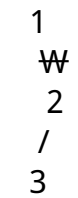
二叉树的前中后遍历

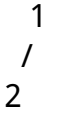
二叉树的前序遍历

给你二叉树的根节点 root ，返回它节点值的 前序 遍历。
示例 1:
输入:



输出: [1,2,3]

示例 2:
输入:



输出: [1,2]

递归

首先我们需要了解什么是二叉树的前序遍历：按照访问根节点——左子树——右子树的方式遍历这棵树，而在访问左子树或者右子树的时候，我们按照同样的方式遍历，直到遍历完整棵树。因此整个遍历过程天然具有递归的性质，我们可以直接用递归函数来模拟这一过程。代码如下：

```
class TreeTraversal {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        preorder(root, res);
        return res;
    }
```

```
    public void preorder(TreeNode root, List<Integer> res) {
        if (root == null) {
            return;
        }
        res.add(root.val);
        preorder(root.left, res);
        preorder(root.right, res);
    }
}
```

非递归

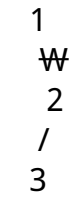
一般面试会进一步问你非递归的实现。区别在于递归的时候隐式地维护了一个栈，而我们在迭代的时候需要显式地将这个栈模拟出来，其余的实现与细节都相同，具体可以参考下面的代码。

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

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode node = root;
        while (!stack.isEmpty() || node != null) {
            while (node != null) {
                res.add(node.val);
                stack.push(node);
                node = node.left;
            }
            node = stack.pop();
            node = node.right;
        }
        return res;
    }
}
```

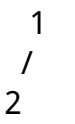
二叉树的中序遍历

给定一个二叉树的根节点 root ，返回它的 中序 遍历。
示例 1:
输入:



输出: [1,3,2]

示例 2:
输入:



输出: [2,1]

递归

二叉树的中序遍历思想：按照访问左子树——根节点——右子树的方式遍历这棵树，而在访问左子树或者右子树的时候我们按照同样的方式遍历，直到遍历完整棵树。因此整个遍历过程天然具有递归的性质，我们可以直接用递归函数来模拟这一过程。

```
class TreeTraversal {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        inorder(root, res);
        return res;
    }

    public void inorder(TreeNode root, List<Integer> res) {
        if (root == null) {
            return;
        }
        inorder(root.left, res);
        res.add(root.val);
```

```
        inorder(root.right, res);
    }
}
```

非递归

一般面试会进一步问你非递归的实现。区别在于递归的时候隐式地维护了一个栈，而我们在迭代的时候需要显式地将这个栈模拟出来，其余的实现与细节都相同，具体可以参考下面的代码。

```
class TreeTraversal {
    public List<Integer> inorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        Stack<TreeNode> stack = new Stack<TreeNode>();
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            root = stack.pop();
            res.add(root.val);
            root = root.right;
        }
        return res;
    }
}
```

二叉树的后序遍历

给定一个二叉树，返回它的 后序 遍历。
示例:
输入:

```
  1
   \
    2
   /
  3
```

输出: [3,2,1]

递归

二叉树的后序遍历思想：按照访问左子树——右子树——根节点的方式遍历这棵树，而在访问左子树或者右子树的时候，我们按照同样的方式遍历，直到遍历完整棵树。因此整个遍历过程天然具有递归的性质，我们可以直接用递归函数来模拟这一过程。

```
class TreeTraversal {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> res = new ArrayList<Integer>();
        postorder(root, res);
        return res;
    }

    public void postorder(TreeNode root, List<Integer> res) {
        if (root == null) {
            return;
        }
        postorder(root.left, res);
        postorder(root.right, res);
        res.add(root.val);
    }
}
```

非递归

一般面试会进一步问你非递归的实现。区别在于递归的时候隐式地维护了一个栈，而我们在迭代的时候需要显式地将这个栈模拟出来，其余的实现与细节都相同，具体可以参考下面的代码。

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

        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode prev = null;
        while (root != null || !stack.isEmpty()) {
            while (root != null) {
                stack.push(root);
                root = root.left;
```

```
            }
            root = stack.pop();
            if (root.right == null || root.right == prev) {
                res.add(root.val);
                prev = root;
                root = null;
            } else {
                stack.push(root);
                root = root.right;
            }
        }
        return res;
    }
}
```