LeetCode Explore Problems Mock Contest Articles Discuss Store

Articles > 114. Flatten Binary Tree to Linked List

Previous Next

114. Flatten Binary Tree to Linked List

Feb. 16, 2020 | 23.8K views

Average Rating: 4.88 (60 votes)

Given a binary tree, flatten it to a linked list in-place.

For example, given the following tree:

```
    1
   / \
  2   5
 / \   \
3   4   6
```

The flattened tree should look like:

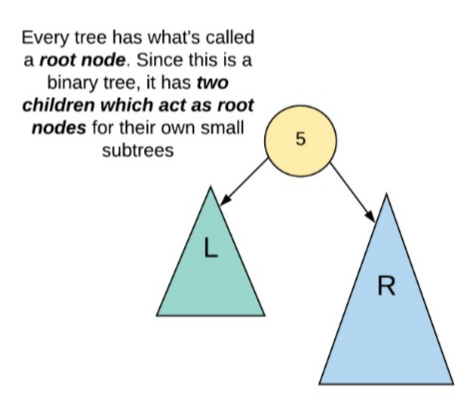
```
1
 \
  2
   \
    3
     \
      4
       \
        5
         \
          6
```

Solution

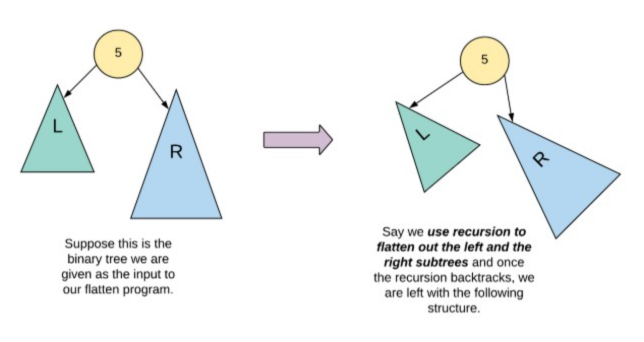
Approach 1: Recursion

Intuition

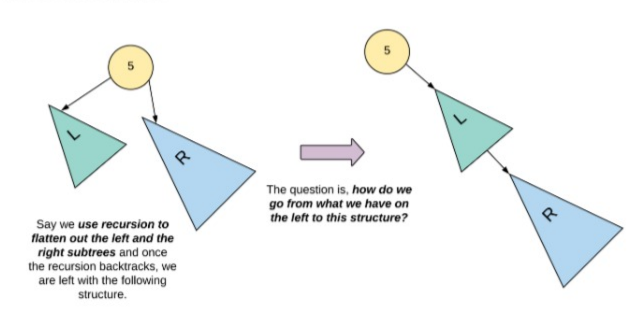
A common strategy for tree modification problems is recursion. A tree is a recursive structure. Every node gets to be a root node of some tree and that tree further has a bunch of smaller subtrees each with their own root nodes. So, when it comes to problems where the structure of the tree has to be modified or we have to traverse the tree in general, recursion is one of the top approaches that comes to mind simply because it's easy enough to code up and also is very intuitive to understand. Let's quickly look at a binary tree structure and then we will talk about how we can solve this problem using a recursive strategy.



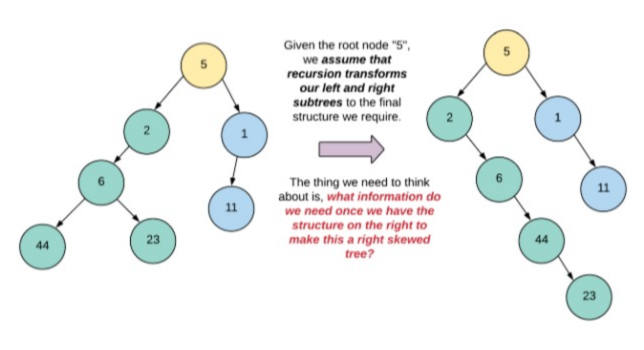
We haven't drawn the entire tree in the above image so as to build that intuition for that recursive solution. The main idea behind a recursive solution is that we use the solutions for subproblems to solve an uber level problem. In the case of a tree, the subtrees are essentially our subproblems. So, a recursive solution for this problem is essentially based on the idea that assuming we have already transformed the left and the right halves of a given root node, how do we establish or modify the necessary connections so that we get a right skewed tree overall. Let's look at what this means diagrammatically to have a better understanding.



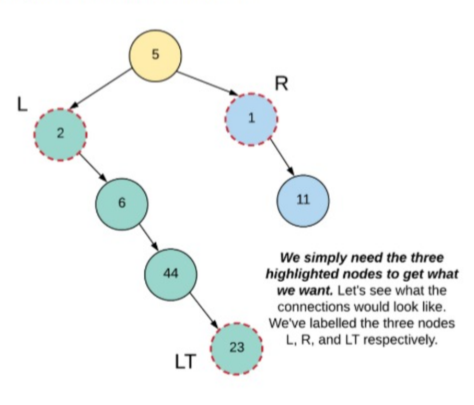
In the above figure, we simply showcase the root node of a tree and its left and right subtrees. A great way to think about recursion here is that we "suppose" that recursion does all the hard work for us and flattens out the left and the right subtrees as shown in the figure. What is it that we have to do then to get our final result? We need a right skewed tree, right? Well, we simply have to shuffle around some pointers to get our final result as shown below.

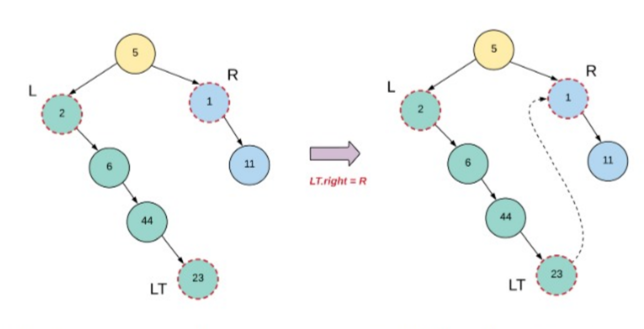


Let's dive a bit deeper and take a look at an exact tree now to see what exact connections we'll need to establish exactly for this work.

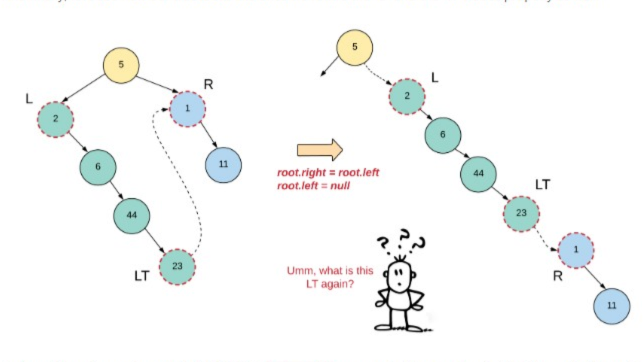


The figure shown below essentially highlights the exact set of nodes that are required for re-wiring the tree to our final right skewed tree. We've marked them "L" for left, "R" for right, and LT for "left tail". We'll get to the reason as to why we call that third node "left tail", later.





And finally, let's see how our tree looks like once we rewire the "L" and the "R" nodes properly as well.



Yeah, we haven't exactly explained what this `left tail` means. So, if you go back a few figures to the point where we had the left and the right subtrees all flattened out and we hadn't done any pointer manipulation yet, you'll notice that each subtree actually looks like a `Linked List`. Every linked list has a head node and in this case, we also need the `tail` node. Once recursion does the hard work for us and flattens out the subtrees, we will essentially get two linked lists and we need the tail end of the left one to attach it to the right one. Let's see what all information we will need in our recursive function at a given node.

- _node_ = The current node
- _leftChild_ = the left child of our current node
- _rightChild_ = the right child of our current node
- _leftTail_ = The tail node of the flattened out left subtree
- _rightTail_ = The tail node of the fully formed tree rooted at _node_. This information is needed by the parent recursive calls since the tree rooted at the current node can be some other's node's left subtree or right subtree.

We have all the information available with us except the tail nodes. That's something that our recursion function will have to return. So, a recursion call for a given `node` will return the tail node of the flattened out tree. In our example, we will return the node `11` as the tail end of our final flattened out tree.

Algorithm

1. We'll have a separate function for flattening out the tree since the main function provided in the problem isn't supposed to return anything and our algorithm will return the `tail` node of the flattened out tree.
2. For a given `node`, we will recursively flatten out the left and the right subtrees and store their corresponding tail nodes in `leftTail` and `rightTail` respectively.
3. Next, we will make the following connections (only if there is a left child for the current node, else the leftTail would be null)

```
leftTail.right = node.right
node.right = node.left
node.left = None
```

4. Next we have to return the tail of the final, flattened out tree rooted at `node`. So, if the `node` has a right child, then we will return the `rightTail`, else, we'll return the `leftTail`.

Java Python Copy

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:

    def flattenTree(self, node):

        # Handle the null scenario
        if not node:
            return None

        # For a leaf node, we simply return the
        # node as is.
        if not node.left and not node.right:
            return node

        # Recursively flatten the left subtree
        leftTail = self.flattenTree(node.left)

        # Recursively flatten the right subtree
        rightTail = self.flattenTree(node.right)

        # If there was a left subtree, we shuffle the connections
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each node of the tree exactly once.
- Space Complexity: $O(N)$ which is occupied by the recursion stack. The problem statement doesn't mention anything about the tree being balanced or not and hence, the tree could be e.g. left skewed and in that case the longest branch (and hence the number of nodes in the recursion stack) would be N.

Approach 2: Iterative Solution using Stack

Intuition

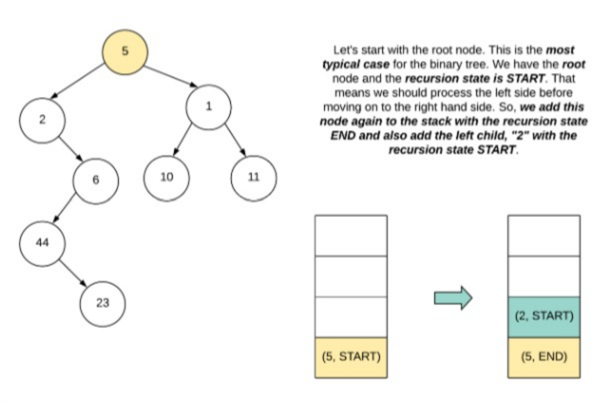
This approach is exactly the same as the previous one except the implementation. In the previous approach we rely on the system stack for our recursion's space requirements. However, as we all know, that stack is limited and for extremely long trees, it might not be feasible to use the system stack. So, we need to use our own stack that will be allocated memory on the heap and will be able to handle much larger sized trees easily.

Algorithm

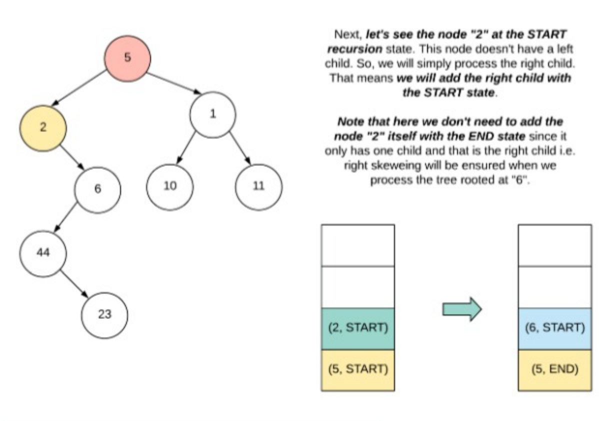
1. So the implementation for this algorithm using a custom stack is a tad bit tricky because now, we have to go one step further and understand how the different states of recursion will unfold and more importantly, we'll have to see how and when to add different nodes to our stack and when to pop them out for good. The basic idea however still follows along the same lines as the previous approach.
2. We will initialize a `stack` which will contain a tuple. The first entry in the tuple will represent our node. The second entry will represent the recursion state that the node is in. We have two different recursion states namely `START` and `END`.
   - START basically means that we haven't started processing the node yet and so, we will try and process its left side if one exists. If not, we will process its right side.
   - When the node is in an END state, that implies we are done processing one side (subtree) of it. If we did process the left subtree of the node, then we need to re-wire the connections so as to make this a right skewed tree. Also, in the END state, we add the right child of the current node to the stack.

   It's important to understand the different cases around these recursion states based on different kinds of (sub)trees.

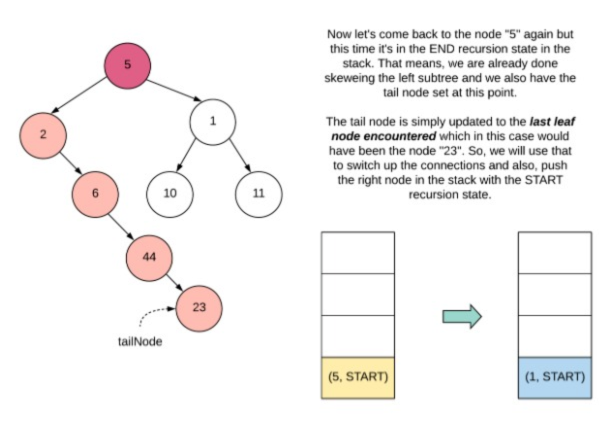
`Case 1: START recursion state, node has a left child`

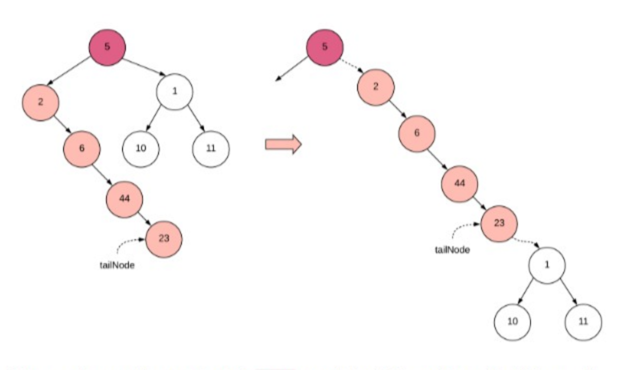


`Case 2: START recursion state, node has NO left child`



`Case 3: END recursion state, node has a right child`





3. If the recursion state of a popped node is `START`, we will check if the node has a left child or not. If it does, we will add the node back to the stack with the `END` recursion state and also add the left child with the `START` recursion state. If there is no left child, then we add the right child `only` with the `START` state.
4. If a node popped from the stack is in the `END` state, that implies it must have had a left child and that means we have a valid `tailNode` set up for re-wiring the connections as shown in the previous figure. Once we are done re-wiring the connections, we push the right child into the stack with the `START` recursion state.
5. Finally, for a popped node that is a leaf node, we will set our `tailNode`.

Java Python Copy

```
import collections
class Solution:

    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """

        # Handle the null scenario
        if not root:
            return None

        START, END = 1, 2

        tailNode = None
        stack = collections.deque([(root, START)])

        while stack:

            currentNode, recursionState = stack.pop()

            # We reached a leaf node. Record this as a tail
            # node and move on.
            if not currentNode.left and not currentNode.right:
                tailNode = currentNode
                continue
```

Complexity Analysis

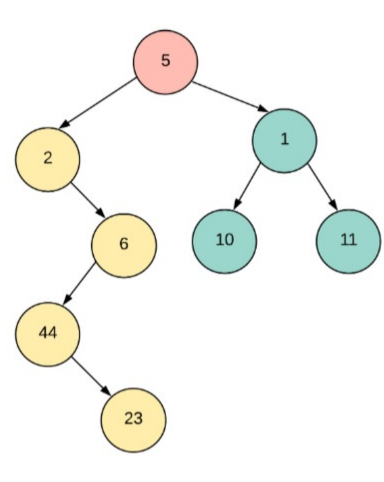
- Time Complexity: $O(N)$ since we process each node of the tree exactly once.
- Space Complexity: $O(N)$ which is occupied by the stack. The problem statement doesn't mention anything about the tree being balanced or not and hence, the tree could be e.g. left skewed and in that case the longest branch (and hence the number of nodes in the recursion stack) would be N.

Approach 3: O(1) Iterative Solution

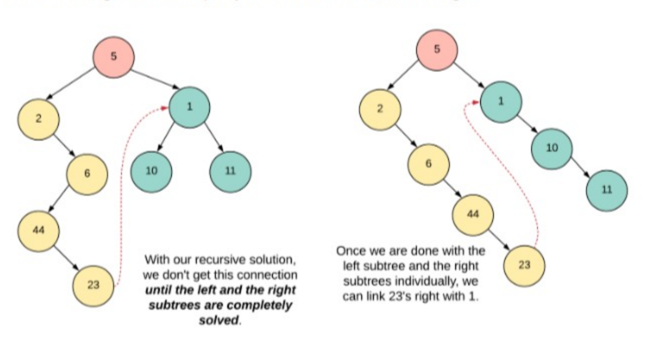
Intuition

We'll get to the intuition for this approach in a bit, but first let's talk about the motivation. For any kind of tree traversal, we always have the easiest of solutions which is based on recursion. Next, we have a custom stack based iterative version of the same solution. Finally, we want a tree traversal that doesn't use any kind of additional space at all. There is a well known tree traversal out there that doesn't use any additional space at all. It's known as `Morris Traversal`. Our solution is based off of the same ideology, but Morris Traversal is not a pre-requisite here.

To understand what's difference between the nodes processing of this approach and basic recursion, let's look at a sample tree.



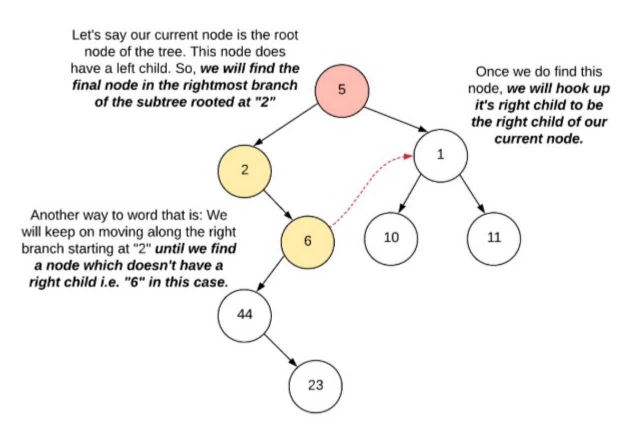
With recursion, we only re-wire the connections for the "current node" once we are already done processing the left and the right subtrees *completely*. Let's see what that looks like in a figure.



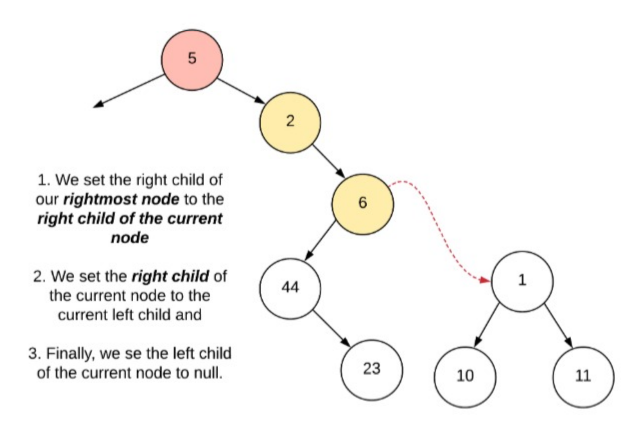
However, the `postponing` of rewiring of connections on the current node until the left subtree is done, is basically what recursion is. Recursion is all about postponing decisions until something else is completed. In order for us to be able to postpone stuff, we need to use the stack. However, in our current approach we want to get rid of the stack altogether. So, we will have to come up with a `greedy` way that will be costlier in terms of time, but will be space efficient in achieving the same results.

> For a current node, we will check if it has a left child or not. If it does, we will find the last node in the rightmost branch of the subtree rooted at this left child. Once we find this "rightmost" node, we will hook it up with the right child of the current node.

Let's look at this idea on our current sample tree.



This might not make a lot of sense just yet. But, bear with me and read on. Let's see what connections we need to establish or shuffle once we find that "rightmost node". We are highlighting "rightmost" here because technically, even without knowing this approach, the node `23` would have made much more sense here, right? Instead, we are doing some vodoo with the node `6`. God knows why!



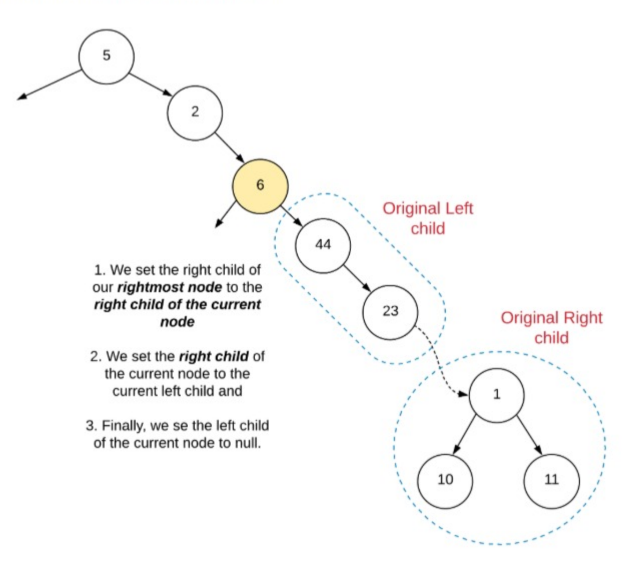
As mentioned in the previous paragraph, this figure would make much more sense if we had just found out the node `23` and set its right child to `1` instead of doing all this with `6`. Why did we do that you might ask? Well, it's an optimization of sorts. To find the actual "rightmost" node of subtree, we might have to potentially traverse most of that subtree. Like in our example. To actually get to the node `23` we would have had to traverse all of the nodes: `2, 6, 44, 23`. Instead, we simply stop at the node 6. We'll see why that also achieves our final purpose. For now, let's move on.

By doing the following operation for every node, we are simply try ing to move stuff to the right hand side one step at a time. The reason we used the node `6` in the above example and not `23` is the very reason we called this approach somewhat `greedy`.

Processing of the node `2` is simple since it doesn't have a left child at all. So we have nothing to do here. Let's come over to the node `6` since this is where things get interesting and start to make sense. We'll again use the same logic as before.

> For a current node, we will check if it has a left child or not. If it does, we will find the last node in the rightmost branch of the subtree rooted at this left child. Once we find this "rightmost" node, we will hook it up with the right child of the current node.

As we can clearly see from the previous figures, the rightmost node here would be `23`. So, let's look at the tree after we are done rewiring the connections.



Now this looks just like the tree after the recursion would have completed on the left subtree and we rewired the connections, right? Exactly!. The reason we stopped at the `first rightmost node with no right child` was because we would eventually end up `rightyfying` all the subtrees through that connection. Even though before we didn't hook up the node `23`, we were able to do it when we arrived at the node `6` here.

Algorithm

1. So basically, this is going to be a super short algorithm and a short-er implementation :)
2. We use a pointer for traversing the nodes of our tree starting from the root. We have a loop that keeps going until the node pointer becomes null which is when we would be done processing the entire tree.
3. For every node we check if it has a left child or not. If it doesn't we simply move on to the right hand side i.e.

```
node = node.right
```

4. If the node does have a left child, we find the first node on the rightmost branch of the left subtree which doesn't have a right child i.e. the almost rightmost node.

```
rightmost = node.left
while rightmost != null:
rightmost = rightmost.right
```

5. Once we find this rightmost node, we rewire the connections as explained in the intuition section.

```
rightmost.right = node.right
node.right = node.left
node.left = null
```

6. And we move on to the right node to continue processing of our tree.

Java Python Copy

```
class Solution:

    def flatten(self, root: TreeNode) -> None:
        """
        Do not return anything, modify root in-place instead.
        """

        # Handle the null scenario
        if not root:
            return None

        node = root
        while node:

            # If the node has a left child
            if node.left:

                # Find the rightmost node
                rightmost = node.left
                while rightmost.right:
                    rightmost = rightmost.right

                # rewire the connections
                rightmost.right = node.right
                node.right = node.left
                node.left = None
```

Complexity Analysis

- Time Complexity: $O(N)$ since we process each node of the tree at most twice. If you think about it, we process the nodes once when we actually run our algorithm on them as the `currentNode`. The second time when we `come across` the nodes is when we are trying to find our `rightmost node`. Sure, this algorithm is `slower` than the previous two approaches but it doesn't use any additional space which is a big win.
- Space Complexity: $O(1)$ boom!.

Rate this article: ★★★★★

Previous Next

Comments: 17 Sort By

Type comment here... (Markdown is supported)

Preview Post

tkblackbelt ★12 April 8, 2020 4:12 AM

Here's mine using a Stack

```
class Solution:
    def flatten(self, root: TreeNode) -> None:
```

Read More

7 Share Reply

trashadewan ★7 February 17, 2020 7:24 AM

@sachinmalhotra1993 Thank you for the detailed explanation :)

7 Share Reply

fsb ★10 April 14, 2020 10:53 AM

Amazing explanation!

3 Share Reply

Kumagai ★22 April 7, 2020 12:18 PM

How about the below for recursive solution?

```
class Solution:
    def flatten(self, root: TreeNode) -> None:
```

Read More

3 Share Reply

karthikkan ★2 February 23, 2020 3:26 AM

In approach 3, Algorithm point 4, it should be rightmost.right!=null

However, this is correct in the actual code implementation

2 Share Reply

vsmourya ★31 April 6, 2020 8:21 PM

Thanks for such a beautiful explanation. I learned a lot. :)

1 Share Reply

lsheng_mel ★167 June 27, 2020 2:16 PM

Awesome article, really well written and articulated.

Only that we don't actually need all those states analysis for stack solution, it can be simply done in a left subtree DFS fashion as following:

Read More

0 Share Reply

rpashuto ★57 June 20, 2020 2:09 PM

A small typo in the article description.
4. If the node does have a left child, we find the first node on the rightmost branch of the left subtree which doesn't have a right child i.e. the almost rightmost node.

Read More

0 Share Reply

kevinwei68 ★1 June 9, 2020 1:56 AM

I don't understand, Why the subtrees are flattened by the recursive function. I think the recursive function just returns the most right bottom node? Hope someone can explain this to me, thx!

0 Share Reply

SHOW 1 REPLY

suhane ★6 May 7, 2020 3:16 PM

Simple Java Solution (100%) using pre order traversal and a temp TreeNode.

class Solution {
TreeNode root;
boolean skipFirst = true;

Read More

0 Share Reply

1 2