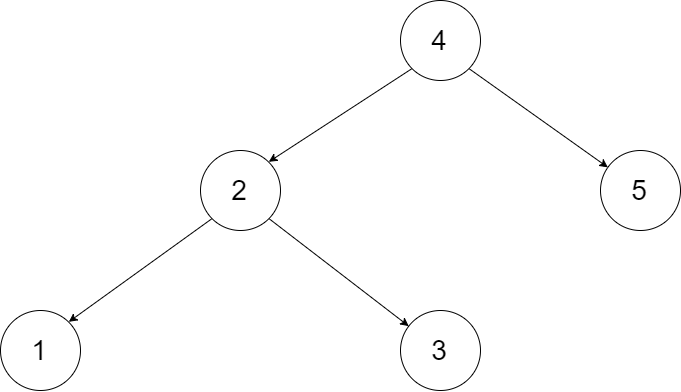
# Question

Convert a **Binary Search Tree** to a sorted **Circular Doubly-Linked List** in place.

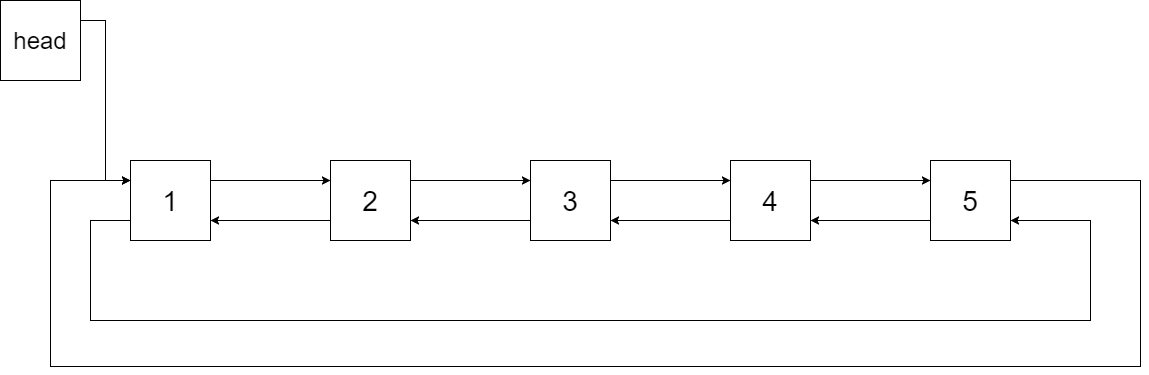
You can think of the left and right pointers as synonymous to the predecessor and successor pointers in a doubly-linked list. For a circular doubly linked list, the predecessor of the first element is the last element, and the successor of the last element is the first element.

We want to do the transformation **in place**. After the transformation, the left pointer of the tree node should point to its predecessor, and the right pointer should point to its successor. You should return the pointer to the smallest element of the linked list.

**Example 1:**

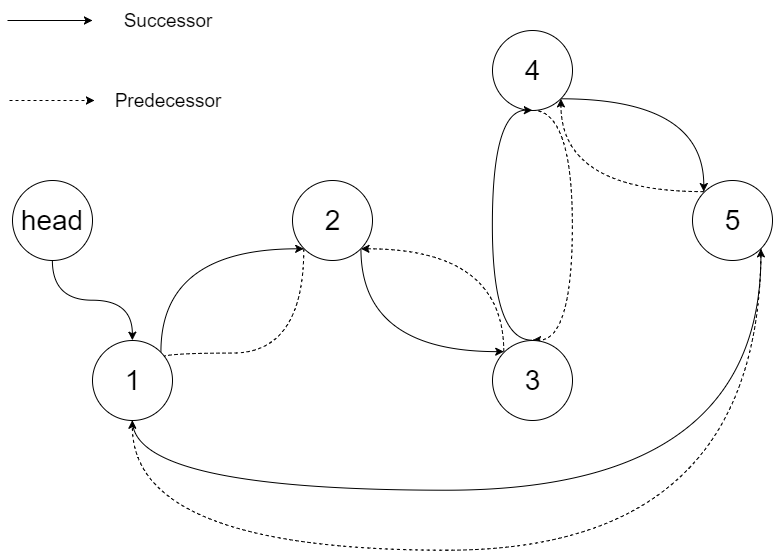


**Input:** root = [4,2,5,1,3]



**Output:** [1,2,3,4,5]

**Explanation:** The figure below shows the transformed BST. The solid line indicates the successor relationship, while the dashed line means the predecessor relationship.



**Example 2:**

**Input:** root = [2,1,3]

**Output:** [1,2,3]

**Example 3:**

**Input:** root = []

**Output:** []

**Explanation:** Input is an empty tree. Output is also an empty Linked List.

**Example 4:**

**Input:** root = [1]

**Output:** [1]

**Constraints:**

* -1000 <= Node.val <= 1000
* Node.left.val < Node.val < Node.right.val
* All values of Node.val are unique.
* 0 <= Number of Nodes <= 2000

# Solution

#### **How to traverse the tree**

There are two general strategies to traverse a tree:

* Depth First Search (DFS)

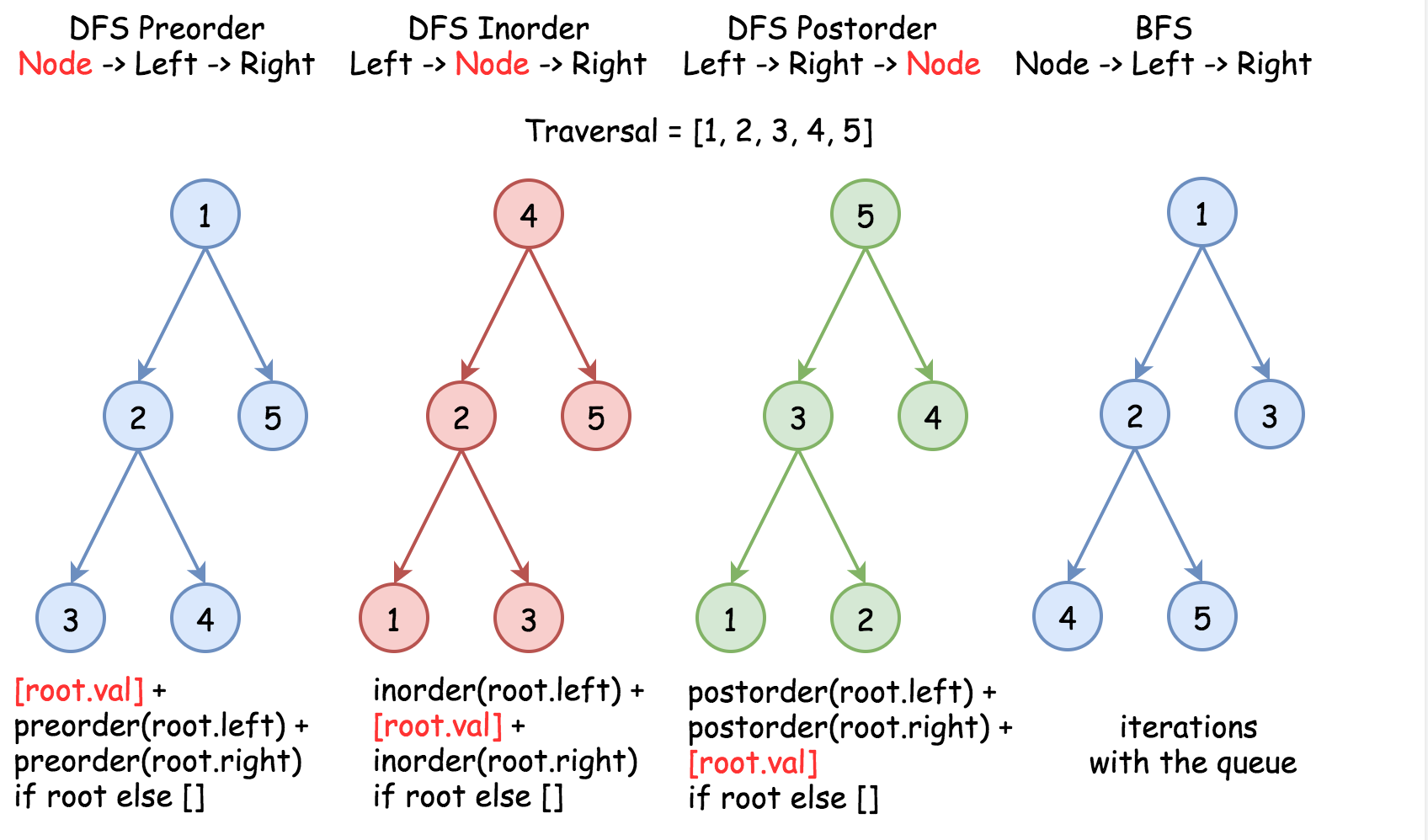
In this strategy, we adopt the depth as the priority, so that one would start from a root and reach all the way down to certain leaf, and then back to root to reach another branch.

The DFS strategy can further be distinguished as preorder, inorder, and postorder depending on the relative order among the root node, left node and right node.

* Breadth First Search (BFS)

We scan through the tree level by level, following the order of height, from top to bottom. The nodes on higher level would be visited before the ones with lower levels.

On the following figure the nodes are numerated in the order you visit them, please follow 1-2-3-4-5 to compare different strategies.



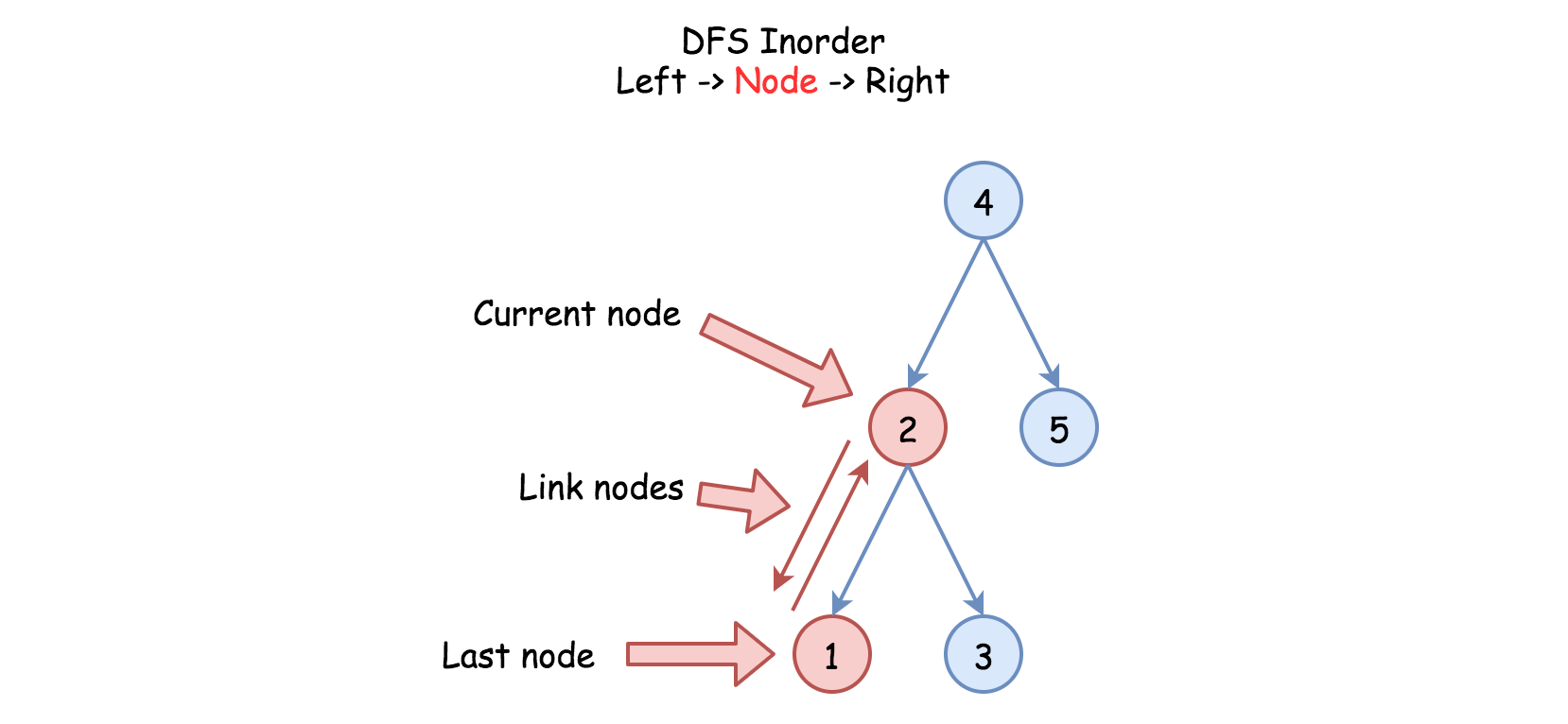
Here the problem is to implement DFS inorder traversal in a textbook recursion way because of in-place requirement.

#### **Approach 1: Recursion**

**Algorithm**

Standard inorder recursion follows left -> node -> right order, where left and right parts are the recursion calls and node part is where all processing is done.

Processing here is basically to link the previous node with the current one, and because of that one has to track the last node which is the largest node in a new doubly linked list so far.



One more detail : one has to keep the first, or the smallest, node as well to close the ring of doubly linked list.

Here is the algorithm :

* Initiate the first and the last nodes as nulls.
* Call the standard inorder recursion helper(root) :
  + If node is not null :
    - Call the recursion for the left subtree helper(node.left).
    - If the last node is not null, link the last and the current node nodes.
    - Else initiate the first node.
    - Mark the current node as the last one : last = node.
    - Call the recursion for the right subtree helper(node.right).
* Link the first and the last nodes to close DLL ring and then return the first node.

**Implementation**

1 / 9

|  |
| --- |
| **class Solution {**  **// the smallest (first) and the largest (last) nodes**  **Node first = null;**  **Node last = null;**  **public void helper(Node node) {**  **if (node != null) {**  **// left**  **helper(node.left);**  **// node**  **if (last != null) {**  **// link the previous node (last)**  **// with the current one (node)**  **last.right = node;**  **node.left = last;**  **}**  **else {**  **// keep the smallest node**  **// to close DLL later on**  **first = node;**  **}**  **last = node;**  **// right**  **helper(node.right);**  **}**  **}**  **public Node treeToDoublyList(Node root) {**  **if (root == null) return null;**  **helper(root);**  **// close DLL**  **last.right = first;**  **first.left = last;**  **return first;**  **}**  **}** |

**Complexity Analysis**

* Time complexity : \mathcal{O}(N)O(*N*) since each node is processed exactly once.
* Space complexity : \mathcal{O}(N)O(*N*). We have to keep a recursion stack of the size of the tree height, which is \mathcal{O}(\log N)O(log*N*) for the best case of completely balanced tree and \mathcal{O}(N)O(*N*) for the worst case of completely unbalanced tree.