions at both the beginning and the end in circular singly and doubly linked lists. The circular nature ensures that the last node points back to the head, creating a loop.

In a circular linked list, deletion can be a bit more complex than in a linear linked list because of the circular references. Here’s how to delete nodes at the beginning, end, or any specific position in both circular singly linked lists and circular doubly linked lists.

**### Circular Singly Linked List**

**#### 1. Deletion at the Beginning**

To delete the first node:

1. If the list is empty, do nothing.

2. If there's only one node (i.e., the node’s `next` points to itself), set the head to `None`.

3. If there are multiple nodes:

- Find the last node (node whose `next` points to the head).

- Update the last node’s `next` to point to the node after the head.

- Set the head to the node after the current head.

Example:

class Node:

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

self.data = data

self.next = None

class CircularSinglyLinkedList:

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

self.head = None

def delete\_at\_beginning(self):

if self.head is None:

print("List is empty")

return

if self.head.next == self.head: # Only one node

self.head = None

else:

last = self.head

while last.next != self.head:

last = last.next

last.next = self.head.next

self.head = self.head.next

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.next

if current == self.head:

break

print()

**#### 2. Deletion at the End**

To delete the last node:

1. If the list is empty, do nothing.

2. If there's only one node, set the head to `None`.

3. For multiple nodes:

- Traverse to the second-to-last node.

- Set its `next` to point to the head.

Example:

def delete\_at\_end(self):

if self.head is None:

print("List is empty")

return

if self.head.next == self.head: # Only one node

self.head = None

else:

second\_last = self.head

while second\_last.next.next != self.head:

second\_last = second\_last.next

second\_last.next = self.head

**#### 3. Deletion at a Specific Position**

To delete a node at a given position `pos`:

1. Handle special cases for deleting the first node or if the list is empty.

2. Traverse to the node just before the target position.

3. Update its `next` pointer to skip the target node.

Example:

def delete\_at\_position(self, pos):

if self.head is None:

print("List is empty")

return

if pos == 0: # Special case to delete the first node

self.delete\_at\_beginning()

return

current = self.head

for i in range(pos - 1):

if current.next == self.head:

print("Position out of bounds")

return

current = current.next

current.next = current.next.next

**### Circular Doubly Linked List**

**#### 1. Deletion at the Beginning**

To delete the first node:

1. If the list is empty, do nothing.

2. If there's only one node, set the head to `None`.

3. For multiple nodes:

- Update the head’s `next` node to point back to the last node.

- Update the last node’s `next` to the new head.

- Update the `head` pointer.

Example:

class DNode:

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

self.data = data

self.next = None

self.prev = None

class CircularDoublyLinkedList:

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

self.head = None

def delete\_at\_beginning(self):

if self.head is None:

print("List is empty")

return

if self.head.next == self.head: # Only one node

self.head = None

else:

last = self.head.prev

self.head = self.head.next

self.head.prev = last

last.next = self.head

**#### 2. Deletion at the End**

To delete the last node:

1. If the list is empty, do nothing.

2. If there's only one node, set the head to `None`.

3. For multiple nodes:

- Update the second-to-last node’s `next` to the head.

- Update the head’s `prev` to the new last node.

Example:

def delete\_at\_end(self):

if self.head is None:

print("List is empty")

return

if self.head.next == self.head: # Only one node

self.head = None

else:

last = self.head.prev

second\_last = last.prev

second\_last.next = self.head

self.head.prev = second\_last

```

**#### 3. Deletion at a Specific Position**

To delete a node at a given position `pos`:

1. Handle special cases for deleting the first node or if the list is empty.

2. Traverse to the node at the specified position.

3. Update the previous node’s `next` and the next node’s `prev` to bypass the target node.

Example:

```python

def delete\_at\_position(self, pos):

if self.head is None:

print("List is empty")

return

if pos == 0: # Special case to delete the first node

self.delete\_at\_beginning()

return

current = self.head

for i in range(pos):

current = current.next

if current == self.head:

print("Position out of bounds")

return

current.prev.next = current.next

current.next.prev = current.prev

Each deletion method carefully maintains the circular structure, ensuring that references to `head` and `prev` pointers (in doubly linked lists) are correctly updated. This way, the list remains circular even after nodes are deleted.

To insert a node at a specific position in a circular linked list, you need to adjust the pointers carefully to maintain the circular structure. Here are the methods for inserting a node at any given position in both circular singly linked lists and circular doubly linked lists.

---

**### Circular Singly Linked List**

**#### Insertion at a Specific Position**

To insert a node at a specific position:

1. If the list is empty and the position is `0`, set the new node as the head and make its `next` point to itself.

2. If the position is `0`, call the `insert\_at\_beginning` method.

3. For any other position:

- Traverse to the node just before the target position.

- Insert the new node by adjusting the `next` pointers accordingly.

Example:

class Node:

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

self.data = data

self.next = None

class CircularSinglyLinkedList:

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

self.head = None

def insert\_at\_position(self, data, pos):

new\_node = Node(data)

# Insert at the beginning if position is 0

if pos == 0:

if self.head is None: # List is empty

self.head = new\_node

new\_node.next = self.head

else:

# Insert new node before the head and adjust last node's next to the new head

last = self.head

while last.next != self.head:

last = last.next

new\_node.next = self.head

last.next = new\_node

self.head = new\_node

return

# Insert at the specified position

current = self.head

for i in range(pos - 1):

current = current.next

if current == self.head: # Reached the end before the position

print("Position out of bounds")

return

new\_node.next = current.next

current.next = new\_node

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" -> ")

current = current.next

if current == self.head:

break

print()

**### Circular Doubly Linked List**

**#### Insertion at a Specific Position**

To insert a node at a specific position:

1. If the list is empty and the position is `0`, set the new node as the head and make its `next` and `prev` pointers refer to itself.

2. If the position is `0`, call the `insert\_at\_beginning` method.

3. For any other position:

- Traverse to the node just before the target position.

- Adjust the `next` and `prev` pointers of the nodes around the insertion point to link the new node into the list.

Example:

```python

class DNode:

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

self.data = data

self.next = None

self.prev = None

class CircularDoublyLinkedList:

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

self.head = None

def insert\_at\_position(self, data, pos):

new\_node = DNode(data)

# Insert at the beginning if position is 0

if pos == 0:

if self.head is None: # List is empty

self.head = new\_node

new\_node.next = new\_node

new\_node.prev = new\_node

else:

# Insert new node before the head and adjust last node's next and new head's prev

last = self.head.prev

new\_node.next = self.head

new\_node.prev = last

last.next = new\_node

self.head.prev = new\_node

self.head = new\_node

return

# Insert at the specified position

current = self.head

for i in range(pos - 1):

current = current.next

if current == self.head: # Reached the end before the position

print("Position out of bounds")

return

new\_node.next = current.next

new\_node.prev = current

current.next.prev = new\_node

current.next = new\_node

def display(self):

if self.head is None:

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" <-> ")

current = current.next

if current == self.head:

break

print()

Each method ensures that the circular structure of the list is maintained by carefully updating the `next` and `prev` pointers, allowing smooth traversal around the circular list. These insertions are efficient and help maintain the circular integrity of both singly and doubly linked lists.

Here’s a comparison of singly, doubly, and circular linked lists based on factors like memory allocation, performance, and their typical use cases:

**### 1. Memory Allocation**

- Singly Linked List:

- Requires minimal memory since each node has only one pointer (usually called `next`) pointing to the next node.

- Each node needs space for one data field and one pointer.

- Doubly Linked List:

- Requires more memory than singly linked lists, as each node has two pointers (`next` and `prev`), one pointing to the next node and one to the previous node.

- Each node needs space for one data field and two pointers.

- Circular Linked List:

- Memory usage depends on whether it's a singly or doubly circular linked list.

- Like singly or doubly linked lists, circular singly linked lists require one pointer per node, and circular doubly linked lists require two.

- However, a circular structure eliminates the need for a separate pointer to the end node, as the last node’s pointer refers back to the head.

**### 2. Performance (Time Complexity)**

- Insertion:

- Singly linked lists and circular singly linked lists allow O(1) insertion at the beginning but O(n) insertion at any other position unless a reference to the node is already available.

- Doubly linked lists provide efficient O(1) insertion both at the beginning and the end due to the availability of `prev` pointers.

- Circular doubly linked lists maintain similar insertion efficiency at both ends but can also improve traversal to the head or tail by referencing nodes in either direction.

- Deletion:

- In singly linked lists, deletion at the beginning is O(1), but deleting at any specific position is O(n) due to one-directional traversal.

- Doubly linked lists allow O(1) deletion at both ends and efficient deletion of nodes in between if a pointer to the target node is available, thanks to bidirectional pointers.

- Circular linked lists handle deletion similarly to singly or doubly linked lists depending on type, with efficient circular deletion when deleting nodes at the start or end.

- Traversal:

- Singly linked lists and circular singly linked lists are slower for backward traversal since they only move in one direction.

- Doubly linked lists and circular doubly linked lists allow traversal in both directions, making them more flexible for certain applications, such as navigating back and forth in a list of items.

**### 3. Ease of Implementation**

- Singly Linked List:

- Easiest to implement due to a single `next` pointer and no circular reference requirements.

- Doubly Linked List:

- More complex than singly linked lists, requiring extra code to handle `prev` pointers.

- Circular Linked List:

- Circular singly linked lists add some complexity by needing to maintain circular pointers.

- Circular doubly linked lists are the most complex due to managing both `next` and `prev` pointers in a circular structure.

**### 4. Typical Use Cases**

- Singly Linked List:

- Ideal for applications where memory is a concern and only forward traversal is required.

- Commonly used in stack implementations, adjacency lists for graphs, and simple list-based structures.

- Doubly Linked List:

- Suitable for applications that require efficient bidirectional traversal, such as navigation applications or undo-redo functionality.

- Commonly used in implementation of complex data structures like deques, stacks with O(1) access at both ends, and certain memory management applications.

- Circular Linked List:

- Useful in applications where continuous looping through elements is required.

- Commonly used in round-robin scheduling, multiplayer gaming systems (for turn-based games), and in implementing buffers and queues.

**### 5. Advantages and Disadvantages**

- Singly Linked List:

- Advantages: Simple and memory efficient for one-directional traversal.

- Disadvantages: Limited to forward traversal; slower performance for certain operations like deletion in the middle.

- Doubly Linked List:

- Advantages: Flexible with both forward and backward traversal, efficient for both ends.

- Disadvantages: Requires more memory and has higher implementation complexity.

- Circular Linked List:

- Advantages: Ideal for applications that need continuous looping, eliminating the need to reset pointers manually.

- Disadvantages: More complex to implement and handle edge cases, especially with circular doubly linked lists.

**QUESTION: Circular Linked List Implementation**

class Node:

"""A class to represent a node in the circular linked list."""

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

self.data = data # Store the data

self.next = None # Reference to the next node

class CLL:

"""A class to implement a Circular Linked List."""

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

self.last = None # Initialize the last node reference

def is\_empty(self):

"""Check if the circular linked list is empty."""

return self.last is None # True if last is None, indicating the list is empty

def insert\_at\_start(self, data):

"""Insert an element at the start of the list."""

new\_node = Node(data)

if self.is\_empty():

# If the list is empty, point new node to itself

self.last = new\_node

self.last.next = new\_node

else:

# Insert new node at the start

new\_node.next = self.last.next # New node points to the first node

self.last.next = new\_node # Last node points to the new node

def insert\_at\_last(self, data):

"""Insert an element at the end of the list."""

new\_node = Node(data)

if self.is\_empty():

# If the list is empty, point new node to itself

self.last = new\_node

self.last.next = new\_node

else:

new\_node.next = self.last.next # New node points to the first node

self.last.next = new\_node # Current last node points to new node

self.last = new\_node # Update the last to the new node

def search(self, key):

"""Search for a node with the specified value."""

if self.is\_empty():

return None # If list is empty, return None

current = self.last.next # Start from the first node

while True:

if current.data == key:

return current # Node found

current = current.next

if current == self.last.next: # Completed one cycle

break

return None # Node not found

def insert\_after(self, prev\_node\_data, data):

"""Insert a new node after a given node with specified data."""

if self.is\_empty():

return # If the list is empty, do nothing

prev\_node = self.search(prev\_node\_data) # Find the previous node

if not prev\_node:

print(f"Node with data {prev\_node\_data} not found.")

return

new\_node = Node(data)

new\_node.next = prev\_node.next # New node points to the next node

prev\_node.next = new\_node # Previous node points to the new node

def delete\_first(self):

"""Delete the first element from the list."""

if self.is\_empty():

print("List is empty. Nothing to delete.")

return

if self.last.next == self.last:

# If there's only one node

self.last = None

else:

# Remove the first node

self.last.next = self.last.next.next # Update last node's next to skip the first node

def delete\_last(self):

"""Delete the last element from the list."""

if self.is\_empty():

print("List is empty. Nothing to delete.")

return

if self.last.next == self.last:

# If there's only one node

self.last = None

else:

current = self.last.next # Start from the first node

while current.next != self.last: # Find the second last node

current = current.next

current.next = self.last.next # Remove the last node

self.last = current # Update last to the second last node

def delete\_item(self, key):

"""Delete the node with the specified value."""

if self.is\_empty():

print("List is empty. Nothing to delete.")

return

current = self.last.next # Start from the first node

previous = self.last

while True:

if current.data == key:

if current == self.last:

self.delete\_last() # If it's the last node, delete it

elif current == self.last.next:

self.delete\_first() # If it's the first node, delete it

else:

previous.next = current.next # Bypass the current node

return

previous = current

current = current.next

if current == self.last.next: # Completed one cycle

break

print(f"Node with data {key} not found.")

def print\_list(self):

"""Print all elements in the circular linked list."""

if self.is\_empty():

print("List is empty.")

return

current = self.last.next # Start from the first node

while True:

print(current.data, end=" -> ")

current = current.next

if current == self.last.next: # Completed one cycle

break

print("(back to head)")

def \_\_iter\_\_(self):

"""Iterator to access all elements of the circular linked list."""

if self.is\_empty():

return

current = self.last.next # Start from the first node

while True:

yield current.data # Yield the data of the current node

current = current.next

if current == self.last.next: # Completed one cycle

break

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

cll = CLL()

cll.insert\_at\_last(10)

cll.insert\_at\_start(20)

cll.insert\_at\_last(30)

cll.insert\_after(20, 25)

print("Circular Linked List:")

cll.print\_list()

print("Searching for 25:", cll.search(25).data if cll.search(25) else "Not found")

print("Deleting first element:")

cll.delete\_first()

cll.print\_list()

print("Deleting last element:")

cll.delete\_last()

cll.print\_list()

print("Deleting node with value 25:")

cll.delete\_item(25)

cll.print\_list()