链滴

常见算法总结 - 二叉树篇

作者：valarchie

原文链接：https://ld246.com/article/1589686517952

来源网站：链滴

许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

本文总结了常见高频的关于二叉树的算法考察。

1.计算一个给定二叉树的叶子节点数目。

可以采用递归的方式进行累加

```
public static int calculateTreeNodeNumber(TreeNode treeNode) {

        if (treeNode == null) {
            return 0;
        }

        return calculateTreeNodeNumber(treeNode.left) + calculateTreeNodeNumber(treeNode.
ight) + 1;

}
```

2.计算二叉树的深度。

跟上题一样采用递归的方式，但需返回左右子树中较深的深度。

```
public static int getTreeDepth(TreeNode tree) {

        if (tree == null) {
            return 0;
        }

        int left = getTreeDepth(tree.left);
        int right = getTreeDepth(tree.right);

        return left >= right ? left + 1 : right + 1;
}
```

3.如何打印二叉树每层的节点。

借助一个队列，先把根节点入队，每打印一个节点的值时，也就是打印队列头的节点时，都会把它的左右孩子入队，并且把该节点出队。直到队列为空。

```
public static void printByLevel(TreeNode tree) {
        if (tree == null) {
            return;
        }

        LinkedList<TreeNode> queue = new LinkedList<>();
        queue.add(tree);

        while (!queue.isEmpty()) {
            TreeNode pop = queue.pop();
            System.out.println(pop.val);
            if (pop.left != null) {
                queue.add(pop.left);
```

```
            }
            if (pop.right != null) {
                queue.add(pop.right);
            }
        }

}
```

4.二叉树的Z型遍历。

借助两个队列，一个正序打印，一个逆序打印。

```
 public static void printByZ(TreeNode tree) {

        if (tree == null) {
            return;
        }

        List<TreeNode> orderQueue = new ArrayList<>();
        List<TreeNode> disorderQueue = new ArrayList<>();

        orderQueue.add(tree);

        while (!orderQueue.isEmpty()|| !disorderQueue.isEmpty()) {
            if (!orderQueue.isEmpty()) {
              for (int i = 0; i < orderQueue.size(); i++) {
                    TreeNode leaf = orderQueue.get(i);
                    if (leaf.left != null) {
                        disorderQueue.add(leaf.left);
                    }
                    if (leaf.right != null) {
                        disorderQueue.add(leaf.right);
                    }
                }

                for (TreeNode node : orderQueue) {
                    System.out.println(node.val);
                }
                orderQueue.clear();

            }


            if (!disorderQueue.isEmpty()) {

                for (int i = 0; i < disorderQueue.size(); i++) {
                    TreeNode leaf = disorderQueue.get(i);
                    if (leaf.left != null) {
                        orderQueue.add(leaf.left);
                    }
                    if (leaf.right != null) {
                        orderQueue.add(leaf.right);
                    }
                }
```

```
            for (int i = disorderQueue.size()-1; i >=0 ; i--) {
                TreeNode leaf = disorderQueue.get(i);
                System.out.println(leaf.val);
            }
            disorderQueue.clear();
        }
    }

}
```

5.一个已经构建好的TreeSet，怎么完成倒排序。

递归更换左右子树即可

```
public static void reverse(TreeNode tree) {

    if (tree.left == null && tree.right == null) {
        return;
    }

    TreeNode tmp = tree.right;

    tree.right = tree.left;
    tree.left = tmp;

    reverse(tree.left);
    reverse(tree.right);

}
```

6.二叉树的前序遍历。

前序递归

```
public static void preOrderRecursion(TreeNode tree) {

    if (tree != null) {
        System.out.print(tree.val + "->");
        preOrderRecursion(tree.left);
        preOrderRecursion(tree.right);
    }

}
```

前序非递归

```
public static void preOrderNonRecursion(TreeNode tree) {

    Stack<TreeNode> stack = new Stack<>();
    TreeNode node = tree;
    while (node != null || !stack.empty()) {
        if (node != null) {
```

```
                System.out.print(node.val + "->");
                stack.push(node);
                node = node.left;
            } else {
                TreeNode tem = stack.pop();
                node = tem.right;
            }
        }
    }
```

7.二叉树的中序遍历。

中序递归

```
public static void middleOrderRecursion(TreeNode tree) {

        if (tree != null) {
            middleOrderRecursion(tree.left);
            System.out.print(tree.val + "->");
            middleOrderRecursion(tree.right);
        }

}
```

中序非递归

```
public static void middleOrderNonRecursion(TreeNode tree) {

        Stack<TreeNode> stack = new Stack<>();
        TreeNode node = tree;
        while (node != null || !stack.isEmpty()) {
            if (node != null) {
                stack.push(node);
                node = node.left;
            } else {
                TreeNode tem = stack.pop();
                System.out.print(tem.val + "->");
                node = tem.right;
            }
        }

    }
```

8.二叉树的后序遍历。

后序递归

```
public static void postOrderTraverseRecursion(TreeNode root) {
        if (root != null) {
            postOrderTraverseRecursion(root.left);
            postOrderTraverseRecursion(root.right);
            System.out.print(root.val + "->");
        }
```

```
}
```

后序非递归

```
public static void postOrderTraverseNonRecursion1(TreeNode root) {

        LinkedList<TreeNode> stack = new LinkedList<>();
        LinkedList<TreeNode> output = new LinkedList<>();
        if (root == null) {
            return;
        }

        stack.add(root);
        while (!stack.isEmpty()) {

            TreeNode node = stack.pollLast();
            output.addFirst(node);

            if (node.left != null) {
                stack.add(node.left);
            }
            if (node.right != null) {
                stack.add(node.right);
            }
        }

        for (TreeNode node : output) {
            System.out.print(node.val + "->");
        }

}
```

笔者个人总结，如有错误之处望不吝指出。