Reverse Circular Linked List

Node Class:

This class defines the basic structure of a linked list node, containing a **next** pointer and a **value** attribute.

CircularLinkedList Class:

This class implements a circular linked list. It has private member variables **head** and **tail** which point to the head and tail nodes of the circular linked list. The **length** variable keeps track of the number of elements in the list.

Constructor:

- Initializes an empty circular linked list with **head**, **tail**, and **length** set to **nullptr** and **0**, respectively.

isEmpty Function:

- Checks if the linked list is empty by examining whether the **head** is **nullptr**.
- Time Complexity: O(1)
- Space Complexity: O(1)

insertWhileEmpty Function:

- Inserts a node into an empty linked list.
- Sets the **head**, **tail**, and the **next** pointers of the node to create a circular link.
- Time Complexity: O(1)
- Space Complexity: O(1)

pushBack Function:

- Appends a new node with the given value to the end of the circular linked list.
- If the list is empty, it calls **insertWhileEmpty**.
- Otherwise, it updates the **next** pointers to maintain the circular structure.
- Time Complexity: O(1)
- Space Complexity: O(1)

display Function:

- Displays the elements of the circular linked list.
- Time Complexity: O(n) where n is the number of elements in the list.
- Space Complexity: O(1)

Destructor:

- Frees the memory of all nodes in the circular linked list.
- Time Complexity: O(n) where n is the number of elements in the list.
- Space Complexity: O(1)

Main Function:

- Creates an instance of the **CircularLinkedList** class.
- Appends elements to the circular linked list using the **pushBack** function.
- Displays the original list.
- Reverses the list using the **reverseIteratively** function and displays it.
- Reverses the list again using the **reverseUsingStack** function and displays it.

Approach 1: Reverse the linked list iteratively

In this approach, we'll iteratively reverse the circular linked list by updating the **next** pointers of the nodes. Here's a step-by-step breakdown of how the process works:

1. **Initialize Pointers**: We start with three pointers:
    - **currNode**: Points to the current node we're processing.
    - **prevNode**: Points to the previous node that has already been reversed.
    - **nextNode**: Temporary pointer to store the next node before modifying the **next** pointer of the current node.
2. **Iterative Reversal**:
    - While **currNode** is not equal to the initial **head** node, we perform the following steps:
        - Store the current node's **next** pointer in **nextNode**.
        - Update the current node's **next** pointer to point to the previous node (**prevNode**).
        - Move **prevNode** to the current node (**currNode**).
        - Move **currNode** to the next node (**nextNode**).
3. **Update Head and Tail**:
    - Once the loop completes, **prevNode** will point to the last node in the original list, and **currNode** will be the new head of the reversed list.

- Update the **head** pointer to point to **currNode**, which is the start of the reversed list.
- Update the **next** pointer of the tail to point to the new head to complete the circular linkage.

This approach effectively reverses the direction of the **next** pointers, resulting in a reversed circular linked list.

Time Complexity: O(n) where n is the number of elements in the list.

Space Complexity: O(1)

Approach 2: Reverse the linked list using a stack

In this approach, we'll reverse the circular linked list using a stack data structure. Here's how the process works:

1. **Push Nodes onto Stack**:
    - Start from the initial **head** node and traverse the circular linked list, pushing each node onto the stack. Since the stack follows the Last-In-First-Out (LIFO) principle, the nodes will be stored in reverse order.
2. **Pop Nodes and Update Pointers**:
    - After all nodes are pushed onto the stack, start popping nodes off the stack. Each time you pop a node, update the **next** pointer of the previously popped node to point to the current node.
    - Continue popping nodes until the stack is empty. This step effectively reverses the direction of the **next** pointers.
3. **Update Head and Tail**:
    - The node at the top of the stack after all popping operations is the last node of the original list, which will be the new **head** of the reversed list.
    - Update the **head** pointer to point to this node.
    - Also, update the **next** pointer of the last node to point to the new **head** to complete the circular linkage.

This approach essentially utilizes the stack to temporarily store the nodes in reverse order, allowing us to reverse the **next** pointers efficiently.

Time Complexity: O(n)

Space Complexity: O(n)