Doubly-Linked List Problem Description The task is to convert a Binary Search Tree (BST) into a sorted Circular Doubly-Linked List, utilizing an in-place transformation. A

Medium Stack Tree

426. Convert Binary Search Tree to Sorted Doubly Linked List **Depth-First Search**

its predecessor.

structure:

doubly-linked list.

Solution Approach

Here's a step-by-step breakdown:

processed node in the in-order traversal.

argument root, which initially is the root of the BST.

predecessor) and root (the current node).

pointers to transform the BST structure into a doubly-linked list format.

of the doubly-linked list.

circular.

Initialization:

Example Walkthrough

Let's consider a simple BST for this example:

Let's walk through the transformation process:

Using DFS for In-order Traversal:

Recursive Traversal:

is updated to 5.

Closing the Circular List:

Python Solution

class Node:

class Solution:

10

11

13

14

15

16

17

18

19

20

21

23

24

26

27

28

29

30

31

32

33

34

35

36 37

39

40

44

45

46

47

48

6

9

10

11

12

13

14

15

16

17

18

19

20

21

22

24

25

26

27

28

30

31

32

33

34

35

36

37

38

39

40

43

44

45

46

48

49

50

51

52

53

54

55

57

8

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

32

33

34

35

36

37

38

39

41

42

43

44

45

46

48

10

11

28

29

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

57

58

59

60

61 }

12 }

47 };

56 }

C++ Solution

1 class Solution {

Node* prevNode;

Node* listHead;

2 public:

Post traversal, prev is pointing to 5.

self.value = value

self.right = right

def in_order_traverse(node):

in_order_traverse(node.left)

Process the current node.

previous.right = node

node.left = previous

in_order_traverse(node.right)

If the input tree is empty, return None.

Return the head of the doubly linked list.

// The 'head' node will serve as the head of the doubly linked list

return head; // Return the head of the doubly linked list

// Convert a binary search tree to a sorted, circular, doubly-linked list

// Initialize 'previous' and 'head' as null before the depth-first search

return null; // If the tree is empty, there is nothing to process or convert

// Start the inorder traversal from the root to convert the BST into a sorted list

// After the traversal, connect the last node with the 'head' to make it circular

// In the inorder traversal, 'previous' will be null only for the leftmost node

// Update 'previous' to be the current node before moving to the right subtree

// Connect the previous node's right to the current node

// Connect the current node's left to the previous node

// Main function to convert a BST to a sorted circular doubly-linked list.

if (!root) return nullptr; // If the tree is empty, return nullptr.

// Reset the prevNode and listHead pointers before starting the DFS.

// After the DFS is done, connect the head and tail to make it circular.

// Return the head of the doubly linked list.

if (!currentNode) return; // Base case: if the current node is null, just return.

// If this is the leftmost node, it will be the head of the doubly linked list.

// Perform a depth-first search to traverse the tree in order.

return; // Base case for the recursive function, stop if the current node is null

// If 'previous' is null, it means we are at the leftmost node which is the 'head' of the list

nonlocal previous, head

head = node

previous = node

if node is None:

return

if previous:

else:

if root is None:

return None

head = previous = None

in_order_traverse(root)

previous.right = head

head.left = previous

return head

private Node previous;

if (root == null) {

previous = null;

inorderTraversal(root);

previous.right = head;

head.left = previous;

if (node == null) {

head = null;

public Node treeToDoublyList(Node root) {

// Inorder traversal of the binary search tree

// Recursively process the left subtree

// Recursively process the right subtree

private void inorderTraversal(Node node) {

inorderTraversal(node.left);

previous.right = node;

node.left = previous;

inorderTraversal(node.right);

Node* treeToDoublyList(Node* root) {

prevNode->right = listHead;

void DFSInOrder(Node* currentNode) {

DFSInOrder(currentNode->left);

// When processing the current node:

prevNode->right = currentNode;

currentNode->left = prevNode;

listHead = currentNode;

// Traverse the right subtree.

DFSInOrder(currentNode->right);

prevNode = currentNode;

// Helper DFS function to perform in-order traversal.

// Traverse the left subtree first (in-order traversal).

// If prevNode is not null, link it with the current node.

// Move prevNode to the current node before traversing the right subtree.

constructor(val: number, left: Node | null = null, right: Node | null = null) {

listHead->left = prevNode;

prevNode = nullptr;

listHead = nullptr;

DFSInOrder(root);

return listHead;

if (prevNode) {

} else {

Typescript Solution

// Definition for a Node.

left: Node | null;

right: Node | null;

this.val = val;

this.left = left;

this.right = right;

val: number;

class Node {

if (previous != null) {

head = node;

previous = node;

} else {

private Node head;

Java Solution

class Solution {

self.left = left

proceeding with the traversal to the right.

(DFS) in-order traversal.

BST is a tree data structure where each node has at most two children, for which the left child is less than the current node, and the

element, and the list is connected end-to-end.

The conversion should maintain the following conditions:

 Each node's right pointer should reference its successor in the list. The list should be circular, with the last element pointing to the first as its successor, and the first element pointing to the last as

By performing an in-order traversal, we can visit each node in the BST in sorted order, and then adjust their left and right pointers on the fly to link them as if they were in a doubly-linked list. The following steps illustrate the approach:

nodes in a doubly-linked fashion.

The resultant structure needs to be returned with a pointer to its smallest element, effectively the head of the list. The solution to transforming a BST into a sorted doubly-linked list lies in the properties of the BST. The in-order traversal of a BST yields the nodes in ascending order, which is exactly what we want for our sorted doubly-linked list.

Each node's left pointer should reference its predecessor in the list.

right child is greater. A Circular Doubly-Linked List is a series of elements in which each element points to both its previous and next

1. Recursively perform an in-order traversal (left node, current node, right node) of the BST.

3. For the first node, which does not have a prev, we set it as the head of our linked list.

node, and then going right (dfs(root.right)). This is the essence of in-order traversal.

• If prev is not None, we set root.left to prev and prev.right to root, creating reciprocal links between prev (the

5. Update Previous Node: After linking the current node to its predecessor, we update prev to point to the current node before

6. Closing the Circular List: Once the DFS is done processing all nodes, the prev will be pointing to the last (largest) element in the

BST. At this point, we connect the previright to head and head. left to prev to close the loop and make the doubly-linked list

In summary, the complexity of the algorithm lies in the recursive traversal of the BST and the efficient updating of the node's

In this BST, the traversal order for converting it into a sorted doubly-linked list using in-order traversal would be: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$.

head: initially None, will point to the node with the smallest value (in this example, it will be 1).

• prev: initially None, will keep track of the last node processed to link it with the current node.

• Define and begin the recursive in-order traversal with the root (the node with the value 4).

• Process left subtree (2). This takes us further to the left to node 1, which is the start of the list.

• Return to node 2 and link it with prev (which is 1). Set 1. right to 2 and 2.left to 1.

• Close the circular list by linking 5. right to head (which is pointing to 1) and 1. left to 5.

The result is a circular doubly-linked list with the following connections (illustrated as pointers):

The circular list starts at 1 and ends at 5, with 5 pointing back to 1 and 1 pointing back to 5.

def treeToDoublyList(self, root: 'Optional[Node]') -> 'Optional[Node]':

Recursive case: traverse the left subtree.

Recursive case: traverse the right subtree.

Helper function to perform the in-order traversal of the tree.

Base case: if the node is None, return to the previous call.

Link the previous node with the current node from the left.

Link the current node with the previous node from the right.

Initialize the head and previous pointer used during the in-order traversal.

Perform the in-order traversal to transform the tree to a doubly linked list.

Connect the last node visited (previous) with the head of the list to make it circular.

// This 'previous' node will help in keeping track of the previously processed node in the inorder traversal

Mark the current node as the previous one before the next call.

If this node is the leftmost node, it will be the head of the doubly linked list.

• Return to 4, link 3. right to 4 and 4. left to 3. prev set to 4.

def __init__(self, value, left=None, right=None):

• At node 1, there is no prev, so set head to 1. Since there's no previous node, we can't link it yet. prev becomes 1.

• There is no left subtree to the node 3, so we just set previright which is 2 right to 3 and 3 left to 2 prev is updated to 3.

• Finally, process right subtree (5). Since there's no left child for the node with 5, we directly link 4. right to 5 and 5. left to 4. prev

• If prev is None, it means we are processing the very first node (smallest element) of the BST which should become the head

Binary Search Tree

Linked List

Binary Tree

Leetcode Link

2. As we visit each node during the traversal, we connect it with the previously visited node (prev) to simulate the linked list's Make prev. right point to the current node (root), and make the current node's left point to prev. This process connects the 4. After we have visited all the nodes, we connect the last visited node with the head to make the list circular. By carefully updating the pointers as we traverse the BST, we manage to rearrange the original tree structure into a sorted circular

The implementation of the solution follows the intuition discussed and is executed in several steps, utilizing a depth-first search 1. Initialization: Before starting the DFS, we declare two variables head and prev as None. The head will eventually point to the smallest element of the BST, which will become the head of the doubly-linked list. The prev is used to keep track of the last 2. Using DFS for In-order Traversal: The function dfs is defined nested within the treeToDoublyList function. It takes a single 3. Recursive Traversal: The dfs function is designed to be called recursively, going left (dfs(root.left)), processing the current

4. Connecting Nodes: Upon visiting each node root during the traversal, we execute logic to redefine its left and right pointers:

13 /** 14 * Converts a binary search tree to a sorted circular doubly-linked list. * @param {Node | null} root - The root node of the binary search tree. * @returns {Node | null} function treeToDoublyList(root: Node | null): Node | null { if (!root) return root; 20 21 22 let previous: Node | null = null; let head: Node | null = null; 24 25 /** * Depth-first search (In-order traversal) to iterate over the tree and create the doubly linked list. * @param {Node | null} node - The current node being visited.

if (!node) return;

if (previous) {

head = node;

previous = node;

inOrderTraversal(root);

if (head && previous) {

return head;

Time Complexity:

} else {

// Traverse the left subtree

inOrderTraversal(node.left);

previous.right = node;

node.left = previous;

// Traverse the right subtree

inOrderTraversal(node.right);

// Start the in-order traversal

previous.right = head;

head.left = previous;

function inOrderTraversal(node: Node | null): void {

// Link the current node with the previous node

// Set the head if this is the leftmost node

// Move the 'previous' pointer to the current node

// Connect the head and tail to make the list circular

Each node in the tree is visited exactly once during the traversal.

The time complexity of the code is determined by the in-order traversal of the binary search tree.

• The operations performed for each node are constant time operations, including updating the previous (prev) and current node's

Therefore, if there are n nodes in the tree, the in-order traversal will take O(n) time. As a result, the overall time complexity of the

62 **Time and Space Complexity** The given Python code is designed to convert a binary search tree into a sorted, circular doubly-linked list in-place. We analyze the time complexity and space complexity of the provided code as follows:

Space Complexity: The space complexity of the code is determined by: • The implicit stack space used during the recursive in-order traversal (due to dfs calls). No additional data structures are used that are proportional to the size of the input. Hence, the maximum space is taken by the recursion stack, which in the worst case (a completely unbalanced tree) could have n

pointers.

code is O(n).

recursion stack would be O(log(n)). Worst case (skewed tree): 0(n)

recursive calls on the stack. However, in the best case (a completely balanced tree), the height would be log(n), and therefore the In accordance with this, the space complexity of the code is: Average case (balanced tree): 0(log(n))