二叉树

04 重建二叉树

- **题目**
  输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。
- **思路**
  首先选取前序遍历的第一个节点 x 为根节点，在中序遍历序列中找到这个x，x前面的与后面的划成两个子中序遍历序列，通过这两个子序列的长度对当前前序遍历也划分成两个前序子序列，进行递归。
- **基础概念**
  前序遍历：根结点 ---> 左子树 ---> 右子树
  中序遍历：左子树---> 根结点 ---> 右子树
  后序遍历：左子树 ---> 右子树 ---> 根结点
  层次遍历：仅仅需按层次遍历就可以

- **代码实现**

```
/**
 * Definition for binary tree
 * struct TreeNode {
 * int val;
 * TreeNode *left;
 * TreeNode *right;
 * TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode * combine_tree(vector<int>pre ,int a_begin,int a_last,vector<int>vin,int b_begin,int b_last){
        if(a_begin > a_last|| b_begin > b_last){
            return NULL;
        }

        TreeNode * root = new TreeNode(pre[a_begin]);
        int i;
        for( i = b_begin ; i <= b_last ; i++){
            if(vin[i] == pre[a_begin])
                 break;
        }

        root -> left = combine_tree(pre,a_begin+1,a_begin+i-b_begin,vin,b_begin,i-1);
        root -> right = combine_tree(pre,a_begin+i+1-b_begin,a_last,vin,i+1,b_last);
        return root;
    }

    TreeNode* reConstructBinaryTree(vector<int> pre,vector<int> vin) {
        return combine_tree(pre,0,pre.size()-1,vin,0,vin.size()-1);
    }
};
```

17 树的子结构【1】【递归遍历】

- **题目**
  输入两棵二叉树A，B，判断B是不是A的子结构。（ps：我们约定空树不是任意一个树的子结构）
- **思路**
  很明显分为两部分：
  1、遍历整个树A，当前值与树B相同，则判断是否为子结构；不相同则递归树A的左右子树和树B
  2、判断是否为子结构，在树B为空时说明符合，树1和数2值相同时，继续判断各自的左右子树。
  比较不错的一个题

1. 首先对当前两个指针判断是否有个为空，若为空则返回false
2. 如果 A 的值与 B 的值相同，则调用判断函数，传入两个当前指针比较是否为子结构。
3. 若不是子结构，则递归判断A树的左右子树内是否有子结构。
   关于判断是否为子结构函数：
4. 如果当前B树为空，则返回true
5. 如果A树为空，则返回false
6. 继续判断 A与B的值是否相同，若相同则对A和B的左右子树再进行验证，不相同则返回fasle。

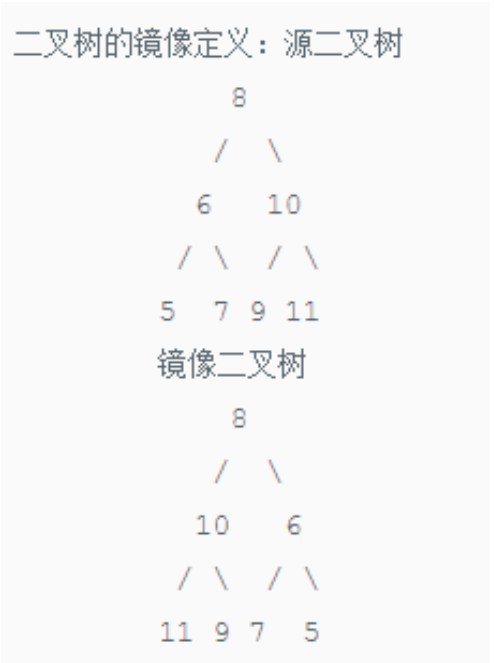
- 代码实现

```
class Solution {
    bool FindSon(TreeNode* p1, TreeNode* p2){
        if(!p2)
            return true;
        if(!p1 || p1->val != p2->val)
            return false;
        return FindSon(p1->left, p2->left) && FindSon(p1->right, p2->right);
    }
public:
    bool HasSubtree(TreeNode* pRoot1, TreeNode* pRoot2)
    {
```

```
        if(!pRoot1 || !pRoot2)
            return false;
        bool res = false;
        if(pRoot1 -> val == pRoot2 -> val)
            res = FindSon(pRoot1, pRoot2);
        if(!res)
            res = HasSubtree(pRoot1->left, pRoot2) ||
                  HasSubtree(pRoot1->right, pRoot2);
        return res;
    }
};
```

18 二叉树的镜像

- **题目**
  操作给定的二叉树，将其变换为源二叉树的镜像。



- **思路**
  递归实现
- 代码实现

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root)
            return NULL;
```

```
        queue<TreeNode *> q;
        q.push(root);
        while(!q.empty()){
            TreeNode * temp = q.front(); q.pop();
            if(temp->left){
                q.push(temp->left);
            }
            if(temp->right)
                q.push(temp->right);
            TreeNode * t = temp->left;
            temp->left = temp->right;
            temp->right = t;
        }
        return root;
    }
};
```

22 从上往下打印二叉树

- **题目**
  从上往下打印出二叉树的每个节点，同层节点从左至右打印。
- **思路**
  用队列的性质来完成层次遍历。
- 代码实现

```
class Solution {
public:
    vector<int> PrintFromTopToBottom(TreeNode* root) {
        vector<int> result;
        if(root == NULL)
            return result;
        queue <TreeNode*> ptr;
        ptr.push(root);
        while(!ptr.empty()){
            TreeNode * p_temp = ptr.front();
            if(p_temp ->left != NULL)
                ptr.push(p_temp ->left);
            if(p_temp ->right != NULL)
```

```
                ptr.push(p_temp ->right);
            result.push_back(p_temp->val);
            ptr.pop();
        }
        return result;
    }
};
```

24 二叉树中和为某一值的路径【1】【深度递归遍历】

- **题目**
  输入一颗二叉树的跟节点和一个整数，打印出二叉树中结点值的和为输入整数的所有路径。路径定义为从树的根结点开始往下一直到叶结点所经过的结点形成一条路径。(注意: 在返回值的list中，数组长度大的数组靠前)
- **思路**
  首先用的是递归的深度遍历，越长的叶子节点，最开始判断路径和是否符合要求的
  1、当前指针为空，则直接返回NULL（不能确定父节点是不是叶子节点）
  2、当前指针不为空，把当前值val压入一维路径数组，当前和sum减去val，深度遍历左子树和右子树。
  接着如果sum为0且是叶子节点则把一维路径数组存入结果，弹出一维路径数组的当前值（注意为空时不能弹出），返回result
- 代码实现

```
class Solution {
    vector<int> temp;
    vector<vector<int>> res;
public:
    vector<vector<int> > FindPath(TreeNode* root,int sum) {
         if(!root)
             return { };
        temp.push_back(root->val);
        sum -= root->val;
        FindPath(root->left, sum);
        FindPath(root->right, sum); //深度遍历
        if(sum == 0 && !root->left && !root->right)  //叶子节点
            res.push_back(temp);
```

```
        if(!temp.empty())  //处理传入的是空指针
            temp.pop_back();//temp保存当前遍历路径的值
        return res;
    }
};
```

38 二叉树的深度

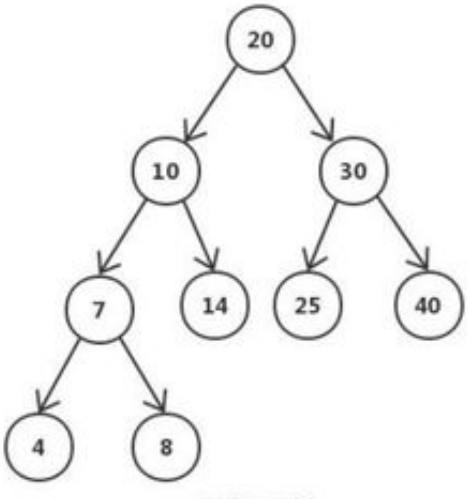
- **题目**
  输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。
- **思路**
  1、递归的思路，指针为空返回0，不为空返回左右子树的最大值+1（过于简单，面试应该会考察非递归的方法）
  2、非递归，采用队列容器，根节点入队，容器不为空时，深度加1，计算队列所含的节点个数（该层的节点数），遍历每个节点，把非空的左右孩子入队，并弹出节点。
- 代码实现

```
//递归
class Solution {
public:
    int TreeDepth(TreeNode* pRoot){
        return pRoot? max(TreeDepth(pRoot->left), TreeDepth(pRoot->right)) + 1:0;
    }
};
//非递归
class Solution {
public:
    int TreeDepth(TreeNode* pRoot){
        if(!pRoot)
            return 0;
        int depth = 0;
        queue<TreeNode*> que;
        que.push(pRoot);
        while(!que.empty()){
            ++depth;
```

```
            for(int i = 0,len = que.size(); i < len; ++i){
                TreeNode * p = que.front();
                que.pop();
                if(p->left)
                    que.push(p->left);
                if(p->right)
                    que.push(p->right);
            }
        }
        return depth;
    }
```

39 平衡二叉树【1】【深度遍历求深度+剪枝优化】

- **题目**
  输入一棵二叉树，判断该二叉树是否是平衡二叉树。
- **思路**
  判断是否为平衡二叉树通过数的深度差是否小于等于1
  通过深度遍历，从底向上判断
  每一轮比较左右子树是否平衡（判断条件为高度差小于1），平衡则返回子树的深度。
- **基础知识**
  平衡二叉树又称为AVL树，且具有以下性质：他是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且两个子树都是一棵平衡二叉树。平衡二叉树常见的实现方法有红黑树、AVL、替罪羊树、Treatp、伸展树等。（有个疑惑，平衡二叉树是排序树吗，即右子树 > 根节点 > 右子树）



平衡二叉树

红黑树

红黑树是一种自平衡二叉查找树，是在计算机科学中用到的一种数据结构，典型的用途是实现关联数组。它是在1972年由Rudolf Bayer发明的，他称之为"对称二叉B树"，它现代的名字是在 Leo J. Guibas 和 Robert Sedgewick 于1978年写的一篇论文中获得的。它是复杂的，但它的操作有着良好的最坏情况运行时间，并且在实践中是高效的: 它可以在O(log n)时间内做查找，插入和删除，这里的n是树中元素的数目。

AVL

AVL是最先发明的自平衡二叉查找树算法。在AVL中任何节点的两个儿子子树的高度最大差别为一，所以它也被称为高度平衡树，n个结点的AVL树最大深度约1.44log2n。查找、插入和删除在平均和最坏情况下都是O（log n）。增加和删除可能需要通过一次或多次树旋转来重新平衡这个树。

Treap

Treap是一棵二叉排序树，它的左子树和右子树分别是一个Treap，和一般的二叉排序树不同的是，Treap纪录一个额外的数据，就是优先级。Treap在以关键码构成二叉排序树的同时，还满足堆的性质(在这里我们假设节点的优先级大于该节点的孩子的优先级)。但是这里要注意的是Treap和二叉堆有一点不同，就是二叉堆必须是完全二叉树，而Treap并不一定是。

伸展树

伸展树（Splay Tree）是一种二叉排序树，它能在O(log n)内完成插入、查找和删除操作。它由Daniel Sleator和Robert Tarjan创造。它的优势在于不需要记录用于平衡树的冗余信息。在伸展树上的一般操作都基于伸展操作。

- 代码实现

```
//推荐 该方法
class Solution {
    int GetDepth(TreeNode * p){
        if(!p)
            return 0;
        int left = GetDepth(p->left); //获取左子树深度
        if(left == -1) //说明左子树不平衡
            return -1;
        int right = GetDepth(p->right);
        if(right == -1)
            return -1;
        return abs(left - right) < 2 ? max(left, right)+1:-1;//不平衡则返回-1（根据这个-1可以优化很多不必要的重复判断），否则返回子树的深度
    }
```

```
public:
    bool IsBalanced_Solution(TreeNode* pRoot) {
        return GetDepth(pRoot) != -1;
    }
};


//方法1 递归判断节点的深度，存在多次重复的情况，不建议
class Solution {
    int depth(TreeNode * p){
        return p ? max(depth(p->left),depth(p->right)) + 1:0;
    }
public:
    bool IsBalanced_Solution(TreeNode* pRoot) {
        if(!pRoot)
            return true;
        return (abs(depth(pRoot->left) - depth(pRoot->right)) < 2) &&
            IsBalanced_Solution(pRoot->left) &&
IsBalanced_Solution(pRoot->right); //获取树的深度时，存在重读遍历子树
    }
};
```

57 二叉树的下一个节点

- **题目**
  给定一个二叉树和其中的一个结点，请找出中序遍历顺序的下一个结点并且返回。注意，树中的结点不仅包含左右子结点，同时包含指向父结点的指针。
- **思路**
  给定某个节点的指针p，找它中序遍历的下一个节点。
  中序遍历：中序遍历左子树；访问根节点；中序遍历右子树；
  如果p为NULL则返回空指针；
  如果p右子树不为空；那么下个节点肯定在右子树上，且在右子树的左子树上；
  只能找父节点；如果当前节点=父节点的左子树，则返回父节点值，否则继续往上找父节点。
  返回NULL;
- 代码实现

```
class Solution {
```

```
public:
    TreeLinkNode* GetNext(TreeLinkNode* pNode)
    {
        if(pNode == NULL)
            return NULL;
        if(pNode->right){    //下个节点在右子树上
            pNode = pNode->right;
            while(pNode->left){
                pNode = pNode ->left;;
            }
            return pNode;
        }
        while(pNode->next){  //下个节点在父节点上（可能为NULL）
            TreeLinkNode* p = pNode ->next;
            if(p->left == pNode)
                return p;
            pNode = p;
        }
        return NULL;
    }
};
```

58 对称的二叉树

- **题目**
  请实现一个函数，用来判断一颗二叉树是不是对称的。注意，如果一个二叉树同此二叉树的镜像是同样的，定义其为对称的。
- **思路**
  采用递归的方法，去遍历
- 代码实现

```
class Solution {
public:
    bool isSymmetrical(TreeNode* pRoot)
    {
        if(pRoot == NULL)
            return true;
        return judge(pRoot->left , pRoot->right);
```

```
    }
private:
    bool judge(TreeNode*p1 , TreeNode* p2){
        if(!p1&&!p2)    //都为空指针
            return true;
        if(p1&&p2&&(p1->val ==p2->val)){    //对应节点相等
            return judge(p1->left,p2->right)&&judge(p1->right,p2->left);
        }
        else
            return false;
    }

};
```

59 按之字行打印二叉树

- **题目**
  请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。
- **思路**
  思路1，用两个栈，分别存取奇数层和偶数层的节点指针
  思路2，还是依旧用队列，得出每一层的结果，根据奇偶数层决定一维数组是否翻转，再进行存入到结果数组中
  思路3，用双向链表，根据奇偶数层正向读或反向读实现之字形输出。
- 代码实现

```
class Solution {
public:
    vector<vector<int> > Print(TreeNode* pRoot) {
        if(!pRoot)
            return{};
        vector<vector<int>> res;
        stack <TreeNode*> stack1,stack2; //1负责奇数行，2负责单数行
        stack1.push(pRoot);
        while(!stack1.empty() || !stack2.empty()){
            vector<int> temp;
```

```
            TreeNode * node;
            if(!stack1.empty()){    //奇数行不为空
                while(!stack1.empty()){
                    node = stack1.top();
                    stack1.pop();
                    temp.push_back(node->val);
                    if(node->left)      //左孩子、右孩子节点存入偶数行
                        stack2.push(node->left);
                    if(node->right)
                        stack2.push(node->right);
                }
            }
            else{
                while(!stack2.empty()){
                    node = stack2.top();
                    stack2.pop();
                    temp.push_back(node->val);
                    if(node->right)  //偶数行按右孩子左孩子的顺序存入栈
                        stack1.push(node->right);
                    if(node->left)
                        stack1.push(node->left);
                }
            }
            res.push_back(temp); //存入每一行结果
        }
        return res;
    }
};
```

60 把二叉树打印成多行

- **题目**
  从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。
- **思路**
  首先，层序遍历使用队列的数据结构很明显，接着每一层输出一行，这个可以先统计当前队列的节点数，可以知道该层有多少节点需要输出。
  -**基础知识**
  关于队列和二维数组
  队列：queue q1;q1.push(x);q1.pop();q1.front();q1.back();q1.empty();q1.size();

二维数组：vector <vector > res ; vector temp; res.push_back(temp)实现插入一行

- 代码实现

```
class Solution {
public:
        vector<vector<int> > Print(TreeNode* pRoot) {
            if(!pRoot)
                return {};
            vector< vector<int> > res;
            queue <TreeNode*> p;
            p.push(pRoot);
            while(!p.empty()){
                vector <int> tem;
                for(int i = 0,num_node = p.size(); i < num_node; i++)
{
                    TreeNode * temp = p.front();
                    p.pop();
                    tem.push_back(temp ->val);
                    if(temp->left)
                        p.push(temp->left);
                    if(temp->right)
                        p.push(temp->right);
                }
                res.push_back(tem);
            }
            return res;
        }
};
```

二叉搜索树

23 二叉搜索树的后序遍历序列

- **题目**
  输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出Yes,否则输出No。假设输入的数组的任意两个数字都互不相同。
- **基础知识**

二叉搜索树，也称二叉排序树。它或者是一棵空树，或者是具有下列性质的二叉树： 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值； 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值； 它的左、右子树也分别为二叉排序树。

- **思路**
  数组最后一个数为根节点，以这个根节点，判断数组前面元素能否分成两个左右序列。不能返回false,能则对这两个左右序列进行递归判断。 若判断的序列长度为1，则返回true。
- 代码实现

```
class Solution {
    bool isBST(vector <int> &a ,int low,int high){
        if(low >= high)
            return true;
        int i = low;
        while(i < high){
            if(a[i]< a[high])
                i++;
            else
                break;
        }
        int mid = i;
        while(i < high){
            if(a[i] > a[high])
                i++;
            else if(i < high)
                return false;
        }
        return isBST(a,low,mid-1)&&isBST(a,mid,high-1);
    }
public:
    bool VerifySquenceOfBST(vector<int> seq) {
         int len = seq.size();
        if(len == 0)
            return false;
        return isBST(seq,0,len-1);
    }
};
```

26 二叉搜索树与双向链表【1】【中序遍历】

- **题目**
  输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。
- **思路**
  二叉搜索的树的中序遍历结果即为从小到大的排序。
  中序非递归的方法采用栈来实现
  首先指针cur跟踪每一个节点，当栈或cur不为空时，循环压入左子树直至叶子，然后弹出节点（第一个节点是还需要把root指针指向它），与上一个节点进行连接，再当前节点cur等于右子树
- 代码实现

```
class Solution {
public:
    TreeNode* Convert(TreeNode* root)
    {
        if(!root)
            return NULL;
        stack <TreeNode *> sta;
        bool isfirst = true;

        TreeNode * pre;
        TreeNode * p = root;
        while(!sta.empty() || p){
            while(p){            //遍历左子树入栈
                sta.push(p);
                p = p ->left;
            }

            p = sta.top();    //出栈，访问根节点
            sta.pop();
            if(!isfirst){
                p->left = pre;
                pre->right = p;
                pre = p;
            }
            else{
```

```
                isfirst = false;
                root = p;
                pre = p;;
            }

            p = p->right;    //遍历右子树
        }
        return root;
    }
};
```

62 二叉搜索树的第k个节点

- **题目**
  给定一棵二叉搜索树，请找出其中的第k小的结点。例如，（5，3，7，2，4，6，8）中，按结点数值大小顺序第三小结点的值为4。
- **思路**
  树的非递归遍历方法不熟悉，多练。
  此题采用中序遍历的非递归方法，第k个打印的节点即为第k小的节点。
- **代码实现**

```
class Solution {
    typedef TreeNode * pnode;
public:
    TreeNode* KthNode(TreeNode* pRoot, int k){
        if(!pRoot || !k)
            return NULL;
        stack<pnode> pstack;
        int count = 0;
        pnode p = pRoot;
        while(!pstack.empty() || p){  //栈不为空或p指针不为空
            while(p){ //p不为空不断压入左孩子
                pstack.push(p);
                p = p->left;
            }
            if(!pstack.empty()){  //栈不为空，那么出栈
                p = pstack.top();
                pstack.pop();
```

```
                ++count;
                if(count == k)
                    return p;
                p = p->right;
            }
        }
        return NULL;
    }
};
```