**A circular doubly linked list** is a type of linked list in which each node contains a reference to both the next and the previous node, and the last node points back to the first node, creating a circular structure. This allows traversal in both directions (forward and backward) and enables easier manipulation of the list's structure.

**### Structure of a Node**

Each node in a circular doubly linked list typically contains:

- `data`: the value stored in the node.

- `next`: a pointer/reference to the next node in the list.

- `prev`: a pointer/reference to the previous node in the list.

**### Operations on Circular Doubly Linked List**

1. Insertion

- At the beginning: A new node is added at the start of the list.

- At the end: A new node is added at the end of the list.

- At a specific location: A new node is inserted at a specified position in the list.

2. Deletion

- From the beginning: The node at the start of the list is removed.

- From the end: The node at the end of the list is removed.

- From a specific location: A node is removed from a specified position in the list.

3. Traversing

- From the beginning: Traverse the list starting from the first node.

- From the end: Traverse the list starting from the last node.

- From a specific location: Traverse the list starting from a specified position.

4. Searching

- From the beginning: Search for a value starting from the first node.

- From the end: Search for a value starting from the last node.

- From a specific location: Search for a value starting from a specified position.

5. Check for Empty: Verify if the list is empty by checking if the head pointer/reference is null.

**### Implementation in Python**

Here's a simple implementation of a circular doubly linked list with the aforementioned operations.

```python

class Node:

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

self.data = data

self.next = None

self.prev = None

class CircularDoublyLinkedList:

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

self.head = None

def is\_empty(self):

return self.head is None

# Insertion

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

new\_node = Node(data)

if self.is\_empty():

new\_node.next = new\_node.prev = new\_node

self.head = new\_node

else:

tail = self.head.prev

new\_node.next = self.head

new\_node.prev = tail

tail.next = new\_node

self.head.prev = new\_node

self.head = new\_node

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

new\_node = Node(data)

if self.is\_empty():

new\_node.next = new\_node.prev = new\_node

self.head = new\_node

else:

tail = self.head.prev

new\_node.next = self.head

new\_node.prev = tail

tail.next = new\_node

self.head.prev = new\_node

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

if position < 1:

print("Position should be >= 1")

return

new\_node = Node(data)

if position == 1:

self.insert\_at\_beginning(data)

return

current = self.head

for \_ in range(position - 2):

current = current.next

if current == self.head:

print("Position out of bounds")

return

new\_node.next = current.next

new\_node.prev = current

current.next.prev = new\_node

current.next = new\_node

# Deletion

def delete\_from\_beginning(self):

if self.is\_empty():

print("List is empty")

return

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

self.head = None

else:

tail = self.head.prev

tail.next = self.head.next

self.head.next.prev = tail

self.head = self.head.next

def delete\_from\_end(self):

if self.is\_empty():

print("List is empty")

return

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

self.head = None

else:

tail = self.head.prev

tail.prev.next = self.head

self.head.prev = tail.prev

def delete\_from\_position(self, position):

if self.is\_empty():

print("List is empty")

return

if position < 1:

print("Position should be >= 1")

return

if position == 1:

self.delete\_from\_beginning()

return

current = self.head

for \_ in range(position - 1):

current = current.next

if current == self.head:

print("Position out of bounds")

return

current.prev.next = current.next

current.next.prev = current.prev

# Traversing

def traverse\_forward(self):

if self.is\_empty():

print("List is empty")

return

current = self.head

while True:

print(current.data, end=" ")

current = current.next

if current == self.head:

break

print()

def traverse\_backward(self):

if self.is\_empty():

print("List is empty")

return

current = self.head.prev

while True:

print(current.data, end=" ")

current = current.prev

if current == self.head.prev:

break

print()

# Searching

def search(self, data):

if self.is\_empty():

print("List is empty")

return

current = self.head

index = 1

while True:

if current.data == data:

print(f"Found {data} at position {index}")

return

current = current.next

index += 1

if current == self.head:

break

print(f"{data} not found in the list")

# Example Usage

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

cdll = CircularDoublyLinkedList()

# Insertion

cdll.insert\_at\_beginning(10)

cdll.insert\_at\_end(20)

cdll.insert\_at\_position(15, 2)

# Traversing

print("Traversing forward:")

cdll.traverse\_forward()

# Searching

cdll.search(15)

cdll.search(25)

# Deletion

cdll.delete\_from\_beginning()

print("After deleting from the beginning:")

cdll.traverse\_forward()

cdll.delete\_from\_end()

print("After deleting from the end:")

cdll.traverse\_forward()

cdll.delete\_from\_position(1)

print("After deleting from position 1:")

cdll.traverse\_forward()

# Check if the list is empty

print("Is the list empty?", cdll.is\_empty())

**### Explanation of Code**

1. Node Class: Represents each node in the circular doubly linked list, storing data and references to the next and previous nodes.

2. CircularDoublyLinkedList Class:

- Contains methods to perform various operations like insertion, deletion, traversing, searching, and checking if the list is empty.

- Insertion Methods: Add a new node at the beginning, end, or a specified position, updating the relevant pointers accordingly.

- Deletion Methods: Remove nodes from the beginning, end, or a specified position, handling edge cases where the list may become empty.

- Traversal Methods: Traverse the list forward or backward, printing the node values.

- Search Method: Searches for a value in the list and returns its position if found.

- is\_empty Method: Checks whether the list is empty.

### Example Usage

The code includes an example usage that demonstrates how to create a circular doubly linked list, perform insertions, traversals, searches, and deletions.