# Onestepoffer算法第九讲

指针数据结构 --树(Tree) part1

# 目录

# 1. 树(Tree)的数据结构介绍

链表的升级版，我们通常接触的树都是二叉树（binary tree），即每个节点最多有

两个子节点；且除非题目说明，默认树中不存在循环结构。默认的树表示方法如下

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

# 2. 树的递归

437. Path Sum III (Medium)

给定一个整数二叉树，求有多少条路径节点值的和等于给定值。

输入一个二叉树和一个给定整数，输出一个整数，表示有多少条满足条件的路径。

Input: sum = 8, tree =
10
/ \
5 -3
/ \ \
3 2 11
/ \ \
3 -2 1
Output: 3

在这个样例中，和为 8 的路径一共有三个:[[5,3],[5,2,1],[-3,11]]。

# 题目分析

递归每个节点时，需要分情况考虑：

（1）如果选取该节点加入路径，则之后必须继续加入连

续节点，或停止加入节点

（2）如果不选取该节点加入路径，则对其左右节点进行重新进行考虑。

因此一个方便的方法是我们创建一个辅函数，专门用来计算连续加入节点的路径。

# C++ 解决方案

```
// 主函数
int pathSum(TreeNode* root, int sum) {
        return root? pathSumStartWithRoot(root, sum) + pathSum(root->left, sum) +
pathSum(root->right, sum): 0;
}
// 辅函数
int pathSumStartWithRoot(TreeNode* root, int sum) {
    if (!root)  return 0;
    int count = root->val == sum? 1: 0;
    count += pathSumStartWithRoot(root->left, sum - root->val);
    count += pathSumStartWithRoot(root->right, sum - root->val);
    return count;
}
```

# Java 解答

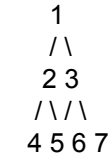
```
// 主函数
public int pathSum(TreeNode root, int sum) {
    return root == null? 0: pathSumStartWithRoot(root, sum) +
        pathSum(root.left, sum) + pathSum(root.right, sum);
}

// 辅函数
public int pathSumStartWithRoot(TreeNode root, int sum) {
    if (root == null) return 0;

    int count = (root.val == sum) ? 1: 0;

    count += pathSumStartWithRoot(root.left, sum - root.val);
    count += pathSumStartWithRoot(root.right, sum - root.val);
    return count;
}
```

# 1110. Delete Nodes and Return Forests(Medium)

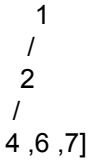
题目描述：给定一个整数二叉树和一些整数，求删掉这些整数对应的节点后，剩余的子树。

输入是一个整数二叉树和一个一维整数数组，输出一个数组，每个位置存储一个子树（的根节点）。

Input: to_delete = [3,5], tree =



Output: [



4 ,6 ,7]

# 题目分析

这道题最主要需要注意的细节是如果通过递归处理原树，以及需要在什么时候断开指针。

同时，为了便于寻找待删除节点，可以建立一个哈希表方便查找。

建议各位同学在看完题解后，自己写一遍本题，加深对于递归的理解和运用能力。

# C++ 解答

```
// 主函数
vector<TreeNode*> delNodes(TreeNode* root, vector<int>& to_delete) {
     vector<TreeNode*> forest;
     unordered_set<int> dict(to_delete.begin(), to_delete.end());
     root = helper(root, dict, forest);
     if (root) forest.push_back(root);
     return forest;
}

// 辅函数
TreeNode* helper(TreeNode* root, unordered_set<int> & dict, vector<TreeNode*> &forest) {
    if (!root) return root;
    root->left = helper(root->left, dict, forest);
    root->right = helper(root->right, dict, forest);
    if (dict.count(root->val)) {
         if (root->left) {
            forest.push_back(root->left);
        }
        if (root->right) {
            forest.push_back(root->right);
        }
       root = NULL;
   }
   return root;
}
```

# Java 解答

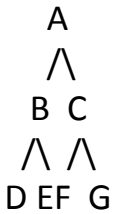
```
class Solution {
    List<TreeNode> forest = new ArrayList<>();
    Set<Integer> dict = new HashSet<>();
    public List<TreeNode> delNodes(TreeNode root, int[] to_delete) {
        for (int ele: to_delete) dict.add(ele);
        root = helper(root);
        if (root != null) forest.add(root);
        return forest;
    }
```

```
    public TreeNode helper(TreeNode root) {
        if (root == null) return root;
        root.left = helper(root.left);
        root.right = helper(root.right);
        if (dict.contains(root.val)) {
            if (root.left != null) forest.add(root.left);
            if (root.right != null) forest.add(root.right);
            root = null;
        }
        return root;
    }
}
```

# 3. 层次遍历

**深度优先遍历(DFS)**：从根节点出发，沿着左子树方向进行**纵向遍历**，直到找到叶子节点为止。然后回溯到前一个节点，进行右子树节点的遍历，直到遍历完所有可达节点为止。

**广度优先遍历(BFS)**：从根节点出发，在**横向遍历**二叉树层段节点的基础上纵向遍历二叉树的层次。




DFS:ABDECFG

BFS:ABCDEFG

# 637. Average of Levels in Binary Tree

给定一个二叉树，求每一层的节点值的平均数。

输入是一个二叉树，输出是一个一维数组，表示每层节点值的平均数。

Input:
3
/ \
9 20
/ \
15 7
Output: [3, 14.5, 11]

# C++ 解法

```
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> ans;
    if (!root) return ans;
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int count = q.size();
        double sum = 0;
        for (int i = 0; i < count; ++i) {
            TreeNode* node = q.front();
            q.pop();
            sum += node->val;
            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }
    }
    ans.push_back(sum / count);
    return ans;
}
```

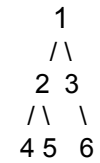
# Java 解法

```
public List < Double > averageOfLevels(TreeNode root) {
    List < Integer > count = new ArrayList < > ();
    List < Double > res = new ArrayList < > ();
    average(root, 0, res, count);
    for (int i = 0; i < res.size(); i++)
        res.set(i, res.get(i) / count.get(i));
    return res;
}
public void average(TreeNode t, int i, List < Double > sum, List < Integer > count) {
    if (t == null) return;

    if (i < sum.size()) {
        sum.set(i, sum.get(i) + t.val);
        count.set(i, count.get(i) + 1);
    } else {
        sum.add(1.0 * t.val);
        count.add(1);
    }
    average(t.left, i + 1, sum, count);
    average(t.right, i + 1, sum, count);
}
```

# 前中后遍历

前序遍历、中序遍历和后序遍历是三种利用深度优先搜索遍历二叉树的方式。

它们是在对节点访问的顺序有一点不同，其它完全相同。考虑如下一棵树




前序遍历先遍历父结点，再遍历左结点，最后遍历右节点，我们得到的遍历顺序是 [1 2 4 5 3 6]。

```
void preorder(TreeNode* root) {
    visit(root);
    preorder(root->left);
    preorder(root->right);
}
```

# 前中后遍历

中序遍历先遍历左节点，再遍历父结点，最后遍历右节点，我们得到的遍历顺序是 [4 2 5 1 3 6]。

```
void inorder(TreeNode* root) {
    inorder(root->left);
    visit(root);
    inorder(root->right);
}
```

后序遍历先遍历左节点，再遍历右结点，最后遍历父节点，我们得到的遍历顺序是 [4 5 2 6 31]。

```
void postorder(TreeNode* root) {
    postorder(root ->left);
    postorder(root ->right);
    visit(root);
}
```

# 课后作业

### 105. Construct Binary Tree from Preorder and Inorder Traversal (Medium)

给定一个二叉树的前序遍历和中序遍历结果，尝试复原这个树。已知树里不存在重复值的节点。

### 144. Binary Tree Preorder Traversal (Medium)

不使用递归，实现二叉树的前序遍历。

### 99. Recover Binary Search Tree (Hard)

给定一个二叉查找树，已知有两个节点被不小心交换了，试复原此树。

### 208. Implement Trie (Prefifix Tree) (Medium)

尝试建立一个字典树，支持快速插入单词、查找单词、查找单词前缀的功能。