708. Insert into a Sorted Circular Linked List

Medium Linked List

Leetcode Link

Problem Description

This problem involves a special kind of linked list called a Circular Linked List, where the last element points back to the first, creating a closed loop. Your task is to insert a new value into the list so that the list's non-descending (sorted) order is maintained.

The nuances of this task are:

1. The node you are given could be any node in the loop, not necessarily the smallest value.
2. You may insert the new value into any position where it maintains the sort order.
3. If the list is empty (i.e., the given node is null), you must create a new circular list that contains the new value and return it.

You're expected to return a reference to any node in the modified list.

Intuition

The intuition behind the solution starts with understanding the properties of a sorted circular linked list. In such a list, values gradually increase as you traverse the list, before suddenly "resetting" to the smallest value once the loop completes.

Given these properties, the solution involves handling three distinct scenarios:

1. *The list is empty:* If the list is empty, you create a new node that points to itself and return it as the new list.
2. *Normal insertion:* In a populated list, you traverse the nodes looking for the right place to insert the new value. The right place would be between two nodes where the previous node's value is less than or equal to the new value, and the next node's value is greater than or equal to the new value.
3. *Edge insertion:* If you don't find such a position, it means the new value either goes before the smallest value in the list or after the largest one. This specific case occurs at the "reset" point where the last node in sort order (with the largest value) points to the first node (with the smallest value).

The algorithm involves iterating through the list, comparing the `insertVal` with the current and next node values to find the correct insertion point. Once found, adjust the necessary pointers to insert the new node.

We also guarantee that we loop no more than once by comparing the current node to the initial head node. If we reach the head again, it means we've checked every node, which handles cases where all nodes in the list have the same value and the `insertVal` could go anywhere.

So the high-level steps are:

- Handle the edge case where the list is empty.
- Traverse the list looking for the insertion point.
- Insert the node when the correct spot is found.
- If the loop completes without finding the exact position, insert the node just after the largest value or before the smallest value.

With this approach, we insert the new value while maintaining the list's sorted order.

Solution Approach

The solution implementation involves a straightforward but careful traversal of the circular linked list to maintain its sorted property after insertion.

First, we need to handle a special case:

- **Empty list**: If the list is empty (`head is None`), we create a new node that points to itself and return it as the new list head (circular in nature).

For a non-empty list, we follow these steps:

1. Initialize two pointers, `prev` and `curr` - starting from `head` and `head.next` respectively. We'll move these pointers forward until we find the correct insert location.
2. Loop through the circular list and look for the correct position to insert the `insertVal`. The loop will terminate if:
   - We find a spot where the value directly after `prev` (`curr`) is greater than or equal to `insertVal`, and the value at `prev` is less than or equal to `insertVal`. This means `insertVal` fits right between `prev` and `curr`.
   - We find the point where the values "reset", indicating `prev` holds the maximum value and `curr` holds the minimum value in the list. If `insertVal` is greater than or equal to `prev.val` (greater than the list's max value) or `insertVal` is less than or equal to `curr.val` (less than the list's min value), it means `insertVal` must be inserted here.
3. For insertion:
   - Create a new node with `insertVal`.
   - Adjust the `next` pointer of `prev` to point to the new node.
   - Set the new node's `next` pointer to `curr`.
4. Return the head of the list.

Here is the algorithm in Python code snippet from the Reference Solution Approach:

```python
class Node:
    def __init__(self, val=None, next=None):
        self.val = val
        self.next = next

class Solution:
    def insert(self, head: 'Optional[Node]', insertVal: int) -> 'Node':
        node = Node(insertVal)
        if head is None:
            node.next = node
            return node
        prev, curr = head, head.next
        while curr != head:
            if prev.val <= insertVal <= curr.val or (
                prev.val > curr.val and (insertVal >= prev.val or insertVal <= curr.val)
            ):
                break
            prev, curr = curr, curr.next
        prev.next = node
        node.next = curr
        return head
```

By utilizing the `Node` class to instantiate the new element to be inserted and updating pointers in a singly linked list fashion, you efficiently maintain both the circular nature of the list and the sorted order with a simple yet effective algorithm that walks through the list once at most.

Example Walkthrough

Let's walk through a small example to illustrate the solution approach.

Suppose we have a circular sorted linked list and we need to insert a value `insertVal` into the list. Our circular linked list is as follows:

```
3 -> 4 -> 1
^         |
|_________|
```

Here, 3 is the smallest element before the reset and 1 is the last element pointing back to 3. Now let's say we want to insert 5. We must find the correct spot for 2 so that the list remains sorted in non-descending order. The step will be as follows:

1. Since the list is not empty, we initialize `prev` to this node (3) and `curr` to the next node (4).
2. We start traversing the list. As `insertVal` (5) is not between 3 (`prev.val`) and 4 (`curr.val`), we move forward.
3. Now, `prev` is 4 and `curr` is 1. We check:
   - Is 4 (`prev.val`) less than 5 (`insertVal`) and 5 (`insertVal`) less than 1 (`curr.val`)? No.
   - Does 4 (`prev.val`) > 1 (`curr.val`) indicating a reset point in the list ordering? Yes.
   - Is 5 (`insertVal`) >= 4 (`prev.val`) or 5 (`insertVal`) <= 1 (`curr.val`)? The first condition is true.
4. We have reached the reset point of our list, and since 5 (`insertVal`) is less than 1 (`curr.val`), it must be inserted between 4 (`prev`) and 1 (`curr`).
5. We create a new node with the value 5, update the `prev.next` to point to this new node, and set the new node's `next` pointer to `curr`.

Now our list looks like this:

```
3 -> 4 -> 5 -> 1
^              |
|______________|
```

6. We return the head of the list, which can be any node in the list; in this case, we still consider the node with value 3 as the head.

Through this example, we can see how the list is effectively iterated once to find the correct spot, and how the pointers are adjusted to maintain the order after insertion. The critical part of the approach is handling the edge cases where the `insertVal` is greater than the maximal value or smaller than the minimal value in the list.

Python Solution

```python
class Node:
    def __init__(self, value=None, next_node=None):
        self.value = value
        self.next_node = next_node


class Solution:
    def insert(self, head: 'Optional[Node]', insert_value: int) -> 'Node':
        # Create a new node with the insert_value
        new_node = Node(insert_value)

        # If the linked list is empty, initialize it with the new node
        if head is None:
            new_node.next_node = new_node  # Point the new_node to itself
            return new_node

        # Initialize two pointers for iterating the linked list
        previous, current = head, head.next_node

        # Traverse the linked list
        while current != head:
            # Check if the insert_value should be inserted between previous and current
            # The first condition checks for normal ordered insertion
            # The second condition checks for insertion at the boundary of the largest and smallest values
            if (
                previous.value <= insert_value <= current.value or
                (previous.value > current.value and (insert_value >= previous.value or insert_value <= current.value))
            ):
                break  # Correct insertion spot is found

            # Move to the next pair of nodes
            previous, current = current, current.next_node

        # Insert new_node between previous and current
        previous.next_node = new_node
        new_node.next_node = current

        # Return the head of the modified linked list
        return head

```

Java Solution

```java
// Class definition for a circular linked list node
class Node {
    public int value;
    public Node next;

    // Constructor for creating a new node without next reference
    public Node(int value) {
        this.value = value;
    }

    // Constructor for creating a new node with next reference
    public Node(int value, Node next) {
        this.value = value;
        this.next = next;
    }
}

class Solution {
    // Method to insert a node into a sorted circular linked list
    public Node insert(Node head, int insertValue) {
        // Create the node to be inserted
        Node newNode = new Node(insertValue);

        // If the linked list is empty, point the new node to itself and return it
        if (head == null) {
            newNode.next = newNode;
            return newNode;
        }

        // Pointers for tracking the current position and the previous node
        Node previous = head, current = head.next;

        // Iterate through the list to find the insertion point
        while (current != head) {

            // Check if the new node fits between previous and current nodes
            if ((previous.value <= insertValue && insertValue <= current.value)
                // or if it's the end/beginning of the list,
                // as the next value is less than the previous due to the circular nature
                || (previous.value > current.value && (insertValue >= previous.value || insertValue <= current.value))) {
                break; // Found the place to insert
            }

            // Move to next pair of nodes
            previous = current;
            current = current.next;
        }

        // Insert the new node between previous and current
        previous.next = newNode;
        newNode.next = current;

        // Return the head of the list
        return head;
    }
}

```

C++ Solution

```cpp
/*
// Definition for a circular singly-linked list Node.
class Node {
public:
    int val;      // Value of the node
    Node* next;   // Pointer to the next node

    Node() {}

    Node(int value) {
        val = value;
        next = nullptr;
    }

    Node(int value, Node* nextNode) {
        val = value;
        next = nextNode;
    }
};
*/

class Solution {
public:
    Node* insert(Node* head, int insertVal) {
        Node* newNode = new Node(insertVal); // Create a new node with the insertVal

        // If the list is empty, initialize it with the new node which points to itself
        if (!head) {
            newNode->next = newNode; // Points to itself to maintain circularity
            return newNode; // Return new node as the new head of the list
        }

        Node *prev = head;
        Node *current = head->next;
        bool inserted = false;

        while (true) {
            // Check if we have found the correct place to insert the new node
            if ((prev->val <= insertVal && insertVal <= current->val) ||   // Case 1: Value lies between prev and current
                (prev->val > current->val && (insertVal >= prev->val || insertVal <= current->val))) { // Case 2: At tail part after
                // Insert new node here
                prev->next = newNode;
                newNode->next = current;
                inserted = true;
                break;
            }

            prev = current;
            current = current->next;

            // If we completed one full circle around the list and didn't insert the new node, break loop to insert at the end.
            if (prev == head) {
                break;
            }
        }

        // If the new node hasn't been inserted yet, it should be placed between the tail and the head.
        // This case also covers a list with uniform values.
        if (!inserted) {
            prev->next = newNode;
            newNode->next = current;
        }

        // Return the head of the list
        return head;
    }
};

```

Typescript Solution

```typescript
// Definition for a circular singly-linked list node.
class ListNode {
    val: number;
    next: ListNode | null;

    constructor(val?: number, next?: ListNode | null) {
        this.val = (val === undefined ? 0 : val);
        this.next = (next === undefined ? null : next);
    }
}

// Function to create a new node with a given value
function createNode(value: number): ListNode {
    return new ListNode(value);
}

// Function to insert a new node with insertVal into the circular linked list
function insert(head: ListNode | null, insertVal: number): ListNode | null {
    // Create a new node with the insertVal
    const newNode = createNode(insertVal);

    // If the list is empty, initialize it with the new node which points to itself
    if (head === null) {
        newNode.next = newNode; // Points to itself to maintain circularity
        return newNode; // Return new node as the new head of the list
    }

    let prev = head;
    let current = head.next;
    let inserted = false;

    do {
        // Check if we have found the correct place to insert the new node
        // Case 1: Value lies between prev and current
        // Case 2: At tail part after the max val or before the min val in sorted circular list
        if ((prev.val <= insertVal && insertVal <= current.val) ||
            (prev.val > current.val && (insertVal >= prev.val || insertVal <= current.val))) {
            // Insert new node here
            prev.next = newNode;
            newNode.next = current;
            inserted = true;
            break;
        }

        prev = current;
        current = current.next;

        // If we completed one full circle around the list and didn't insert the new node, break loop to insert at the end.
    } while (prev !== head);

    // If the new node hasn't been inserted yet, it should be placed between the tail and the head.
    // This case also covers a list with uniform values.
    if (!inserted) {
        prev.next = newNode;
        newNode.next = current;
    }

    // Return the head of the list
    return head;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of nodes in the circular linked list. This complexity arises because in the worst-case scenario, the code would iterate through all the elements of the linked list once to find the correct position to insert the new value. The `while` loop continues until it returns to the head, meaning it could traverse the entire list.

Space Complexity

The space complexity of the given code is `O(1)`. The only extra space used is for creating the new node to be inserted, and no additional space that grows with the size of the input linked list is used.