# Circular Linked List in Python

This document provides a **simple, clean, and well-commented implementation** of a Circular Linked List in Python. It also includes **descriptive explanations and notes** for each operation to help understand how the data structure works.

## What is a Circular Linked List?

A **circular linked list** is a linear data structure where each node points to the **next node**, and the last node points back to the **first node**, forming a circle.

There are two types:

* **Singly Circular Linked List**: Only next pointers are used.
* **Doubly Circular Linked List**: Both prev and next pointers are used.

In this implementation, we’ll focus on **Singly Circular Linked List**.

## Node Structure

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data # Data of the node  
 self.next = None # Pointer to the next node (circular reference later)

## CircularLinkedList Class

This class maintains a tail pointer (last node), which makes insertion and traversal easier.

class CircularLinkedList:  
 def \_\_init\_\_(self):  
 self.tail = None # Tail points to the last node in the list

## Time Complexity Comparison

| Operation | Singly Linked List | Doubly Linked List | Circular Linked List |
| --- | --- | --- | --- |
| Insert at Beginning | O(1) | O(1) | O(1) |
| Insert at End | O(n) | O(n) | O(1) (with tail) |
| Insert at Position | O(n) | O(n) | O(n) |
| Delete at Beginning | O(1) | O(1) | O(1) |
| Delete at End | O(n) | O(n) | O(n) |
| Delete at Position | O(n) | O(n) | O(n) |
| Search | O(n) | O(n) | O(n) |
| Update at Position | O(n) | O(n) | O(n) |
| Length | O(n) | O(n) | O(n) |

**Note:**

* Circular linked lists allow looping through the list without checking for None, but care must be taken to avoid infinite loops.
* Insertion at end is efficient if we maintain a tail pointer.

## Detailed Operation Explanations and Use Cases

### 1. Insert at Beginning

**How it works:** A new node is inserted right after the tail node. This makes the new node the effective head of the list. **Use case:** Useful in round-robin scheduling where a new task is added with high priority.

### 2. Insert at End

**How it works:** A new node is added after the tail, and the tail pointer is updated to point to the new node. **Use case:** Used in streaming systems where new data keeps coming at the end of the stream.

### 3. Delete at Beginning

**How it works:** The node after the tail (i.e., the head) is removed, and tail’s next pointer is updated. **Use case:** Removing a completed job in a cyclic task manager.

### 4. Delete at End

**How it works:** Requires traversal to the node before the tail. Once found, it becomes the new tail. **Use case:** Used when the oldest task in a queue must be discarded.

### 5. Display List

**How it works:** Starts from the head (tail.next) and loops until it comes back to the same node. **Use case:** Viewing all participants in a game rotation or network ring.

### 6. Length

**How it works:** Loops through the list counting each node. **Use case:** Checking how many processes are running in a circular scheduler.

### 7. Search

**How it works:** Traverse from head to tail checking each node’s data. **Use case:** Verifying if a particular user is still in a conference call loop.

## Real-World Applications

* **CPU Scheduling**: Circular linked lists are used in round-robin scheduling algorithms.
* **Multiplayer Games**: Managing players’ turns in a circle.
* **Network Topology**: Token ring networks use circular structure.
* **Music Playlist Loops**: Where the end links back to the beginning for continuous play.
* **Circular Buffers**: Implementing buffers that wrap around (like audio/video streaming buffers).

## Full Python Code

# Node class for singly circular linked list  
class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data # Store data  
 self.next = None # Reference to the next node  
  
  
# Circular Linked List class  
class CircularLinkedList:  
 def \_\_init\_\_(self):  
 self.tail = None # Tail node to simplify operations  
  
 # Insert at the beginning of the list  
 # Description: Adds a new node right after tail, becomes new head  
 def insert\_at\_beginning(self, data):  
 new\_node = Node(data)  
  
 # If list is empty, point node to itself and make it tail  
 if self.tail is None:  
 new\_node.next = new\_node  
 self.tail = new\_node  
 else:  
 new\_node.next = self.tail.next  
 self.tail.next = new\_node  
  
 # Insert at the end of the list  
 # Description: Adds a new node after the tail and updates tail pointer  
 def insert\_at\_end(self, data):  
 new\_node = Node(data)  
  
 # If list is empty  
 if self.tail is None:  
 new\_node.next = new\_node  
 self.tail = new\_node  
 else:  
 new\_node.next = self.tail.next  
 self.tail.next = new\_node  
 self.tail = new\_node  
  
 # Delete from beginning  
 # Description: Removes the head node (node after tail)  
 def delete\_at\_beginning(self):  
 if self.tail is None:  
 print("List is empty!")  
 return  
  
 head = self.tail.next  
  
 # If only one node  
 if head == self.tail:  
 self.tail = None  
 else:  
 self.tail.next = head.next  
  
 # Delete from end  
 # Description: Removes the tail node (requires traversal)  
 def delete\_at\_end(self):  
 if self.tail is None:  
 print("List is empty!")  
 return  
  
 head = self.tail.next  
  
 # If only one node  
 if head == self.tail:  
 self.tail = None  
 return  
  
 # Traverse to node before tail  
 current = head  
 while current.next != self.tail:  
 current = current.next  
  
 current.next = self.tail.next  
 self.tail = current  
  
 # Display the list  
 # Description: Prints all elements in circular manner  
 def display(self):  
 if self.tail is None:  
 print("List is empty!")  
 return  
  
 current = self.tail.next  
 while True:  
 print(current.data, end=" -> ")  
 current = current.next  
 if current == self.tail.next:  
 break  
 print("(back to start)")  
  
 # Count length of list  
 # Description: Returns the number of nodes in the list  
 def length(self):  
 if self.tail is None:  
 return 0  
  
 count = 1  
 current = self.tail.next  
 while current != self.tail:  
 count += 1  
 current = current.next  
  
 return count  
  
 # Search for a value  
 # Description: Returns True if value is found  
 def search(self, key):  
 if self.tail is None:  
 return False  
  
 current = self.tail.next  
 while True:  
 if current.data == key:  
 return True  
 current = current.next  
 if current == self.tail.next:  
 break  
  
 return False  
  
  
# Example Usage  
if \_\_name\_\_ == "\_\_main\_\_":  
 cll = CircularLinkedList()  
  
 cll.insert\_at\_end(10)  
 cll.insert\_at\_end(20)  
 cll.insert\_at\_beginning(5)  
 cll.insert\_at\_end(30)  
  
 cll.display()  
 print("Length:", cll.length())  
 print("Search 20:", cll.search(20))  
 print("Search 100:", cll.search(100))  
  
 cll.delete\_at\_beginning()  
 cll.display()  
 cll.delete\_at\_end()  
 cll.display()