二叉树复习

二叉树的创建

```C++
class TreeNode {
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode(int val) :val(val), left(nullptr), right(nullptr) {

    }
};
```

搜索二叉树的创建

```
class Tree {
private:
    TreeNode* root;

    TreeNode* InsertNode(TreeNode* cur, int val) {
        //第一步，判断递归函数的返回值类型

        if (root == NULL) {
            return new TreeNode(val);
        }//第二步，确定终止条件

        if (val > cur->val) {
            cur->right = InsertNode(cur->right, val);
        }
        else if (val < cur->val) {
            cur->left = InsertNode(cur->left, val);
        }

        return cur;//往上传递生成的结果
    }
public:
    void Insert(int val) {//对外的接口函数
        root = InsertNode(root, val);
    }
};
```

前、中、后序遍历

```
class Tree {
private:
    TreeNode* root;
    void PreTraversal(TreeNode* cur, vector<int>& result) {
        //这里的result是引用，作为最终收集数据的
        //所以无需返回值

        if (cur == NULL) {//终止条件
            return;
        }
        result.push_back(cur->val);//中
        PreTraversal(cur->left, result);//左
        PreTraversal(cur->right, result);//右
    }
public:
    vector<int> PreOrder() {//对外接口
        vector<int> result;
        PreTraversal(root, result);
        return result;
    }
};
```

前、中、后序遍历只是**中、左、右**那三个位置进行调换。

中序用于二叉搜索树遍历时，将元素从小到大保存到数组中

后序用于修改子树的数据并且返回给节点数

层序遍历

递归法

```
class Tree {//递归法
private:
    TreeNode* root;
    void LevelTraversal(TreeNode* cur, vector<vector<int>>& result,int depth) {
        if (cur == NULL) {//终止条件
            return;
        }
```

```
        if (depth == result.size()) {//如果深度等于result行的数量，说明当前层没有进行遍历
            result.push_back(vector<int>());
        }

        result[depth].push_back(cur->val);//在确定的一行插入数据
        LevelTraversal(cur->left, result, depth + 1);
        LevelTraversal(cur->right, result, depth + 1);
        //这份代码没有直接让depth+=1，而是让参数+1，使用到了回溯的思想
    }
public:
    vector<vector<int>> LevelOrder() {
        vector<vector<int>> result;
        int depth = 0;//
        LevelTraversal(root, result,depth);
        return result;
    }
};
```

迭代法

```
class Tree {//迭代法
private:
    TreeNode* root;
public:
    vector<vector<int>> LevelOrder() {
        queue<TreeNode*> que;
        if (root) {
            que.push(root);
        }

        vector<vector<int>> result;
        while (!que.empty()) {
            int size = que.size();
            vector<int> vec;
            while (size--) {
                TreeNode* temp = que.front();
                que.pop();
                vec.push_back(temp->val);
                if (root->left) {
                    que.push(root->left);
                }
                if (root->right) {
                    que.push(root->right);
                }
            }
```

```
            result.push_back(vec);
        }
        return result;
    }
};
```

这份代码应该熟悉到可以背出来

最大深度

需要明确的是：
深度指的是从根节点到叶子节点，**从上到下递增**
高度指的是从叶子节点到根节点，**从下到上递增**

递归法

求最大深度，使用**后序遍历**，即求从**叶子节点到根节点的最大高度。**

本质上还是求**最大高度**

```
class Tree {//后序遍历求最大高度
private:
    TreeNode* root;
    int PostTraversal(TreeNode* cur) {
        if (cur == NULL) {//终止条件
            return 0;
        }

        int leftdepth = PostTraversal(cur->left);//左递归
        int rightdepth = PostTraversal(cur->right);//左递归
        int depth = max(leftdepth, rightdepth) + 1;//当前节点的最大深度

        return depth;
    }
public:
    int MaxDepth() {
        return PostTraversal(root);
    }
};
```

迭代法

层序遍历的天然优势就是能直接求出层数

```
class Tree {//层序遍历求最大深度
private:
    TreeNode* root;
public:
    int MaxDepth() {
        if (root == NULL) {
            return 0;
        }

        queue<TreeNode*> que;
        que.push(root);
        int depth=0;
        while (!que.empty()) {
            int size = que.size();
            depth++;//层序遍历的天然优势，直接能求出层数
            while (size--) {
                TreeNode* temp = que.front();
                que.pop();
                if (temp->left) {
                    que.push(temp->left);
                }
                if (temp->right) {
                    que.push(temp->right);
                }
            }
        }

        return depth;
    }
};
```

最小深度

注意最小深度判断的是叶子节点。例如**左空右不空**以及**左不空右空**的不是叶子节点，则不是最小深度

递归法

```
class Tree {//最小深度
private:
    TreeNode* root;
    int PostTraversal(TreeNode* cur) {
```

```
        if (cur == NULL) {
            return 0;
        }

        int leftDepth = PostTraversal(cur->left);//左
        int rightDepth = PostTraversal(cur->right);//右

        if (cur->left == NULL && cur->right != NULL) {//左空右不空
            return rightDepth + 1;
        }
        else if (cur->left != NULL && cur->right == NULL) {//右空左不空
            return leftDepth + 1;
        }
        else {//左右皆不为空
            return min(leftDepth, rightDepth) + 1;
        }
    }
public:
    int MinDepth() {
        return PostTraversal(root);
    }
};
```

迭代法

仍然使用层序遍历，当 cur 的左右指针都为空的时候立即返回

```
class Tree {
private:
    TreeNode* root;
public:
    int MinDepth() {
        if (root == NULL) {
            return 0;
        }

        queue<TreeNode*> que;
        que.push(root);
        int depth = 0;

        while (!que.empty()) {
            int size = que.size();
            depth++;
            while (size--) {
                TreeNode* temp = que.front();
                que.pop();
```

```
                if (temp->left) {
                    que.push(temp->left);
                }
                if (temp->right) {
                    que.push(temp->right);
                }

                if (temp->left == NULL && temp->right == NULL) {
                    //一旦遇到叶子节点立即返回
                    //说明是最小深度
                    return depth;
                }
            }
        }

        return -1;//假装返回
    }
};
```

翻转二叉树

```
class Tree {//翻转二叉树数
private:
    TreeNode* root;
    TreeNode* Invert(TreeNode* cur) {
        if (cur == NULL) {
            return NULL;
        }

        swap(cur->left, cur->right);
        Invert(cur->left);
        Invert(cur->right);
        //这里使用的是前序，从上到小进行左右节点交换
        //也可以使用后序，则是从下到上进行左右节点交换
    }
public:
    TreeNode* InvertTree() {//直接在
        return Invert(root);
    }
};
```

判断是否为对称二叉树

即判断每个**对称位置**的节点的值是否相同

```
class Tree {//判断是否镜像对称
private:
    TreeNode* root;
    bool compare(TreeNode* left, TreeNode* right) {//首先确定参数与返回值的类型
        if (left == NULL && right == NULL) {
            return true;
        }
        else if ((left != NULL && right == NULL) || (left == NULL && right != NUL
L)) {
            return false;
        }
        else if (left->val != right->val) {
            return false;
        }

        bool outside = compare(left->left, right->right);
        bool inside = compare(left->right, right->left);
        //比较两个同层节点的内测与外侧

        return outside && inside;
    }
public:
    bool isSymmetric() {
        if (root == NULL) {
            return true;
        }

        return compare(root->left, root->right);
    }
};
```

二叉树的节点个数

递归法

不知不觉使用**中序**，因为只用中序才能将其左右节点的相关参数传递给根节点

```
class Tree {
private:
    TreeNode* root;
    int Count(TreeNode* cur) {
        if (cur == NULL) {
```

```
            return 0;
        }

        int leftCount = Count(cur->left);//左
        int rightCount = Count(cur->right);//右

        return leftCount + rightCount + 1;//中
    }
public:
    int CountNodes() {
        if (root == NULL) {
            return 0;
        }

        return Count(root);
    }
};
```

迭代法

使用层序遍历，直接记录每一层的节点数量

层序遍历真 TM 好用啊

```
class Tree {
private:
    TreeNode* root;
public:
    int CountNodes() {
        if (root == NULL) {
            return 0;
        }

        queue<TreeNode*> que;
        que.push(root);
        int count = 0;
        while (!que.empty()) {
            int size = que.size();
            while (size--) {
                TreeNode* temp = que.front();
                que.pop();
                count++;
                if (temp->left) {
                    que.push(temp->left);
                }
                if (temp->right) {
```

```
                    que.push(temp->right);
                }
            }
        }

        return count;
    }
};
```

判断是否为平衡二叉树

平衡二叉树的定义：一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

思路是找左右子树的最大高度。如果**左右子树的最大高度差大于 1**，则说明不是二叉树

```
class Tree {
private:
    TreeNode* root;
    int GetHeight(TreeNode* cur) {
        if (cur == NULL) {
            return 0;
        }

        int LeftHeight = GetHeight(cur->left);
        if (LeftHeight == -1) {
            return -1;
        }
        int RightHeight = GetHeight(cur->right);
        if (RightHeight == -1) {
            return -1;
        }
        //如果以已经返回了-1，说明不符合平衡二叉树条件，直接向上返回

        if (abs(LeftHeight - RightHeight) > 1) {//高度绝对值之差
            return -1;
        }
        else {
            return max(LeftHeight, RightHeight) + 1;//返回最大高度
        }


    }
public:
    bool IsBalanced() {
```

```
        if (GetHeight(root) == -1) {
            return false;
        }
        else {
            return true;
        }
    }
};
```

输出从根节点到叶子节点的所有路径

题目中不仅要求将路径保存在 string 类型的数组中，而且还让在两个节点之间输出"->"，这就比较阴间了。

这要求当遍历到叶子节点的时候，不再进行递归，在插入叶子节点的数据后直接返回。

这也就导致递归代码中会有变化

本题也会用到回溯

```
class Tree {
private:
    TreeNode* root;
    void Traversal(TreeNode* cur, vector<int>& path, vector<string>& result) {

        if (cur->left == NULL && cur->right == NULL) {//遍历到这个路径的终点，终止条件
            path.push_back(cur->val);//加上最后一个数据
            string ret;
            for (int i = 0;i < path.size() - 1;i++) {
                ret += to_string(path[i]);
                ret += "->";
            }
            ret += to_string(path[path.size() - 1]);
            result.push_back(ret);
            //path.clear();
            // 这样会删除所有的当前节点前的所有的数据
            //为了模拟出箭头指向的效果
            return;
        }

        path.push_back(cur->val);//中

        //叶子节点不参与递归，所以要加上判断左右节点是否为空的判断
        if (cur->left) {//左
            Traversal(cur->left, path, result);
            path.pop_back();
```

```
        }
        if (cur->right) {//右
            Traversal(cur->right, path, result);
            path.pop_back();
        }
    }
public:
    vector<string> TreePath() {
        vector<string> result;//最终的返回
        vector<int> path;//用来记录每一条路径的数据
        Traversal(root, path, result);
        return result;
    }
};
```

计算左叶子之和

```
class Tree {
private:
    TreeNode* root;
    int LeftSum(TreeNode* cur) {
        if (root == NULL) {
            return 0;
        }

        int leftVal = LeftSum(cur->left);//左
        int rightVal = LeftSum(cur->right);//右
        if (cur->left != NULL && cur->left->left == NULL && cur->left->right == NUL
L) {
            leftVal = root->left->val;
        }//只计算左子树

        return leftVal + rightVal;//中
    }
public:
    int GetLeftSum() {
        return LeftSum(root);
    }
};
```

返回坐下角节点的值

暴力解法

层序遍历得到数组，返回数组最后一行的第一个元素即可

前序遍历

```
class Tree {
private:
    TreeNode* root;
    int result;
    int Maxdepth=-1;
    void LeftBottom(TreeNode* cur, int depth) {
        if (cur->left == NULL && cur->right == NULL) {//找到左下角的节点，就不参与递归了
            if (Maxdepth < depth) {//只有当深度更大的时候才会更新result
                Maxdepth = depth;
                result = cur->val;
            }
            return;
        }

        if (root->left) {//注意这里有判断为空的条件要求
            LeftBottom(root->left, depth + 1);
        }
        if (root->right) {
            LeftBottom(root->right, depth + 1);
        }//这里使用到了回溯
    }
public:
    int GetLeftBottom() {
        LeftBottom(root, 0);
        return result;
    }
};
```

判断二叉树中是否有一个路径值的总和等于目标值

算法：target 每查询一条路径的时候，都减去这个节点的值。最后判断执行到**叶子节点**的时候 target 为 0，说明存在这一条路径

这份代码不需要对**中**进行修改或者操控

```
class Tree {
private:
    TreeNode* root;
    bool Traversal(TreeNode* cur, int count) {
        if (cur->left == NULL && cur->right == NULL && count == cur->val) {//终止条
```

```
件
            return true;
        }
        else if (cur->left == NULL && cur->right == NULL && count != cur->val) {
            return false;
        }

        bool Left,Right;
        if (cur->left) {
            Left = Traversal(cur->left, count - cur->val);
        }//左
        if (cur->right) {
            Right = Traversal(cur->right, count - cur->val);//这里用到了回溯
        }//右

        return Left || Right;//只要左右一个成立即可
    }
public:
    bool HasSum(int sum) {
        if (root == NULL) {
            return false;
        }

        return Traversal(root, sum);
    }
};
```

利用后序数组和中序数组生成一个二叉树

算法：先从后序数组中得到**根节点 root 的 val**，然后再通过 root 的 val 切割中序数组，从**中序数组中得到左以及右。**

循环这个过程

```
class Tree {
private:
    TreeNode* root;
    TreeNode* Build(vector<int>& inorder, vector<int>& postorder) {
        if (postorder.size() == 0) {//终止条件
            return NULL;
        }

        int rootValue = postorder[postorder.size() - 1];
        TreeNode* root = new TreeNode(rootValue);//得到根节点
```

```
        if (postorder.size() == 1) {
            return root;
        }

        int index;
        for (int i = 0;i < inorder.size();i++) {
            if (inorder[i] == rootValue) {
                index = i;
                break;
            }
        }//找到分割点

        //切割中序数组
        vector<int> LeftInorder(inorder.begin(), inorder.begin() + index);
        //[0, index)
        vector<int> RightInorder(inorder.begin() + index, inorder.end());
        //[index + 1, end)

        postorder.erase(postorder.end() - 1);//删除末尾的节点

        //切割后序数组
        vector<int> LeftPostorder(postorder.begin(), postorder.begin() + LeftInorde
r.size());
        vector<int> RightPostorder(postorder.begin() + LeftInorder.size(), postorde
r.end() - 1);

        root->left = Build(LeftInorder, LeftPostorder);//左边的数组构成二叉树分支

        root->right = Build(RightInorder, RightPostorder);//右边的数组构成二叉树的分支

    }
public:
    TreeNode* BuildTree(vector<int>& inorder, vector<int>& postorder) {
        if (inorder.empty() || postorder.empty()) {
            return NULL;
        }

        return Build(inorder, postorder);
    }
};
```

给定一个数组，构造一个最大的二叉树

最大二叉树的定义:

二叉树的根是数组中的最大元素。

左子树是通过数组中最大值左边部分构造出的最大二叉树。

右子树是通过数组中最大值右边部分构造出的最大二叉树。

```
class Tree {
private:

    TreeNode* MaxTree(vector<int>& nums){
        if (nums.empty()) {
            return NULL;
        }
        else if (nums.size() == 1) {
            return new TreeNode(nums[0]);
        }//特殊状况以及终止条件

        int Maxval=nums[0];
        int pos=0;
        for (int i = 0;i < nums.size();i++) {
            if (Maxval < nums[i]) {
                Maxval = nums[i];
                pos = i;
            }
        }
        TreeNode* root = new TreeNode(Maxval);

        vector<int> left(nums.begin(), nums.begin() + pos);
        vector<int> right(nums.begin() + pos + 1, nums.end());
        root->left = MaxTree(left);
        root->right = MaxTree(right);

        return root;

    }
public:
    TreeNode* BuildMaxTree(vector<int>& nums){

        return MaxTree(nums);
    }
};
```

合并两个二叉树

这里创建了一个新的二叉树，而不是在原二叉树上修改

创建二叉树使用的是前序遍历，毕竟只有先有根节点才能有左右节点

```
class Tree {
private:
    TreeNode* Merge(TreeNode* t1, TreeNode* t2) {
        if (t1 == NULL && t2 == NULL) {
            return NULL;
        }
        else if (t1 == NULL && t2 != NULL) {
            return t2;
        }
        else if (t1 != NULL && t2 == NULL) {
            return t1;
        }//终止条件

        //上面都已经处理了特殊情况了，接下那直接生成就好
        TreeNode* cur = new TreeNode(t1->val + t2->val);//中
        cur->left = Merge(t1->left, t2->right);//左
        cur->right = Merge(t1->right, t2->right);//右

        return cur;//向上一级返回cur
    }
public:
    TreeNode* MergeTree(TreeNode* t1, TreeNode* t2) {//习惯性地只留一个接口

        return Merge(t1, t2);
    }
};
```

在二叉搜索树中查找节点

```
class Tree {
private:
    TreeNode* root;
    TreeNode* Search(TreeNode*cur,int val) {
        if (cur == NULL) {
            return NULL;
        }
        else if (cur->val == val) {
            return cur;
        }

        TreeNode* result = NULL;
        //这里又是新的一种情形
        //不能将发挥的结果传递到cur->left或者cur->right中，那样不符合题意，也会修改原来的二叉树
```

```
        //所以使用result，向上一层单独返回一个节点来判断
        //因为是二叉搜索树，不会出现同时要在左右节点里搜索的情况
        if (val > cur->val) {
            result = Search(cur->right, val);
        }
        else if (val < cur->val) {
            result = Search(cur->left, val);
        }

        return result;
    }
public:
    TreeNode* SearchTree(int val) {
        return Search(root,val);
    }
};
```

验证是否为二叉搜索树

算法：判断是否为**二叉搜索树**，即判断其**中序遍历**的数字是否**单调递增**

```
class Tree {//判断是否为二叉搜索树：即判断其中序遍历的数字是否单调递增
private:
    TreeNode* root;
    void Traversal(TreeNode* cur,vector<int>& vec) {
        if (cur == NULL) {
            return;
        }

        Traversal(cur->left, vec);
        vec.push_back(cur->val);
        Traversal(cur->right, vec);
    }
public:
    bool IsValidBST() {
        vector<int> result;
        Traversal(root, result);
        for (int i = 0;i < result.size() - 1;i++) {
            if (result[i + 1] <= result[i]) {//注意这里相等也不行
                return false;
            }
        }

        return true;
```

```
    }
};
```

计算二叉搜索树的最小绝对差

算法：和上一题一样，变成中序遍历变成一个数组，找两个树的最小差

这份代码和上面几乎一致，复习的时候直接跳过

```
class Tree {
private:
    TreeNode* root;
    void Traversal(TreeNode* cur, vector<int>& result) {
        if (cur == NULL) {
            return;
        }

        Traversal(cur->left, result);
        result.push_back(cur->val);
        Traversal(cur->right, result);
    }
public:
    int GetMinDiffer() {
        vector<int> result;
        Traversal(root, result);

        if (result.size() < 2) {
            return 0;
        }

        int min = result[1]-result[0];
        for (int i = 1;i < result.size() - 1;i++) {
            if (result[i + 1] - result[i] < min) {
                min = result[i + 1] - result[i];
            }
        }

        return min;
    }
};
```

找到二叉树里的众数

我认为这是一道很棒的题目

算法：

将节点的数据保存在 map 中，键值为 val，队值为出现次数

再将 map 中的值赋值给 pair 类型的 vector，并且排序

将 pair 类型的 vector 中的最大的数据再赋值给 result，并且返回

```
class Tree {//二叉树中找众数
private:
    TreeNode* root;
    void Traversal(map<int, int>& m, TreeNode* cur) {
        if (cur == NULL) {
            return;
        }

        m[cur->val]++;//中
        Traversal(m,cur->left);//左
        Traversal(m,cur->right);//右
    }

    bool compare(pair<int, int> a, pair<int, int> b) {
        return a.second > b.second;
    }
public:
    vector<int> FindMost() {
        if (root == NULL) {
            return vector<int>();
        }

        vector<int> result;
        map<int, int> m;
        Traversal(m, root);

        vector<pair<int, int>> vec(m.begin(), m.end());
        sort(vec.begin(), vec.end(),compare);

        for (int i = 0;i < vec.size();i++) {
            result.push_back(vec[0].first);
            if (vec[i].second == vec[0].second) {
                result.push_back(vec[i].first);
            }
            else {
                break;
            }
        }
```

```
        return result;
    }
};
```

二叉树的最近公共祖先

我认为这是一道很棒的题目

算法:

从头节点开始找

如果某一个某一个分支找到了，则逐级向上一层进行返回

思考:

这里的 Left 以及 Right，给我一种 bool 类型的感觉。只不过是为了符合递归的要求而修改其类型为 TreeNode* ,每一个都向上级返回被找到的节点
Left 以及 Right 为空代表没有找到，非空代表找到了。只有当 Left 和 Right 都非空的时候，才会返回其祖先节点
当然这里很巧妙，当出现一个节点是另一个节点的父节点的时候，也能返回正确的节点

```
class Tree {//寻找二叉树的最近的公共祖先
private:
    TreeNode* root;
    TreeNode* Ancestor(TreeNode* a, TreeNode* b,TreeNode* cur) {
        if (a == root || b == root) {
            return root;
        }
        if (cur == NULL) {
            return NULL;
        }

        TreeNode* Left = Ancestor(a, b, cur->left);
        TreeNode* Right = Ancestor(a, b, cur->right);

        if (left == NULL && Right == NULL) {//没有找到这两个节点
            return NULL;
        }
        else if (left == NULL && Right != NULL) {//右侧找到了，左侧没找到
            return Right;
        }
        else if (left != NULL && Right == NULL) {//左侧找到了，右侧没有找到
            return Left;
        }
        else {
```

```
            return cur;
        }
    }
public:
    TreeNode* LowestAncestor(TreeNode* a, TreeNode* b) {
        if (root == NULL) {
            return NULL;
        }
        return Ancestor(a, b, root);
    }
};
```

搜索二叉树的创建

之前的代码中以及有了这个，**跳过**

```
class Tree {
private:
    TreeNode* root;
    TreeNode* Insert(TreeNode* cur, int val) {
        if (cur == NULL) {
            return new TreeNode(val);
        }

        if (val > cur->val) {
            cur->right = Insert(cur->right, val);
        }
        else {
            cur->left = Insert(cur -> left, val);
        }

        return cur;//最终根节点的返回
    }
public:
    TreeNode* InsertBST(int val) {
        root = Insert(root,val);
        return root;
    }
};
```

删除搜索二叉树的节点

暴力解法

前序遍历存数组，数组中删除指定数据，最后生成二叉树

```
class Tree {
private:
    TreeNode* root;
    void Traversal(vector<int>& result, TreeNode* cur) {
        if (cur == NULL) {
            return;
        }

        //这里一定要按照前序来存到数组里
        result.push_back(cur->val);
        Traversal(result, cur->left);
        Traversal(result, cur->right);
    }
    TreeNode* insert(TreeNode* cur, int val) {
        if (cur == NULL) {
            cur = new TreeNode(val);
            return cur;
        }

        if (val > cur->val) {
            cur->right = insert(cur->right, val);
        }
        else {
            cur->left = insert(cur->left, val);
        }

        return cur;
    }
public:
    TreeNode* deleteNode(int key) {
        vector<int> result;
        Traversal(result, root);

        for (int i = 0;i < result.size();i++) {
            if (result[i] == key) {
                result.erase(result.begin() + i);
            }
        }

        TreeNode* newNode = nullptr;
        for (auto it : result) {
            newNode = insert(newNode, it);
        }
```

```
        return newNode;
    }
};
```

其他解法

遍历，分情况判断。

返回值为 TreeNode*类型，最后逆向生成二叉树并且返回根节点

```
class Tree {
private:
    TreeNode* rootNode;
    TreeNode* Delete(TreeNode* root, int key) {
        if (root == NULL) {//没找到要删除的节点
            return root;
        }
        if (root->val == key) {
            if (root->right == NULL && root->right == NULL) {
                delete root;
                return NULL;
            }
            else if (root->left == NULL && root->right != NULL) {
                TreeNode* temp = root->right;
                delete root;
                return temp;//返回后面的节点，使得其还能构成一个二叉树
            }
            else if (root->left != NULL && root->right == NULL) {
                TreeNode* temp = root->left;
                delete root;
                return temp;
            }
            else {//左右都不为空的情况，需要调整位置
                TreeNode* cur = root->right;
                while (cur->left != NULL) {
                    cur = cur->left;
                }
                cur->left = root->right;//将root->left后的树放在root->right的左叶子下
面
                TreeNode* temp = root->right;
                delete root;
                return temp;
            }

            if (key < root->val) {//如果是普通的二叉树，调整这里即可
```

```
                root->left = Delete(root->left, key);//每次都向上返回节点，最后返回整个二叉树
            }
            if (key > root->val) {
                root->right = Delete(root->right, key);
            }

            return root;
        }
    }
public:
    TreeNode* DeleteNode(int key) {
        root = Delete(rootNode, key);
        return root;
    }
};
```

修建二叉树（删除不在范围类的节点）

这种返回根节点的题目，往往要逆向生成二叉树

当一个节点不满足条件时，其左右节点仍然可能满足条件

```
class Tree {
private:
    TreeNode* root;
    TreeNode* Trim(TreeNode* cur,int low, int high) {
        if (root == NULL) {
            return NULL;
        }

        if (cur->val < low) {//虽然cur不满足，但是cur的右节点可能满足
            TreeNode* right = Trim(cur->right, low, high);
            return right;
        }
        if (cur->val > high) {//cur的左节点也可能满足
            TreeNode* left = Trim(cur->left, low, high);
            return left;
        }
        //向上级返回在范围类的Left和right节点，用于逆向生成二叉树

        root->left = Trim(root->left, low, high);
        root->right = Trim(root->right, low, high);

        return root;//最终的返回
```

```
    }
public:
    TreeNode* TrimTree(int low, int high) {
        return Trim(root,low, high);
    }
};
```

使用有序数组生成高度平衡的二叉树

我认为这是很棒的一题

这里使用到了类似二分法的范围划分以及终止判断方法

也使用到了分割数组来生成二叉树的方法

```
class Tree {//将有序的数组生成为高度平衡的二叉树
private:
    TreeNode* root;
    TreeNode* Traversal(vector<int> nums, int left, int right) {
        //这里采用的原则是左闭右闭
        if (left > right) {//终止条件
            return nullptr;
        }
        int mid = (left + right) / 2;
        TreeNode* root = new TreeNode(nums[mid]);//中

        root->left = Traversal(nums, left, mid - 1);//左
        root->right = Traversal(nums, mid + 1, right);//右

        return root;
    }
public:
    TreeNode* GenerateBST(vector<int> nums) {
        TreeNode* root = Traversal(nums, 0, nums.size() - 1);
        return root;
    }
};
```