**Steps to Split a Circularly Linked List into Two Equal Parts:**

1. **Check if the list is empty or contains only one node**:
   * If the list is empty or contains only one node, you cannot split it into two non-empty parts. In such a case, return early.
2. **Find the midpoint of the list**:
   * Since the list contains an even number of nodes, the midpoint will be at the n2\frac{n}{2}2n​-th node. Use a slow and fast pointer approach:
     + **Slow pointer**: Starts at the head and moves one step at a time.
     + **Fast pointer**: Starts at the head and moves two steps at a time.
   * When the fast pointer reaches the end of the list (the node pointing back to the head), the slow pointer will be at the midpoint.
3. **Adjust the pointers to split the list**:
   * Once the slow pointer reaches the midpoint, the node just before the midpoint will be the last node of the first list, and the node at the midpoint will be the head of the second list.
   * Disconnect the two halves:
     + Set the next pointer of the last node in the first list to point to the head of the first list, making the first list circular.
     + Set the next pointer of the last node in the second list to point to the head of the second list, making the second list circular.
4. **Return the two lists**:
   * You now have two circularly linked lists of half the size of the original list.

**Algorithm:**

1. **Check for special cases** (empty or single-node list).
2. **Use slow and fast pointers** to find the midpoint.
3. **Break the list into two parts** by adjusting the next pointers.

**Pseudocode:**

SplitCircularList(L):

if L is empty or L has only one node:

return empty or single list

slow = L.head

fast = L.head

// Traverse the list to find the midpoint

while fast.next != L.head and fast.next.next != L.head:

slow = slow.next

fast = fast.next.next

// Now, slow is at the midpoint, split the list

firstHalfHead = L.head

secondHalfHead = slow.next

// Break the circular link for first list

slow.next = firstHalfHead

// Find the last node of the second list

lastNodeOfSecond = secondHalfHead

while lastNodeOfSecond.next != L.head:

lastNodeOfSecond = lastNodeOfSecond.next

// Break the circular link for second list

lastNodeOfSecond.next = secondHalfHead

// Return the two lists

return firstHalfHead, secondHalfHead

**Explanation of the Steps:**

1. **Initial Check**:
   * If the list is empty (L.head == null), we return immediately because there's nothing to split.
   * If the list has only one node (L.head.next == L.head), splitting into two lists is not possible.
2. **Finding the Midpoint**:
   * We use two pointers, slow and fast. The slow pointer moves one node at a time, and the fast pointer moves two nodes at a time.
   * When the fast pointer reaches the end of the list, the slow pointer will be at the midpoint of the list.
   * This way, we find the node that will serve as the head of the second list.
3. **Adjusting the Pointers**:
   * After finding the midpoint, we set the next pointer of the last node of the first half (slow.next) to point to the head of the first list to maintain its circular nature.
   * We then traverse the second half of the list to find the last node and set its next pointer to point to the head of the second list to ensure it's also circular.
4. **Return Two Lists**:
   * Finally, we return the heads of both circularly linked lists.

**Example:**

Suppose we have the following circular linked list LLL with 6 nodes:

L: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> (back to 1)

The slow and fast pointers will meet as follows:

* Start with slow = 1, fast = 1.
* After the first iteration: slow = 2, fast = 3.
* After the second iteration: slow = 3, fast = 5.

The slow pointer is now at node 3, which is the midpoint of the list.

Now, split the list:

* The first half is 1 -> 2 -> 3 -> (back to 1).
* The second half is 4 -> 5 -> 6 -> (back to 4).

**Time Complexity:**

* **Time Complexity**: O(n)O(n)O(n), where nnn is the number of nodes in the list. We only traverse the list once to find the midpoint and then adjust the pointers.
* **Space Complexity**: O(1)O(1)O(1), as we are only using a constant amount of extra space.

**Edge Cases:**

1. **Empty list**: If the list is empty, it cannot be split.
2. **Single node list**: A list with one node cannot be split into two non-empty parts.
3. **Even number of nodes**: This is the expected case for the algorithm, where the list has an even number of nodes to split.