二叉树的遍历

后序遍历（重点）

1、难点

在后序遍历中，要保证左孩子和右孩子都已被访问并且左孩子在右孩子前访问才能访问根结点

2、思路

如果根节点P不存在左孩子和右孩子，则可以直接访问它；或者P存在左孩子或者右孩子，但是其左孩子和右孩子都已被访问过了，则同样可以直接访问该结点。若非上述两种情况，则将P的右孩子和左孩子依次入栈，这样就保证了每次取栈顶元素的时候，左孩子在右孩子前面被访问，左孩子和右孩子都在根结点前面被访问。

```
//后序遍历采用递归的方式
public void postOrder(BinaryTreeNode root) {
    if (root != null) {
        postOrder(root.getLeft());
        postOrder(root.getRight());
        System.out.print(root.getData() + "\t");
    }
}

    // 非递归后序遍历01版本
     public List<Integer> postorderbyStack(TreeNode root) {
    List<Integer> res = new ArrayList<Integer>();
    if(root == null)
        return res;
    Stack<TreeNode> stack = new Stack<TreeNode>();
    stack.push(root);
    while(!stack.isEmpty()){
        TreeNode node = stack.pop();
        if(node.left != null) stack.push(node.left);//和传统先序遍历不一样，先将左结点入栈
        if(node.right != null) stack.push(node.right);//后将右结点入栈
        res.add(0,node.val);                    //逆序添加结点值
    }
    return res;
```

```
}
```

前序遍历

```
//前序遍历递归的方式
public void preOrder(BinaryTreeNode root) {
    if (null != root) {
        System.out.print(root.getData() + "\t");
        preOrder(root.getLeft());
        preOrder(root.getRight());
    }
}
//非递归
public static void preOrderByStack(TreeNode root) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode treeNode = root;
    while (treeNode != null || !stack.isEmpty()) {
        while (treeNode != null) {
            System.out.println(treeNode.val);
            stack.push(treeNode);
            treeNode = treeNode.left;
        }
        if (!stack.isEmpty()) {
            treeNode = stack.pop();
            treeNode = treeNode.right;
        }
    }
}
```

中序遍历

```
//中序遍历采用递归的方式
public void inOrder(BinaryTreeNode root) {
    if (null != root) {
        inOrder(root.getLeft());
```

```
            System.out.print(root.getData() + "\t");
            inOrder(root.getRight());
        }
    }

    /**
     * 非递归中序遍历
     */
    public static void inOrderByStack(TreeNode treeNode) {
        List<Integer> res = new ArrayList<>();
        Stack<TreeNode> stack = new Stack<>();
        while (treeNode != null || !stack.isEmpty()) {
            while (treeNode != null) {
                stack.push(treeNode);
                treeNode = treeNode.left;
            }
            // 打印当前节点
            res.add(treeNode.val);
            // 遍历右子树.
            if (!stack.isEmpty()) {
                treeNode = stack.pop().right;
            }
        }
    }
```

层序遍历

```
//层序遍历
    public void levelOrder(BinaryTreeNode root) {
        BinaryTreeNode temp;
        Queue<BinaryTreeNode> queue = new LinkedList<BinaryTreeNode>();
        queue.offer(root);
        while (!queue.isEmpty()) {
            temp = queue.poll();
            System.out.print(temp.getData() + "\t");
            if (null != temp.getLeft())
                queue.offer(temp.getLeft());
            if (null != temp.getRight()) {
                queue.offer(temp.getRight());
            }
        }
    }
```

```
class BinaryTreeNode {
        private int data;
        private BinaryTreeNode left;
        private BinaryTreeNode right;

        public BinaryTreeNode() {}

        public BinaryTreeNode(int data, BinaryTreeNode left, BinaryTreeNode right) {
            super();
            this.data = data;
            this.left = left;
            this.right = right;
        }

        public int getData() {
            return data;
        }

        public void setData(int data) {
            this.data = data;
        }

        public BinaryTreeNode getLeft() {
            return left;
        }

        public void setLeft(BinaryTreeNode left) {
            this.left = left;
        }

        public BinaryTreeNode getRight() {
            return right;
        }

        public void setRight(BinaryTreeNode right) {
            this.right = right;
        }
    }
```