Description

145. Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

```
For example:
Given binary tree [1,null,2,3],

   1
    \
     2
    /
   3

return [3,2,1].
```

Idea

Firstly we want to clearify the the post order is left-right-root order. Then I can provide a very simple recursive version but point out that the call stack may overflow if the tree is large. We visited each node only once, so O(n) time, and call stack is O(logn) space, where n is the number of total nodes.

Code:

```
class Solution {

    List<Integer> res;
    public List<Integer> postorderTraversal(TreeNode root) {
        res = new ArrayList<>();
        helper(root);
        return res;
    }

    private void helper(TreeNode root) {
        if (root == null) {
            return;
        }
        helper(root.left);
        helper(root.right);
        res.add(root.val);
    }
}
```

Another recursive approach is divide and conquer. We think left child has all the list of integers ready (leftList), and right child has all the list of integers ready (rightList). We can append leftList + rightList + root.val to get the final result. We visited each node only once, so O(n) time, and call stack is O(logn) space, where n is the number of total nodes.

Code:

```
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    if (root == null) {
        return res;
    }

    List<Integer> leftList = postorderTraversal(root.left);
    List<Integer> rightList = postorderTraversal(root.right);

    res.addAll(leftList);
    res.addAll(rightList);
    res.add(root.val);

    return res;
}
```

Since we talked about the stack overflow, we might want to discuss a non-recursive version. Basically we just use a stack to simulate the call stack. The order is left-right-root, and I can't think of a good way to do it, so can I try a reverse version root-right-left? The root-right-left is actually like a pre-order traversal but the right child comes at firt before the left child. So I can solve this problem by doing a pre order traversal (root-right-left) and then reverse the whole list to get the result. We visited each node only once, so O(n) time, and stack is O(logn) space, where n is the number of total nodes.

Code:

```java
public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    if (root == null) {
        return res;
    }

    Deque<TreeNode> stack = new ArrayDeque<>();
    stack.push(root);

    while (!stack.isEmpty()) {
        TreeNode cur = stack.pop();
        res.add(cur.val);
        if (cur.left != null) {
            stack.push(cur.left);
        }
        if (cur.right != null) {
            stack.push(cur.right);
        }
    }

    Collections.reverse(res);
    return res;
}
```

(The last version was introduced by Qinyuan and I think it is a very elegent solution.) When we visit to a node, we actually need to do three sub problem, 1) visit left child, 2) visit right child, 3) print current node. So for a node we have two operations visit and print. Each time the node is actually visited, we change it to the print state. For example, we have [1,2,3], if we visit root 1, we don't want to print it right now, instead we set its state to print, then next time we see node 1 and it's on print state, we print or add to the result list. Then we handle node 2 and 3 and so on. We visited each node exactly twice, so O(n) time, and call stack is O(logn) space, where n is the number of total nodes.

Code:

```java
class Pair {
    TreeNode node;
    boolean print;
    public Pair(TreeNode node, boolean print) {
        this.node = node;
        this.print = print;
    }
}

public List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> res = new ArrayList<>();
    if (root == null) {
        return res;
    }

    Deque<Pair> stack = new ArrayDeque<>();
    stack.push(new Pair(root, false));

    while (!stack.isEmpty()) {
        Pair cur = stack.pop();
        if (cur.node == null) {
            continue;
        }
        if (cur.print) {
            res.add(cur.node.val);
        } else {
            stack.push(new Pair(cur.node, true));
            stack.push(new Pair(cur.node.right, false));
            stack.push(new Pair(cur.node.left, false));
        }
    }

    return res;
}
```

Summary

- Divid and conquer is a very common method to solve a binary tree question.
- Dividing a big problem into a sequence of sub problems is a very impoartant idea.