Binary Tree Upside Down

hiimdaosui Follow
Jul 16, 2018 · 7 min read

> *Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.*
>
> ***Example:***

Input: `[1,2,3,4,5]`

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

Output: `return the root of the binary tree [4,5,2,#,#,3,1]`

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

This is a little bit complicated problem at my first glance. But still, let's break down the big problem and have a list of things to do, then let's hope this question could be easier.

The goal is to make the tree upside down. How do we accomplish this more specifically? Note that if a node has left child, its left child will become the new root and the current root's right child and the current root will become the new root's left and right children, respectively. So let's make a list:

1. If the current root has left child, make the left child right point to the current root.

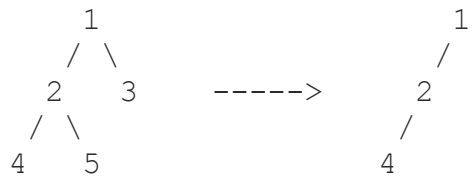
2. If the current root has left child, make the left child left point to the current root.

3. Repeat the operations above until the current root does not have left child.

But how do we actually implement this idea? Note that in the example above,

at the very beginning, we will make node 2 left point to 1 and right point to 3, but then the subtrees of node 2 are lost, so its really hard to keep track of those information.

So let's try to think a little bit differently. First, if we ignore the existence of right subtrees, what could we get in this scenario? In the example above,



What does this new tree look like? It's literally equivalent to a singly linked list, isn't it? If node 1 is the root of this binary tree, then it is also the head of this list, as well. And we can also consider the left pointer of a node the next pointer in a list node. Hence, we can easily transform this problem into linked list reversal problem, and the only additional issue is to connect correctly for the right subtrees.

In terms of connecting right subtrees, let's make some observations. Again, in that example, there are two right subtree connections we need to make. First, we need to left point node 2 to the right subtree of 1, which is node 3. `2->3` . The second one is that we need to point node 4 to node 5, which is the right child of 2. `4->5` . So the only thing we need to do here, is just to left point the new root to the right child on the previous level. Therefore, we can simply save its information in the last round of iteration and use it in the current round.

- **Iterative Solution**

```
TreeNode* upsideDownBinaryTree(TreeNode* root) {
    if (!root || !root->left) {
        return root;
```

```
        }

        TreeNode* prev = NULL
        TreeNode* cur = root;
        TreeNode* next = NULL;
        TreeNode* lastRight = NULL;

        while (cur) {
            next = cur->left;

            cur->left = lastRight;
            lastRight = cur->right;

            cur->right = prev;

            prev = cur;
            cur = next;
        }

        return prev;
    }
```

First, we will do the sanity check. Then, we will just simulate the algorithm of singly linked list reversal along the left subtree track. We can do such analogy,

Variables	Binary Tree	Singly Linked List
prev	parent node	previous node
cur	current node	current node
next	left child node	next node
lastRight	last right child node	

Thus, we can summarize the algorithms in a few steps that are easier to understand:

1. Store the next root, the left child of the current node.
2. Flip. Left point the current node to the right subtree on the last level, and right point the current node to its parent.
3. Update the previous, the current and the last right node for next iteration. Note that the current's right node is supposed to be the last right node in the next round, but

we modified it in step 2, so we can store that value in advance or update it between the two flip operations (as it shows in the code).

Let's go through the example above:

```
Suppose that we are given a binary tree:

    1
   / \
  2   3
 / \
4   5

Initialization:     prev = NULL, cur = 1, next = NULL, lastRight =
NULL

1st Iteration:      store 2 in next
                    let 1 left point to NULL (lastRight)
                    store 3 in lastRight
                    let 1 right point to NULL (prev)
                    move prev to 1 (cur)
                    move cur to 2 (next)

Binary Tree:

    1

  2   3
 / \
4   5

Data Structure:     prev = 1, cur = 2, next = 2, lastRight = 3

2nd Iteration:      store 4 in next
                    let 2 left point to 3 (lastRight)
                    store 5 in lastRight
                    let 2 right point to 1 (prev)
                    move prev to 2 (cur)
                    move cur to 4 (next)

Binary Tree:

    1
   /
  2 - 3

4   5

Data Structure:     prev = 2, cur = 4, next = 4, lastRight = 5
```

```
2nd Iteration:      store NULL in next
                    let 4 left point to 5 (lastRight)
                    store NULL in lastRight
                    let 4 right point to 2 (prev)
                    move prev to 4 (cur)
                    move cur to NULL (next)

Binary Tree:

    1
   /
  2 - 3
 /
4 - 5

Data Structure:     prev = 4, cur = NULL, next = NULL, lastRight =
NULL

return prev = 4
```

Time Complexity: O(n) — *n* represents the length of the path stretching from root all the way down to its left children.

Space Complexity: O(1) — The only main data structure we use in this implementation are those four pointers, which is definitely constant space complexity.

- **Recursive Solution**

We already compared this problem to the simple linked list reversal problem, so it's also supposed to be viable to be solved by recursion. Recall the linked list problem, we first recursively call the sub-problem and try to get the final head of the reversed result list. And then in current call, we make the current node last one in the list. Similarly, we can definitely use the same strategy. First, we don't think too much, which means that we can directly call the sub-problem and try to get the eventual new root from those recursive calls. Next, we make the current root as the right child of its left child, and the current root's right child as the left child of the root's left child. The wording is a bit confusing, but the logic is almost the same as the list problem.

Now, let's analyze the recursive rules:

1. Base case: When the input root is empty or it does not have left child.

2. Recursive Rule: 1) Retrieve the final new root from recursive calls. 2) Make the current node and its right child be the children of the current node's left child. More specifically, Left point the left child to the current root's right child and right point the left child to the current node.

```
TreeNode* upsideDownBinaryTree(TreeNode* root) {
    if (!root || !root->left) {
        return root;
    }

    TreeNode* newRoot = upsideDownBinaryTree(root->left);
    root->left->left = root->right;
    root->left->right = root;
    root->left = NULL;
    root->right = NULL;

    return newRoot;
}
```

We can also compare this recursive solution to the linked list reversal problem's one. The base case is that when the input root is empty or it does not have left child, we can directly return the root. Note that the empty case here is also a corner case instead of the base case since if the tree is not empty, we will never reach the case that the recursive call gets a `NULL` pointer. This is also because we need to return the final new root to the former function calls, so we have to return the last node but not the empty one. If the current root has no left child, it will be the eventual new root after flip, so we return that one.

For the recursive rule, note that in the linked list problem, we directly go fetch the final result in the subsequent sub-list. Here, we can definitely apply the same strategy to fetch the final result at first, and then complete the task in our current function. Recall that in that list problem, in each function, we set the current node to the last node in the list. Now, in a binary tree, we need to set the current node and its right child to the last level in the tree. And everything is almost the same. We let the left child point to the current root and its right child, and then set the children of the current root to empty.

```
╔═══════════╦═══════════════════════════════╦═══════════════════════════════╗
║   Logic   ║       Flip Binary Tree        ║    Reverse Singly Linked List ║
╠═══════════╬═══════════════════════════════╬═══════════════════════════════╣
```

Base Case	Root is empty	Head is empty
	Root has no left child	Head has no next node
Recursive Call	Call root.left to get the final new root	Call head.next to get the final ne
Current Call	Let root.left.right point to root	Let head.next.next point to head
	Let root.left.left point to root.right	
	Let root.left point to NULL	Let head.next point to NULL
	Let root.right point to NULL	
Return Value	The final new root	The final new head

table.txt hosted with ♡ by GitHub view raw

Comparison Between Flip Binary Tree And Reverse Singly Linked List Problem

It's time to go through the example:

```
Suppose that we are given a binary tree:

    1
   / \
  2   3
 / \
4   5

Binary Tree:

    1
   / \
  2   3
 / \
4   5

In Main Function:               call upsideDownBinaryTree(1)

Binary Tree:

    1
   / \
  2   3
 / \
4   5

In upsideDownBinaryTree(1):     call upsideDownBinaryTree(2)

Binary Tree:
```

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

In upsideDownBinaryTree(2): call upsideDownBinaryTree(4)

Binary Tree:

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

In upsideDownBinaryTree(4): hit base case
return 4

Binary Tree:

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

In upsideDownBinaryTree(2): let 4 left point to 5
let 4 right point to 2
let 2 left point to NULL
let 2 right point to NULL
return 4

Binary Tree:

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

In upsideDownBinaryTree(1): let 2 left point to 3
let 2 right point to 1
let 1 left point to NULL
let 1 right point to NULL
return 4

Binary Tree:

```
    1
   /
  2 - 3
```

```
 /
4 - 5

In Main Function:                get 4
```

Time Complexity: O(n) — The time complexity is the same as the iterative solution since we also need to dive into the deepest node on the most straight left path from root.

Space Complexity: O(n) — With recursion we need to consider the usage of call stacks, so in this problem the space complexity is also O(n) since we need n levels of call stacks for the deepest recursion path.

Programming Leetcode Binary Tree Recursion Linked Lists