#树的实战代码

> 树，主要是二叉树，实战题大概分三类：
>
> - Preorder/Inorder/Postorder Traverse相关的题，递归和迭代，略占20%；
> - Levelorder Traverse相关的题(BFS)，递归和迭代，略占10%；
> - 剩下70%的题，都用递归或者分治法来做，Iterative的方法不强求。

二叉树的数据结构

```
class TreeNode {
    int val;
    // Left Child
    TreeNode left;
    // Right Child
    TreeNode right;
}
```

1. 求二叉树的最大深度

也就是根节点到最远叶子节点的距离

```
int maxDepth(TreeNode root) {
    if(root == null) {
        return 0;
    }
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);

    return Math.max(left, right) + 1; //递归的层数+1
}
```

2. 求二叉树的最小深度

也就是根节点到最近叶子节点的距离

```
int getMinDepth(TreeNode node) {
    if (root == null) {
        return 0;
    }
    return getMin(root);
}
int getMin(TreeNode root) {
    if (root == null) {
        return Integer.MAX_VALUE;
    }
    if (root.left == null && root.right == null) {//没有左右儿子了就在该结点处返回1
        return 1;
    }
    return Math.min(getMin(root.left), getMin(root.right)) + 1;
}
```

3. 求二叉树中结点的个数

```
int numOfTreeNodes(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int left = numOfTreeNodes(root.left);
    int right = numOfTreeNodes(root.right);

    return left + right + 1;//算上root自己, +1
}
```

4. 求二叉树中叶子结点的个数

```
int numOfChildNodes(Tree root) {
    if (root == null) {
        return 0;
    }
    if (root.left == null && root.right == null) {
        return 1;
    }

    return numOfChildNodes(root.left) + numOfChildNodes(root.right);//这里就不用+1了
}
```

5. 求二叉树中第k层的结点的个数

k从1开始计数

```
int numOfLevelKNodes(TreeNode root, int k) {
    if (root == null || k < 1) {
        return 0;
    }
    if (k == 1) {
        return 1;
    }

    //k-1, 每次递归往k层递进一步
    int left = numOfLevelKNodes(root.left, k - 1);
    int right = numOfLevelKNodes(root.right, k - 1);

    return left + right;
}
```

6. 判断是否为平衡二叉树

递归检查左右子树的最大深度的差是否超过1

```
boolean isBalancedBinaryTree(TreeNode root) {
    if (root == null) {
        return true;
    }
    return maxDepth(root) != -1;
}

//返回最大深度
int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    //分别递归检查左右子树之间是否平衡和各自是否平衡
    if (left == -1 || right == -1 || Math.abs(left - right) > 1) {
        return -1;
    }

    //optional: 是平衡树的情况下返回最大深度
    return Math.max(left, right) + 1;
}
```

7. 判断是否是完全二叉树

```
boolean isCompleteBinaryTree(TreeNode root) {
    if (root == null) {
        return false; //注意null结点是返回false
    }
    boolean result = true;
    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.add(root);
    boolean hasNoChild = false;// 判断左子树或右子树是否还有孩子

    while (!queue.isEmpty()) {
        TreeNode current = queue.remove();
        if (hasNoChild) {//其中左子树和右子树的其中之一没有孩子了
            if (current.left != null || current.right != null) {// 而左子树和右子树的其中之一则还有孩子
                result = false;//肯定不平衡
                break;
            }
        } else {
            if (current.left != null && current.right != null) {//左右孩子都还有，入队列继续检查
                queue.add(current.left);
                queue.add(current.right);
            } else if (current.left != null && current.right != null) {//存在左孩子，没有右孩子，可能是可能不是
                queue.add(current.left);
                hasNoChild = true;//只有左孩子是非满结点
            } else if (current.left == null && current.right != null) {//没有左孩子，存在右孩子，根据完全二叉树的
定义，肯定不是
                result = false;
                break;
            } else { // 左右孩子都存在，并为非满的状态，需要看后续
                hasNoChild = true;
            }
        }
    }
    return result;
}
```

8. 判断两个二叉树是否相同

```
boolean isSameTree(TreeNode node1, TreeNode node2) {
    //递归的终止条件1
    if (node1 == null && node2 == null) {
        return true;
    } else if (node1 == null || node2 == null) {
        return false;
    }
    //递归的终止条件2
    if (node1.val != node2.val) {
        return false;
    }

    boolean left = isSameTree(node1.left, node2.left);
    boolean right = isSameTree(node2.right, node2.right);

    return left && right;
}
```

9. 判读两个二叉树是否互为镜像

```
boolean isSymmetryTree(TreeNode node1, TreeNode node2) {
    if (node1 == null && node2 == null) {
        return true;
    } else if (node1 == null || node2 == null) {
        return false;
    }
    if (node1.val != node2.val) {
        return false;
    }

    boolean left = isSymmertyTree(node1.left, node2.right);
    boolean right = isSymmetryTree(node1.right, node2.left);

    return left && right;
}
```

10. 翻转二叉树(镜像二叉树)

```
TreeNode reverseTree(TreeNode root) {
    if (root == null) {
        return root;
    }
    //新建两个变量来保存左右子树
    TreeNode left = reverseTree(root.left);
    TreeNode right = reverseTree(root.right);
    //赋值
    root.left = right;
    root.right = left;

    return root;
}
```

11. 求两个二叉树的最低公共祖先

二叉树两个节点的LCA的查找可以使用自顶向下，自底向上和寻找公共路径的方法

> 如果是BST，那么可以直接和root的val比较，方便很多

自顶向下，这个办法会重复遍历结点(查看结点在哪里)

```
TreeNode lowestCommonAncestor(TreeNode root, TreeNode node1, TreeNode node2) {
    if (root.left, node1) {//node1在左子树的情况下
        if (root.right, node2) {//node1在左子树的情况下，node2在右子树
            return root;
        } else {//node1在左子树的情况下，node2也在左子树，继续递归
            lowestCommonAncestor(root.left, node1,node2);
        }
    } else {//node1在右子树的情况下
        if (root.left, node2) {//node1在右子树的情况下，node2在左子树
            return root;
        } else {//node1在右子树的情况下，node2也在右子树，继续递归
            lowestCommonAncestor(root.right, node1, node2);
        }
    }
}
//查找结点是否在当前的二叉树中
boolean hasNode(TreeNode root, TreeNode node) {
    if (root == null || node == null) {
        return false;
    }
    if (root == node) {
        return true;
    }

    return hasNode(root.left, node) || hasNode(root.right, node);
}
```

自底向上，一旦遇到结点等于p或者q，则将其向上传递给它的父结点。父结点会判断它的左右子树是否都包含其中一个结点，如果是，则父结点一定是这两个节点p和q的LCA，传递父结点到root。如果不是，继续向上传递其中的包含结点p或者q的子结点，或者NULL(如果子结点不包含任何一个)。该方法时间复杂度为O(N)。

```
TreeNode lowestCommonAncestor(TreeNode root, TreeNode node1, TreeNode node2) {
    if (root == null) {
        return null;
    }
    //递归时遇到两个nodes之一的停止条件
    if (root == node1 || root == node2) {
        return root;
    }
    TreeNode left = lowestCommonAncestor(root.left, node1, node2);
    TreeNode right = lowestCommonAncestor(root.right, node1, node2);

    if (left != null && right != null) {//node1和node2分别在左右子树
        return root;
    }
    return left != null ? left : right;//node1和node2在都在左子树中，或者二者都不在左子树(在右子树)
}
```

公共路径法，依次得到从根结点到结点p和q的路径，找出它们路径中的最后一个公共结点即是它们的LCA。该方法时间复杂度为O(N)。剑指offer第50题。

12. 二叉树的前序遍历

迭代解法

```
List<TreeNode> preorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();

    TreeNode current = root;
    while (current != null) {
        result.add(current.val);
        if (current.right != null) {//从上到下，依次放入右儿子
            stack.push(current.right);
        }
        current = current.left;
        if (current == null && !stack.isEmpty()) {//这时候左儿子已经放完了
            current = stack.pop();
        }
    }
    return result;
}
```

递归解法

```
List<TreeNode> preorderTraversal(TreeNode root) {
    List<TreeNode> result = new ArrayList<>();
    preorderTraversalHelper(root, result);
    return result;
}
private void preorderTraversalHelper(TreeNode root, List<TreeNode> result) {
    if (root == null) {
        return;
    }
    result.add(root.val);
    preorderTraversalHelper(root.left, result);
    preorderTraversalHelper(root.right, result);
}
```

13. 二叉树的中序遍历

迭代解法

```
List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();

    TreeNode current = root;

    while (current != null || !stack.isEmpty()) {
        while (current != null) {
            stack.add(current);
            current = current.left;//先移动指针去左儿子
        }
        //左儿子没了开始从二叉树的最底层弹，同时考虑每个结点的右儿子
        current = stack.pop();
        result.add(current.val);//这时候可以add了
        current = current.right;
    }
    return result;
}
```

递归解法

```
public List<Integer> inorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    inorderTraversalHelper(root, result);
    return result;
}
private void inorderTraversalHelper(TreeNode root, List<Integer> result) {
    if (root == null) {
        return;
    }
    inorderTraversalHelper(root.left, result);
    result.add(root.val);
    inorderTraversalHelper(root.right, result);
}
```

14. 二叉树的后序遍历

迭代解法

```
List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) {
        return result;
    }
    Stack<TreeNode> stack = new Stack<>();

    stack.push(root);

    while (!stack.isEmpty()) {
        TreeNode current = stack.pop();
        result.add(0, current.val);
        if (current.left != null) {
            stack.push(current.left);
        }
        if (current.right != null) {
            stack.push(current.right);
        }
    }
    return result;
}
```

递归解法

```
List<Integer> postorderTraversal(TreeNode root) {
    List<Integer> result = new ArrayList<Integer>();
    postorderTraversal (root, result);
    return result;
}
private void postorderTraversal(TreeNode root, List<Integer> result) {
    if (root == null) {
        return;
    }
    postorderTraversal (root.left, result);
    postorderTraversal (root.right, result);
    result.add(root.val);
}
```

15. 前序+中序构建二叉树

```
TreeNode buildTree(int[] preorder, int[] inorder) {
    return buildTree(preorder, 0, preorder.length - 1, inorder, 0, inorder.length - 1);
}
private TreeNode buildTree(int[] preorder, int pLeft, int pRight, int[] inorder, int iLeft, int iRight) {
    if (pLeft > pRight || iLeft > iRight) {
        return null;
    }
    int i = 0;
    for (i = iLeft; i <= iRight; i++) {
        if (preorder[pLeft] == inorder[i]) {
            break;
        }
    }
    TreeNode current = new TreeNode(preorder[pLeft]);
    current.left = buildTree(preorder, pLeft + 1, pLeft + i - iLeft, inorder, iLeft, i - 1);
    current.right = buildTree(preorder, pLeft + i - iLeft + 1, pRight, inorder, i + 1, iRight);

    return current;
}
```

16. 中序+后序构建二叉树

```
TreeNode buildTree(int[] inorder, int[] postorder) {
    return buildTree(inorder, 0, inorder.length-1, postorder, 0, postorder.length - 1);
}
private TreeNode buildTree(int[] in, int inStart, int inEnd, int[] post, int postStart, int postEnd) {
    if (inStart > inEnd || postStart > postEnd) {
        return null;
    }
    int rootVal = post[postEnd];
    int rootIndex = 0;
    for (int i = 0; i <= inEnd; i++) {
        if (in[i] == rootVal) {
            rootIndex = i;
            break;
        }
    }
    int len = rootIndex - inStart;
    TreeNode root = new TreeNode(rootVal);
    root.left = buildTree(in, inStart, rootIndex - 1, post, postStart, postStart + len- 1);
    root.right = buildTree(in, rootIndex + 1, inEnd, post, postStart + len, postEnd - 1);

    return root;
}
```

17. 前序+后序构建二叉树-任意一个

前序+后序不能确定唯一的二叉树, 返回任意一个.
迭代

```
/**
* 遍历pre，逐个重建node
* 与pre+in和in+post一样，利用queue存储当前路径
* node = new TreeNode(pre[i])，如果不是左孩子，就加到右孩子上面
* 如果在pre和post中遇到了同样的值，说明当前子树的构建完成，从queue中pop出来
*/
TreeNode constructFromPrePost(int[] pre, int[] post) {
    Deque<TreeNode> queue = new ArrayDeque<>();
    queue.offer(new TreeNode(pre[0]));
    for (int i = 1,j =0; i < pre.length; i++) {
        TreeNode node = new TreeNode(pre[i]);
        while (queue.getLast().val == post[j]) {
            queue.pollLast();
            j++;
        }
        if (queue.getLast().left == null) {
            queue.getLast().left = node;
        } else {
            queue.getLast().right = node;
        }
        queue.offer(node);
    }
    return queue.getFirst();
}
```

递归

```
 TreeNode constructFromPrePost(int[] pre, int[] post) {
    return constructFromPrePost(pre, 0, pre.length-1, post, 0, post.length-1);
}

TreeNode constructFromPrePost(int[] pre, int preL, int preR, int[] post, int postL, int postR) {
    if (preL > preR || postL > postR) {
        return null;
    }
    TreeNode root = new TreeNode(pre[preL]);
    if (preL == preR) {
        return root;
    }

    int index = -1;
    for (int i = postL ; i < postR ; i++) {
        if (pre[preL+1] == post[i]) {
            index = i;
            break;
        }
    }
    if (index == -1) {
        return root;
    }

    root.left = constructFromPrePost(pre, preL+1, preL+1+(index-postL), post, postL, index);
    root.right = constructFromPrePost(pre, preL+1+(index-postL)+1, preR, post, index+1, postR);

    return root;
}
```

18. 二叉树的层序遍历

迭代

```
List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    if (root == null) {
        return result;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);

    while (!queue.isEmpty()) {
        int size = queue.size();// 每一层的元素个数
        List<Integer> level = new ArrayList();
        while (size > 0) {//BFS
            TreeNode node = queue.poll();
            level.add(node.val);
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
            size--;
        }
        result.add(level);
    }
    return result;
}
```

递归

```
List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<List<Integer>>();
    if (root == null) {
        return result;
    }
    levelOrderHelper(result, root, 0);
    return result;
}
private void levelOrderHelper(List<List<Integer>> result, TreeNode current, int level) {
    if (current == null) {
        return;
    }
    if (result.size() == level) {
        result.add(new ArrayList<Integer>());
    }
    result.get(level).add(current.val);

    levelOrderHelper(result, current.left, level + 1);
    levelOrderHelper(result, current.right, level + 1);
}
```

19. 在二叉树中插入结点

```
TreeNode insertNode(TreeNode root,TreeNode node){
        if(root == node){
            return node;
        }
        TreeNode tmp = new TreeNode();
        temp = root;
        TreeNode last = null;
        while(temp!=null){
            last = temp;
            if(temp.val>node.val){
                temp = temp.left;
            }else{
                temp = temp.right;
            }
        }
        if(last!=null){
            if(last.val>node.val){
                last.left = node;
            }else{
                last.right = node;
            }
        }
        return root;
    }
```

20. 输入一个二叉树和一个整数，打印出二叉树中节点值的和等于输入整数所有的路径

```
void findPath(TreeNode r,int i){
    if(root == null){
        return;
    }
    Stack<Integer> stack = new Stack<Integer>();
    int currentSum = 0;
    findPath(r, i, stack, currentSum);
}
void findPath(TreeNode r,int i,Stack<Integer> stack,int currentSum){
    currentSum+=r.val;
    stack.push(r.val);
    if(r.left==null&&r.right==null){
        if(currentSum==i){
            for(int path:stack){
                System.out.println(path);
            }
        }
    }
    if(r.left!=null){
        findPath(r.left, i, stack, currentSum);
    }
    if(r.right!=null){
        findPath(r.right, i, stack, currentSum);
    }
    stack.pop();
}
```

21. 二叉树的搜索区间

给定两个值 k1 和 k2（k1 < k2）和一个二叉查找树的根节点。找到树中所有值在 k1 到 k2 范围内的节点。即打印所有x (k1 <= x <= k2) 其中 x 是二叉查找树的中的节点值。返回所有升序的节点值。

```
ArrayList<Integer> result;
ArrayList<Integer> searchRange(TreeNode root,int k1,int k2){
    result = new ArrayList<Integer>();
    searchHelper(root,k1,k2);
    return result;
}
void searchHelper(TreeNode root,int k1,int k2){
    if(root == null){
        return;
    }
    if(root.val>k1){
        searchHelper(root.left,k1,k2);
    }
    if(root.val>=k1&&root.val<=k2){
        result.add(root.val);
    }
    if(root.val<k2){
        searchHelper(root.right,k1,k2);
    }
}
```

22. 二叉树中两个结点的最长距离

二叉树中两个节点的最长距离可能有三种情况:

1. 左子树的最大深度+右子树的最大深度为二叉树的最长距离
2. 左子树中的最长距离即为二叉树的最长距离
3. 右子树种的最长距离即为二叉树的最长距离
   因此, 递归求解即可

```
private static class Result{
    int maxDistance;
    int maxDepth;
    public Result() {}
    public Result(int maxDistance, int maxDepth) {
        this.maxDistance = maxDistance;
        this.maxDepth = maxDepth;
    }
}
int getMaxDistance(TreeNode root){
  return getMaxDistanceResult(root).maxDistance;
}
Result getMaxDistanceResult(TreeNode root){
    if(root == null){
        Result empty = new Result(0,-1);
        return empty;
    }
    Result lmd = getMaxDistanceResult(root.left);
    Result rmd = getMaxDistanceResult(root.right);
    Result result = new Result();
    result.maxDepth = Math.max(lmd.maxDepth,rmd.maxDepth) + 1;
    result.maxDistance = Math.max(lmd.maxDepth + rmd.maxDepth,Math.max(lmd.maxDistance,rmd.maxDistance));
    return result;
}
```

23. 不同的二叉树

给出 n, 问由 1...n 为节点组成的不同的二叉查找树有多少种？

```
int numTrees(int n ){
    int[] counts = new int[n+2];
    counts[0] = 1;
    counts[1] = 1;
    for(int i = 2;i<=n;i++){
        for(int j = 0;j<i;j++){
            counts[i] += counts[j] * counts[i-j-1];
        }
    }
    return counts[n];
}
```

24. 判断二叉树是否为合法BST

一棵BST定义为:

- 节点的左子树中的值要严格小于该节点的值。
- 节点的右子树中的值要严格大于该节点的值。
- 左右子树也必须是二叉查找树。
- 一个节点的树也是二叉查找树。

```
public int lastVal = Integer.MAX_VALUE;
public boolean firstNode = true;
public boolean isValidBST(TreeNode root) {
    if(root==null){
        return true;
    }
    if(!isValidBST(root.left)){
        return false;
    }
    if(!firstNode&&lastVal >= root.val){
        return false;
    }
    firstNode = false;
    lastVal = root.val;
    if (!isValidBST(root.right)) {
        return false;
    }
    return true;
}
```