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.
- Intuition
- gradually increase as you traverse the list, before suddenly "resetting" to the smallest value once the loop completes.
- You're expected to return a reference to any node in the modified list.
 - The intuition behind the solution starts with understanding the properties of a sorted circular linked list. In such a list, values

 - 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.

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.

- 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:

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

If the loop completes without finding the exact position, insert the node just after the largest value or before the smallest value.

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

• 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

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

(circular in nature).

First, we need to handle a special case:

after insertion.

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:

1. Initialize two pointers, prev and curr - starting from head and head next respectively. We'll move these pointers forward until we

• 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

 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 previval (greater than the list's max value) or insertVal is less than or equal

class Solution:

the list once at most.

3 -> 4 -> 1

Example Walkthrough

11

13

14

15

16

17

18

Create a new node with insertVal.

node = Node(insertVal)

node.next = node

prev, curr = head, head.next

prev, curr = curr, curr.next

if head is None:

return node

while curr != head:

break

):

Set the new node's next pointer to curr.

Adjust the next pointer of prev to point to this new node.

def insert(self, head: 'Optional[Node]', insertVal: int) -> 'Node':

if prev.val <= insertVal <= curr.val or (</pre>

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

to curr.val (less than the list's min value), it means insertVal must be inserted here. 3. For insertion:

than or equal to insertVal. This means insertVal fits right between prev and curr.

- 4. Return the head of the list. Here is the algorithm in Python code snippet from the Reference Solution Approach:
- 1 class Node: def __init__(self, val=None, next=None): self.val = val self.next = next
- 19 prev.next = node 20 node.next = curr 21 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

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:

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

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

must find the correct spot for 2 so that the list remains sorted in non-descending order. The step will be as follows:

2. We start traversing the list. As insertVal (5) is not between 3 (prev.val) and 4 (curr.val), we move forward.

1. Since the list is not empty, we initialize prev to this node (3) and curr to the next node (4).

prev.val > curr.val and (insertVal >= prev.val or insertVal <= curr.val)</pre>

 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.

curr.

(prev) and 1 (curr).

Now our list looks like this:

Python Solution

class Node:

class Solution:

13

14

15

16

17

18

19

20

23

24

25

26

27

28

29

35

36

37

38

39

40

36

37

38

39

40

41

42

43

44

45

46

48

49

50

51

52

53

54

55

57

8

9

10

12

13

14

15

16

17

18

20

21

24 25

26

27

28

29

30

31

32

33

34

19 };

*/

23 public:

56 }

3. Now, prev is 4 and curr is 1. We check:

5. We create a new node with the value 5, update the previnext to point to this new node, and set the new node's next pointer to

the maximal value or smaller than the minimal value in the list.

Create a new node with the insert_value

previous, current = head, head.next_node

new_node = Node(insert_value)

if head is None:

return new_node

Traverse the linked list

previous.next_node = new_node

// Class definition for a circular linked list node

Return the head of the modified linked list

// Constructor for creating a new node without next reference

// Check if the new node fits between previous and current nodes

// or if it's the end/beginning of the list,

break; // Found the place to insert

// Insert the new node between previous and current

// Move to next pair of nodes

previous = current;

previous.next = newNode;

newNode.next = current;

return head;

C++ Solution

class Node {

int val;

Node() {}

class Solution {

Node(int value) {

val = value;

val = value;

if (!head) {

Node *prev = head;

Node *current = head->next;

// Definition for a circular singly-linked list node.

// Function to create a new node with a given value

function createNode(value: number): ListNode {

// Create a new node with the insertVal

const newNode = createNode(insertVal);

return new ListNode(value);

if (head === null) {

let prev = head;

do {

constructor(val?: number, next?: ListNode | null) {

this.next = (next === undefined ? null : next);

// Function to insert a new node with insertVal into the circular linked list

function insert(head: ListNode | null, insertVal: number): ListNode | null {

// If the list is empty, initialize it with the new node which points to itself

newNode.next = newNode; // Points to itself to maintain circularity

// Check if we have found the correct place to insert the new node

// Case 1: Value lies between prev and current

return newNode; // Return new node as the new head of the list

this.val = (val === undefined ? 0 : val);

class ListNode {

9

10

11

14

16

19

20

22

23

24

25

26

27

28

29

30

31

32

33

34

35

15 }

val: number;

next: ListNode | null;

next = nextNode;

next = nullptr;

Node(int value, Node nextNode) {*

public:

current = current.next;

// Return the head of the list

2 // Definition for a circular singly-linked list Node.

Node* next; // Pointer to the next node

Node* insert(Node* head, int insertVal) {

// Value of the node

if ((previous.value <= insertValue && insertValue <= current.value)</pre>

Node* newNode = new Node(insertVal); // Create a new node with the insertVal

newNode->next = newNode; // Points to itself to maintain circularity

return newNode; // Return new node as the new head of the list

// If the list is empty, initialize it with the new node which points to itself

// as the next value is less than the previous due to the circular nature

new_node.next_node = current

while current != head:

if (

):

return head

public int value;

public Node next;

def insert(self, head: 'Optional[Node]', insert_value: int) -> 'Node':

If the linked list is empty, initialize it with the new node

Initialize two pointers for iterating the linked list

break # Correct insertion spot is found

new_node.next_node = new_node # Point the new_node to itself

The first condition checks for normal ordered insertion

previous.value <= insert_value <= current.value or</pre>

Check if the insert_value should be inserted between previous and current

The second condition checks for insertion at the boundary of the largest and smallest values

(previous.value > current.value and (insert_value >= previous.value or insert_value <= current.value))</pre>

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

def __init__(self, value=None, next_node=None): self.value = value self.next_node = next_node

30 31 # Move to the next pair of nodes 32 previous, current = current, current.next_node 33 # Insert new_node between previous and current 34

Java Solution

2 class Node {

```
public Node(int value) {
           this.value = value;
9
10
11
       // Constructor for creating a new node with next reference
12
       public Node(int value, Node next) {
13
           this.value = value;
           this.next = next;
14
15
16 }
17
   class Solution {
19
       // Method to insert a node into a sorted circular linked list
       public Node insert(Node head, int insertValue) {
20
21
           // Create the node to be inserted
           Node newNode = new Node(insertValue);
22
23
24
           // If the linked list is empty, point the new node to itself and return it
25
           if (head == null) {
26
               newNode.next = newNode;
27
               return newNode;
28
29
30
           // Pointers for tracking the current position and the previous node
           Node previous = head, current = head.next;
31
32
33
           // Iterate through the list to find the insertion point
34
           while (current != head) {
35
```

|| (previous.value > current.value && (insertValue >= previous.value || insertValue <= current.value))) {</pre>

35 bool inserted = false; 36 37 38 39

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 40 41 // Insert new node here prev->next = newNode; 42 43 newNode->next = current; 44 inserted = true; 45 break; 46 47 prev = current; 48 49 current = current->next; 50 51 // If we completed one full circle around the list and didn't insert the new node, break loop to insert at the end. 52 if (prev == head) { 53 break; 54 55 56 57 // If the new node hasn't been inserted yet, it should be placed between the tail and the head. 58 // This case also covers a list with uniform values. if (!inserted) { 59 prev->next = newNode; 60 newNode->next = current; 61 62 63 64 // Return the head of the list 65 return head; 66 67 }; 68 Typescript Solution

// Case 2: At tail part after the max val or before the min val in sorted circular list

let current = head.next;

let inserted = false;

if ((prev.val <= insertVal && insertVal <= current.val) ||</pre> 36 37 (prev.val > current.val && (insertVal >= prev.val || insertVal <= current.val))) {</pre> 38 // Insert new node here 39 prev.next = newNode; newNode.next = current; 41 inserted = true; 42 break; 43 44 45 prev = current; 46 current = current.next; 47 // If we completed one full circle around the list and didn't insert the new node, break loop to insert at the end. 48 49 } while (prev !== head); 50 // If the new node hasn't been inserted yet, it should be placed between the tail and the head. 51 // This case also covers a list with uniform values. 53 if (!inserted) { 54 prev.next = newNode; 55 newNode.next = current; 56 57 // Return the head of the list 58 return head; 59 60 } 61

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.

Time Complexity

Time and Space Complexity

Space Complexity The space complexity of the given code is 0(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.