

## Topic Introduction: Transforming Binary Trees

Today, let's dig into a trio of classic interview problems: **transforming binary trees into new shapes or forms**. You've probably worked with traversals before — preorder, inorder, postorder — but these problems take it a step further. Instead of just visiting nodes in some order, you'll *restructure* the tree as you go.

### What's the big idea?

At the heart of these challenges is the **tree traversal** pattern. This means visiting every node in a binary tree in some systematic way, and often, making changes as you visit each node.

You may use recursion, a stack, or even Morris traversal (threaded trees) for efficient solutions.

### Why does this matter for interviews?

- Tree transformations show how well you understand recursion, references/pointers, and data structure manipulation.
- They force you to think about the *order* in which you process nodes — a crucial interview skill.
- These problems often require in-place modification, testing your ability to keep track of changing relationships.

### Quick Example (not one of today's problems):

Imagine you want to mirror a binary tree (flip it left-to-right).

You'd traverse the tree, and at each node, swap its left and right children.

Simple, but the key is visiting every node — and doing the work at just the right moment.

## Why These Problems?

### Today's group:

- [Flatten Binary Tree to Linked List](#)
- [Convert BST to Greater Tree](#)
- [Binary Tree to Doubly Linked List](#)

All three involve **transforming a tree by traversing it in a specific order, and updating node relationships or values along the way**.

They differ in *how* you traverse and *what* you do at each step, but the core pattern is the same:

**"Visit every node in a specific order, and make smart changes as you go."**

Let's get started!

## Problem 1: Flatten Binary Tree to Linked List

### Problem Link:

[Flatten Binary Tree to Linked List](#)

## In Your Own Words

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

## PrepLetter: Flatten Binary Tree to Linked List and similar

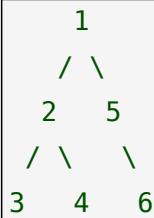
The linked list should use the tree's right pointers, and the order should follow a *preorder traversal* (root, left, right).

**So:**

- Each node's left child should become **None**.
- Each node's right child should point to the next node in preorder.

### Example

**Input:**



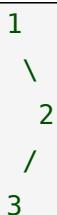
**Output:**

Flattened as a right-skewed linked list:



### Another Test Case

Try this one on paper:



Expected flattened: 1 - 2 - 3 (all via **.right** pointers).

### How to Approach

### Brute Force

- Perform a preorder traversal, collect all nodes in a list.
- Relink them in that order: set each node's `.right` to the next, and `.left` to `None`.

**Time:**  $O(N)$

**Space:**  $O(N)$ , for the extra list.

### Optimal Approach

Let's do it *in-place* with recursion!

The trick is to process the right subtree *after* the left, and reconnect nodes as you back up.

#### Pattern:

- Recursively flatten the right subtree.
- Recursively flatten the left subtree.
- Move the flattened left subtree to the right, and attach the previously flattened right subtree to its end.

#### Step-by-step:

- If the current node is `None`, return.
- Recursively flatten the right subtree.
- Recursively flatten the left subtree.
- Store the original right subtree.
- Move the left subtree to the right.
- Traverse to the end of the new right subtree, and attach the original right subtree.
- Set the left child to `None`.

### Python Solution

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def flatten(self, root):
        """
        Flattens the binary tree to a linked list in-place.
        """
        if not root:
            return
```

```
# Recursively flatten left and right subtrees
self.flatten(root.left)
self.flatten(root.right)

# Store the original right subtree
right_subtree = root.right

# Move the left subtree to the right
root.right = root.left
root.left = None

# Find the end of the new right subtree
current = root
while current.right:
    current = current.right

# Attach the original right subtree
current.right = right_subtree
```

**Time Complexity:**  $O(N)$  — Each node is visited once.

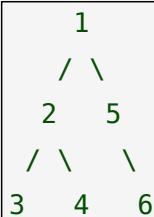
**Space Complexity:**  $O(H)$  —  $H$  = tree height (recursion stack).

### Code Walkthrough

- For each node, recursively flatten its subtrees.
- After flattening, you move the left subtree to the right side.
- Then, walk down to the end of this new right chain — and attach the old right subtree.
- The process ensures the preorder sequence is respected.

### Example Trace

For the example tree:



- Flatten left of 1 (subtree rooted at 2).
  - Flatten left of 2 (subtree 3): 3 is a leaf, nothing happens.
  - Flatten right of 2 (subtree 4): 4 is a leaf, nothing happens.
  - Move left (3) to right, attach right (4) at the end.
- Flatten right of 1 (subtree 5).

## PrepLetter: Flatten Binary Tree to Linked List and similar

- Flatten right of 5 (subtree 6): 6 is a leaf, nothing happens.
- 5 has no left, nothing to move.
- Move left (2) to right of 1, and attach right subtree (5) at the end of the new right chain.

### Try This On Your Own

Test Input:

```
1
/
2
 \
  3
```

After flattening, what does the right-skewed list look like?

**Pause here and try coding it before continuing.**

## Problem 2: Convert BST to Greater Tree

**Problem Link:**

[Convert BST to Greater Tree](#)

### Problem Restated

Given a binary search tree (BST), transform it so every node's value becomes the sum of all values *greater than or equal* to it in the original tree.

This must be done in-place.

### Example

**Input:**

```
5
/
2  13
```

**Output:**

```
18
/
13  13
```

- Node 5 becomes  $5+13=18$ .
- Node 2 becomes  $2+5+13=20$  (but actually,  $2+5+13$ , but since 2 is the smallest, its final value is original  $2+5+13=20$ ).
- Actually, let's clarify with the standard example:

### Official expected output:

```
18
 / \
20   13
```

- Node 5:  $5+13=18$
- Node 2:  $2+5+13=20$
- Node 13: 13

### Why is this similar?

Again, you're traversing the tree and updating nodes as you go.

But here, the order of traversal is crucial:

- For BSTs, the right subtree contains *greater* values.
- So, we need to process the largest nodes first, working from right to left (reverse-inorder traversal).

### Brute Force

- For every node, traverse the entire tree and sum all nodes greater than or equal to its value.
- This is  $O(N^2)$  and not efficient.

### Optimal Approach

#### Pattern:

- Use *reverse inorder* traversal: right, root, left.
- Keep a running **total** sum of all previously visited nodes (which are larger).
- For each node:
  - Add its value to **total**.
  - Update the node's value to **total**.

#### Step-by-step:

- Start with a variable **total = 0**.
- Traverse right subtree (greater values).
- At each node:
  - Add its value to **total**.
  - Update node's value to **total**.
- Traverse left subtree.

### Pseudocode

```
Initialize total = 0
```

```
function traverse(node):
```

```
if node is null:  
    return  
traverse(node.right)  
total += node.val  
node.val = total  
traverse(node.left)
```

```
Call traverse(root)
```

### Example Walkthrough

Input:

```
5  
/ \  
2 13
```

- Visit 13: total = 0+13=13, node.val = 13
- Visit 5: total = 13+5=18, node.val = 18
- Visit 2: total = 18+2=20, node.val = 20

Result:

```
18  
/ \  
20 13
```

### Try This Test Case

Input:

```
2  
/ \  
0 3  
 \  
 1
```

What will this look like as a greater tree?

### Complexity

- **Time:** O(N) — Every node is visited once.
- **Space:** O(H) — H = tree height for recursion stack.

**Take a moment and try implementing this yourself!**

### Problem 3: Binary Tree to Doubly Linked List

#### Problem Link:

[Binary Tree to Doubly Linked List](#)

#### Problem Restated

Given a BST, convert it to a *sorted* circular doubly linked list in-place.

The left and right pointers become prev and next.

The smallest element is the head.

The largest element's right (next) should point to the head; the head's left (prev) should point to the largest node.

#### How is this similar or different?

- Like Problem 2, you need to process nodes in *sorted order* (inorder traversal).
- Unlike earlier, you must connect nodes as you go, and handle *circular* links at the start and end.

#### Brute Force

- Traverse the tree, collect nodes in a list.
  - Set up the prev/next links in the list order.
  - Make it circular.
- **Time:**  $O(N)$
- **Space:**  $O(N)$  — extra list.

#### Optimal Approach

Let's do this *in-place*, using recursion.

#### Pattern:

- Use inorder traversal to process nodes in sorted order.
- Keep track of the previous node visited (**prev**).
- As you visit each node:
  - Set **prev.right = current** (prev's next).
  - Set **current.left = prev** (current's prev).
- At the end, connect head and tail to make it circular.

#### Pseudocode

```
If root is null, return null
```

```
Initialize:  
    prev = None  
    head = None  
  
function inorder(node):  
    if node is null:  
        return  
    inorder(node.left)  
  
    if prev is not None:  
        prev.right = node  
        node.left = prev  
    else:  
        head = node  
  
    prev = node  
    inorder(node.right)  
  
Call inorder(root)  
  
After traversal:  
    head.left = prev  
    prev.right = head  
  
Return head
```

### Example Walkthrough

Input:

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

Inorder: 1, 2, 3, 4, 5

Linked as: 1 <-> 2 <-> 3 <-> 4 <-> 5, then 5.next = 1, 1.prev = 5.

### Try This Test Case

Input:

```
2  
/ \
```

1	3
---	---

- After conversion, should be 1 <-> 2 <-> 3, circular.

### Challenge:

- Implement the function yourself, test with small trees.
- Think about edge cases: empty tree, single node.

### Complexity

- **Time:**  $O(N)$
- **Space:**  $O(H)$  for recursion stack.

### Hint:

Notice how all three problems rely on the *order* in which you traverse, and making the right changes as you visit nodes.

## Summary and Next Steps

### What did we learn?

- These three problems are all about transforming binary trees by traversing them in a specific order.
- The **traversal order** (preorder, inorder, reverse-inorder) is everything!
  - Flatten: preorder
  - Greater Tree: reverse-inorder
  - Doubly Linked List: inorder
- When transforming trees, keep track of your *current* and *previous* nodes carefully.
- *In-place* solutions are often possible with recursion and a few pointers.
- Common mistakes:
  - Losing track of left/right pointers before re-linking.
  - Forgetting to set pointers to **None** where required.
  - Not handling edge cases (empty tree, single node).

## Action List

- Solve all three problems on your own — even the one with code provided.
- For Problem 2 and 3, try both recursive and iterative approaches.
- Think about how you'd handle these if you couldn't use recursion (try with a stack).
- Explore other problems that use tree traversals for transformation.
- Compare your code to official solutions and discuss with peers.
- If you get stuck, revisit the traversal pattern and draw diagrams — it helps!

Keep practicing! Every tree has a story to tell — and with enough practice, you'll be fluent in all their languages.

Happy coding!