我的解题报告

胡庆海

Contents

1 第一章：链表 5

2 第二章：树 7

2.1 二叉树的遍历 7

2.1.1 PreOrederTraversal 7

2.1.2 InOrederTraversal 9

2.1.3 PostOrederTraversal 10

2.1.4 LevelOrderTraversal 11

2.1.5 Recover Binary Search Tree 12

2.2 二叉树的建立 15

2.2.1 Construct Binary Tree from Preorder and Inorder Traversal 15

2.2.2 Construct Binary Tree from Postorder and Inorder Traversal 16

2.2.3 Convert Sorted Array to Binary Search Tree 17

2.2.4 Convert Sorted List to Binary Search Tree 17

2.2.5 Unique Binary Search Trees 18

2.2.6 Unique Binary Search Trees II 19

2.3 二叉树的属性 21

2.3.1 Validate Binary Search Tree 21

2.3.2 Symmetric Tree 23

2.3.3 Same Tree 24

2.3.4 Maximum Depth of Binary Tree 25

2.3.5 Minimum Depth of Binary Tree 26

2.3.6 Path Sum 27

2.3.7 Path Sum II 28

2.3.8 Binary Tree Maximum Path Sum 29

2.3.9 Sum Root to Leaf Numbers 30

2.3.10 Least Common Ancest 31

2.4 其他 32

2.4.1 Flatten Binary Tree to Linked List 32

2.4.2 Populating Next Right Pointers in Each Node 33

2.4.3 Populating Next Right Pointers in Each Node II 33

Chapter 1

第一章： 链表

```c
int main(int argc, char *argv[])
{
    int i;
    return 0;
}
```

Chapter 2

第二章： 树

2.1 二叉树的遍历

二叉树的遍历是解决很多二叉树问题的基础，它的递归写法和非递归写法更是需要都要掌握的，这里的遍历就是将树的节点都依次访问一遍，因为访问顺序的问题，就可以分为前序，中序，后序以及层次等很多种遍历的方法。

2.1.1 PreOrederTraversal

问题: Given a binary tree, return the preorder traversal of its nodes' values.(*leetcode 144*)

递归： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
前序遍历的递归写法，非常简单，只需要先访问跟节点，再递归的执行左子树和右子树

```
void BiTree::PreOrderTraversal(TreeNode* root){
    if(!root)    return;
    cout<<root->val<<endl;
    if(root->left)
        PreOrderTraversal(root->left);
    if(root->right)
        PreOrderTraversal(root->right);
}
```

栈式迭代： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
使用栈来模拟递归过程也是很显而易见的，具体做法就是先访问根节点，然后先让右孩子入栈接着是左孩子，然后左孩子出栈后重复这个过程

```
void BiTree::PreOrderTraversal(TreeNode* root){
    if(!root)    return;
    stack<TreeNode*> s;
    TreeNode *cur;
    s.push(root);
    while(!s.empty()){
        cur = s.top();
        s.pop();
        cout<<cur->val<<endl;
        if(cur->right)
            s.push(cur->right);
        if(cur->left)
            s.push(cur->left);
    }
}
```

Mirror迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

Mirror迭代法是经过Lee介绍过来的，非常的迷人,它的做法就是在遍历的过程中，访问了当前节点之后，先找当前节点的前驱并让此前驱的右孩子指向它，再访问它的左孩子并重复这个过程。在此之后会访问到它前驱然后再次回到当前节点，此时再次试图建立前驱，发现已经建立了，这就说明当前节点左边已经全部遍历完，则继续访问当前节点的右边节点，不断的重复此过程。

```
void BiTree::PreOrderTraversal(TreeNode* root){
    if(!root)   return;
    TreeNode *curr = root, *next;
    while(curr){
        next = curr->left;
        if(!next){
            cout<<curr->val<<endl;
            curr = curr->right;
            continue;
        }
        while(next->right && next->right != curr){
            next = next->right;
        }
        if(next->right == curr){
            next->right = NULL;
            curr = curr->right;
        }else{
            cout<<curr->val<<endl;
            next->right = curr;
            curr = curr->left;
        }
    }
}
```

这个*Mirror*算法一旦掌握后，威力无穷，你可以用它方便的建立二叉树前序索引并且遇到那些要求用迭代来实现的二叉树问题也可以很快的写出

来

2.1.2 InOrederTraversal

问题: Given a binary tree, return the inorder traversal of its nodes' values. *(leetcode 94)*

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
void BiTree::InOrderTraversal(TreeNode* root){
    if(!root)    return;
    if(root->left)
        InOrderTraversal(root->left);
    cout<<root->val<<endl;
    if(root->right)
        InOrderTraversal(root->right);
```

栈式迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

这里需要说一下的是，数据结构那本书上写了两种栈式迭代方法，这是其中之一，使用两重循环的那个

```
void BiTree::InOrderTraversal(TreeNode* root){
    vector<int> data;
    if(!root)    return data;
    stack<TreeNode*> s;
    TreeNode *pos = root;
    while(!s.empty() || pos){
        while(pos){
            s.push(pos);
            pos = pos->left;
        }
        pos = s.top();
        s.pop();
        std::cout<<pos->val<<std::endl;
        pos = pos->right;  //这个非常重要
    }
}
```

栈式迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

这里需要说一下的是，数据结构那本书上写了两种栈式迭代方法，这是其中之二，使用一重循环，实际上是一样的

```
void BiTree::InOrderTraversal(TreeNode* root){
    vector<int> data;
    if(!root)    return data;
    stack<TreeNode*> s;
    TreeNode *pos = root;
    while(!s.empty() || pos){
        if(pos){
            s.push(pos);
            pos = pos->left;
        }else{
            pos = s.top();
            s.pop();
            std::cout<<pos->val<<std::endl;
            pos = pos->right;  //这个非常重要
        }
    }
}
```

Mirror迭代: $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

这里Mirror方法和前序的Mirror方法基本一样，唯一的区别就是打印当前值的时机

```
void BiTree::InOrderTraversal(TreeNode* root){
    if(!root)   return;
    TreeNode *curr = root, *next;
    while(curr){
        next = curr->left;
        if(!next){
            cout<<curr->val<<endl;
            curr = curr->right;
            continue;
        }
        while(next->right && next->right != curr){
            next = next->right;
        }
        if(next->right == curr){
            next->right = NULL;
            cout<<curr->val<<endl;
            curr = curr->right;
        }else{
            next->right = curr;
            curr = curr->left;
        }
    }
}
```

2.1.3 PostOrederTraversal

问题: Given a binary tree, return the postorder traversal of its nodes' values. *(leetcode 145)*

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
void BiTree::PostOrderTraversal(TreeNode* root){
    if(!root)    return;
    if(root->left)
        PostOrderTraversal(root->left);
    if(root->right)
        PostOrderTraversal(root->right);
    cout<<root->val<<endl;
}
```

栈式迭代: $\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$

这里判断一个节点的孩子是否被访问过的方法是：记录上一次打印的节点，如果上一次打印的节点是它的孩子节点，那么必然它的所有孩子及其子树都访问完了，换句话说该访问它本身了.

```
void BiTree::PostOrderTraversal(TreeNode* root){
    if(!root)    return;
    stack<TreeNode*> s;
    TreeNode *pos = root, *last = root;
    s.push(root);
    while(!s.empty()){
        pos = s.top();
        if(pos->left == last || pos->right == last || (!
            pos->left && !pos->right)){
        //孩子已经打印完毕或者根本就没有孩子
            cout<<pos->val<<endl;
            last = pos;
            s.pop();
        }else{
            if(pos->right){
                s.push(pos->right);
            }
            if(pos->left){
                s.push(pos->right);
            }
        }
    }
}
```

这里没有出现万众期待的*Mirror*算法，主要是后序使用*Mirror*很复杂，我暂时还没有想到怎么实现 ^_^

2.1.4 LevelOrderTraversal

问题: Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level). *(leetcode 145)*

队列：$\boxed{\text{时间复杂度O(n)，空间复杂度O(w), w为二叉树最大宽度}}$
使用队列，上一层进入队列，然后添加下一层，直到不再有节点进入队列

```cpp
vector<vector<int> > levelOrder(TreeNode *root) {
    vector<vector<int> > data;
    if(!root)    return data;
    queue<TreeNode*> cur;
    cur.push(root);
    int size = 0;
    TreeNode* now;
    while(!cur.empty()){
        vector<int> tmp;
        size = cur.size();
        for(int i = 0; i < size; i++){
            now = cur.front();
            cur.pop();
            tmp.push_back(now->val);
            if(now->left)
                cur.push(now->left);
            if(now->right)
                cur.push(now->right);
        }
        data.push_back(tmp);
    }
    return data;
}
```

其实层次遍历除了这个先上而下，先左而右的顺序以外还有很多顺序，但是都可以通过这个顺序来转换，所以就不再仔细讨论

2.1.5 Recover Binary Search Tree

问题: Two elements of a binary search tree (BST) are swapped by mistake. Recover the tree without changing its structure. *(leetcode 99)*
BST树的中序遍历是排序好的，那么可以通过中序遍历来看一下哪两个地方发生了交换,发生交换的地方必然是前面那个数比后面的大,只要在遍历过程记录这个位置就可以了.

递归：$\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$
这段代码很有技巧，需要细细品读. 对于1,5,3,4,2这个序列: fisrt是5, second是2;对于1,3,2,4,5这个序列: first是3, second是2.

```cpp
void dfs(TreeNode* root, int& last, TreeNode* &first,
    TreeNode* &second){
    if(root->left){
        dfs(root->left, last, first, second);
    }
    if(root->val < last){
        second = root;
    }
    if(!second){
        first = root;
    }
    last = root->val;
    if(root->right){
        dfs(root->right, last, first, second);
    }
}

void recoverTree(TreeNode* root) {
    if(!root)   return;
    TreeNode *first = NULL, *second = NULL;
    int min = (-1)<<31;
    dfs(root, min, first, second);
    int tmp = first->val;
    first->val = second->val;
    second->val = tmp;
}
```

迭代：$\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$

```
void recoverTree(TreeNode* root) {
    if(!root)   return;
    TreeNode *first = NULL, *second = NULL, *cur;
    int last = INT_MIN;
    stack<TreeNode*> s;
    s.push(root);
    while(!s.empty()){
        cur = s.top();
        while(cur){
            s.push(cur->left);
            cur = cur->left;
        }
        s.pop();
        if(!s.empty()){
            cur = s.top();
            if(cur->val < last){
                second = cur;
            }
            if(!second){
                first = cur;
            }
            last = cur->val;
            s.pop();
            s.push(cur->right);
        }
    }
    int tmp = first->val;
    first->val = second->val;
    second->val = tmp;
}
```

2.2 二叉树的建立

2.2.1 Construct Binary Tree from Preorder and Inorder Traversal

问题: Given preorder and inorder traversal of a tree, construct the binary tree. *(leetcode 105)*

Note: You may assume that duplicates do not exist in the tree.

递归: $\boxed{\text{时间复杂度O}(nlgn)\text{，空间复杂度O}(lgn)}$
前序序列的第一个元素肯定是树的根节点，而且使用这个值在中序序列中查找，找到的那个位置之前的必然是左子树，之后的必然是右子树，所以根据这个特点就可以很容易的使用递归的做法解题。

```
TreeNode* detail(vector<int> &inorder, vector<int> &preorder,
                int in_start, int in_end, int pre_start, int pre_end){
//inorder[in_start, in_end], preorder[pre_start, pre_end]
    if(in_start > in_end || pre_start > pre_end) return NULL;
    int val = preorder[pre_start];

    TreeNode* father = new TreeNode(val);
    TreeNode *left = NULL, *right = NULL;

    int in_pos = in_start, pre_pos = pre_start;
    for(; in_pos <= in_end; in_pos++, pre_pos++){
        if(val == inorder[in_pos])
            break;
    }

    left = detail(inorder, preorder, in_start, in_pos-1, pre_start+1, pre_pos);
    right = detail(inorder, preorder, in_pos+1, in_end, pre_pos+1, pre_end);

    father->left = left;
    father->right = right;

    return father;
}

TreeNode *buildTree(vector<int> &preorder, vector<int> &inorder) {
    int size = inorder.size();
    if(size == 0) return NULL;
```

```
        return detail(inorder, preorder, 0, size - 1, 0,
            size - 1);
}
```

2.2.2 Construct Binary Tree from Postorder and Inorder Traversal

问题: Given postorder and inorder traversal of a tree, construct the binary tree. *(leetcode 106)*

Note: You may assume that duplicates do not exist in the tree.

递归： $\boxed{\text{时间复杂度O}(nlgn)\text{，空间复杂度O}(lgn)}$
后序序列的最后一个元素肯定是树的根节点，而且使用这个值在中序序列中查找，找到的那个位置之前的必然是左子树，之后的必然是右子树，所以根据这个特点就可以很容易的使用递归的做法解题。

```
TreeNode* recursion(vector<int> &inorder, vector<int> &
    postorder,
                        int s_in, int e_in, int s_po, int
                            e_po){
    //inorder[s_in, e_in],postorder[s_po, e_po]
    if(s_in > e_in)     return NULL;
    if(s_in == e_in) return (new TreeNode(inorder[s_in])
        );
    TreeNode *root = new TreeNode(postorder[e_po]);
    int split_in, split_po;
    for(split_in = s_in; split_in <= e_in; split_in++){
        if(postorder[e_po] == inorder[split_in])
            break;
    }
    split_po = s_po + split_in - s_in;
    root->left = recursion(inorder, postorder, s_in,
        split_in - 1, s_po , split_po - 1 );
    root->right = recursion(inorder, postorder, split_in
         + 1, e_in, split_po, e_po - 1);
    return root;
}

TreeNode *buildTree(vector<int> &inorder, vector<int> &
    postorder) {
    if(inorder.size() != postorder.size() || inorder.
        size() == 0)
        return NULL;
    return recursion(inorder, postorder, 0, inorder.size
        () - 1, 0, postorder.size() - 1);
}
```

2.2.3 Convert Sorted Array to Binary Search Tree

问题: Given an array where elements are sorted in ascending order, convert it to a height balanced BST. *(leetcode 108)*

递归: $\boxed{\text{时间复杂度O(n), 空间复杂度O(n)}}$

建立平衡的二叉树，那么我们每次取数组的中间位置那个元素为根节点，然后它之前的部分创建左子树，之后的部分创建右子树，那么很容易就可以使用递归实现.

```
TreeNode* dfs(vector<int> &num, int start, int end){
    if(start > end)    return NULL;
    if(start == end)    return (new TreeNode(num[start]));
    int mid = (start + end) / 2;
    TreeNode* root = new TreeNode(num[mid]);
    root->left = dfs(num, start, mid - 1);
    root->right = dfs(num, mid + 1, end);
    return root;
}

TreeNode *sortedArrayToBST(vector<int> &num) {
    int size = num.size();
    return dfs(num, 0, size - 1);
}
```

因为是数组，所以可以很方便的找到中位数，但是如果是链表，则需要使用一些小技巧

2.2.4 Convert Sorted List to Binary Search Tree

问题: Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST. *(leetcode 109)*

递归: $\boxed{\text{时间复杂度O(n), 空间复杂度O(n)}}$

前面使用数组构造BST树，我们可以看到每次需要求出它的中间的那个数，然后以它创建根节点，但是对于有序链表来说，找到中位数起码要花O(n)时间，那么这样算下来整个程序需要O($nlgn$)的时间！这似乎和数组的O(n)差别比较大。我们可以想一下，摒弃这种自上而下的思维，来一次自下而上的方法：我们先构建左子树，构建完了之后访问的最后一点节点是不是就是根节点的前驱？这样我们记下这个前驱，然后一记next是不是就求出我们梦寐以求的中位数那个节点！然后再next一下，构建右子树，看看发生了什么？我们一边建树一边就得到中间节点，所以就省掉了找中间节点的那个时间！

```
TreeNode* DFS(ListNode* &head, int n){
    if(n == 0)    return NULL;
    TreeNode *root;
    if(n == 1){
        root = new TreeNode(head->val);
        head = head->next;
        return root;
    }
    TreeNode *left = DFS(head, n/2);// head travel the
       list
    root = new TreeNode(head->val);
    head = head->next;
    TreeNode *right = DFS(head, n - 1 - n/2 );
    root->left = left;
    root->right = right;
    return root;
}

TreeNode *sortedListToBST(ListNode *head) {
    if(!head)    return NULL;
    int count = 0;
    ListNode *pos = head;
    while(pos){//compute the length of the list
        pos = pos->next;
        count++;
    }
    return DFS(head, count);
}
```

这里使用自下而上的做法很具有普遍性，我们从上面看得不到的东西，从下面可以积累到

2.2.5 Unique Binary Search Trees

问题: Given n, how many structurally unique BST's (binary search trees) that store values 1...n? *(leetcode 96)*
这道题虽然是一个二叉树的问题，但是其实是数学归纳法的问题，我们可以得到递推公式:

$$\begin{aligned} S(0) &= 1 \\ S(n) &= \sum_{i=0}^{n-1} S(i) * S(n-1-i) \end{aligned} \tag{2.1}$$

这是因为对于有n个元素的BST(1,2,...,n)，我们考虑由谁来作为根节点，如果以i+1为根节点，那么左子树为(1,2,...,i),右子树为(i+2,i+3,...,n),所以可以得到上面的递推关系.

迭代： $\boxed{\text{时间复杂度O}(n^2)\text{，空间复杂度O(n)}}$

```
int numTrees(int n) {
    if(!n)    return 0;
    vector<int> num(n+1, 0);
    num[1] = 1;
    num[2] = 2;
    for(int i = 3; i <= n; i++){
        num[i] = 2*num[i-1];
        for(int j = 1; j < i; j++){
            num[i] = num[i] + num[j]*num[i-1-j];
        }
    }
    return num[n];
}
```

2.2.6 Unique Binary Search Trees II

问题: Given n, generate all structurally unique BST's (binary search trees) that store values 1...n. *(leetcode 95)*

: $\boxed{\text{时间复杂度 TODO , 空间复杂度 TODO}}$
每次对于构造序列(i,i+1,...,j),切分j-i+1次，然后分别递归构造.

```
vector<TreeNode*> generate(int start, int end){
    if(start == end){
        return vector<TreeNode*>(1, new TreeNode(start))
            ;
    }
    vector<TreeNode*> data;
    if(start > end){
        data.push_back(NULL);
        return data;
    }
    for(int i = start; i <= end; i++){
        TreeNode* root;
        vector<TreeNode*> left = generate(start, i - 1);
        vector<TreeNode*> right = generate(i + 1, end);
        for(int j = 0; j < left.size(); j++){
            for(int k = 0; k < right.size(); k++){
                root = new TreeNode(i);
                root->left = left[j];
                root->right = right[k];
                data.push_back(root);
            }
        }
    }
    return data;
}

vector<TreeNode*> generateTrees(int n) {
    return generate(1, n);
}
```

2.3 二叉树的属性

2.3.1 Validate Binary Search Tree

问题: Given a binary tree, determine if it is a valid binary search tree (BST). *(leetcode 98)*
判断一个二叉树是否是合法的BST，我们可以想到BST树的中序序列是非减序列，于是我们可以使用中序遍历这颗二叉树，在遍历的过程中查看是否有反常的数据.
当然，根据上面说的三种中序遍历的方法，这里同样有三种解法.

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
bool dfs(TreeNode *root, int& up){
    if(!root)    return true;
    if(root->left){
        bool left =  dfs(root->left, up);
        if(!left) return false;
    }
    if(root->val <= up && (MIN || up != (-1)<<31)){
        return false;
    }
    if(root->val == (-1)<<31)
        MIN = true;
    up = root->val;
    if(root->right){
        bool right = dfs(root->right, up);
        if(!right)    return false;
    }
    return true;
}

bool isValidBST(TreeNode *root) {
    if(!root)    return true;
    int up = (-1)<<31;
    MIN = false;
    return dfs(root, up);
}
```

这里可以看到一些边界条件的判断，显得有点复杂，其实就是简单的中序遍历

栈式迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
bool isValidBST(TreeNode *root) {
    if(!root)    return false;
    stack<TreeNode*> s;
    TreeNode *p = root;
    s.push(root);
    while(p->left){
        s.push(p->left);
        p = p->left;
    }
    int last = p->val;
    s.pop();
    if(p->right){
        s.push(p->right);
        p = p->right;
        while(p->left){
            s.push(p->left);
            p = p->left;
        }
    }
    while(!s.empty()){
        p = s.top();
        s.pop();
        if(last >= p->val) return false;
        last = p->val;
        if(p->right){
            s.push(p->right);
            p = p->right;
            while(p->left){
                s.push(p->left);
                p = p->left;
            }
        }
    }
    return true;
}
```

Mirror迭代： $\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

这里使用Mirror建立线索然后进行中序遍历，在中序遍历的同时进行判断

```
bool isValidBST(TreeNode *root) {
    if(!root)    return true;
    TreeNode *curr = root, *next;
    int last = INT_MIN;
    bool isFirst = true;
    bool ret = true;
    while(curr){
        if(!curr->left){
            if(!isFirst && curr->val <= last){
                ret = false;
            }
            if(isFirst)
                isFirst = false;
            last = curr->val;
            curr = curr->right;
```

```
                continue;
            }
            next = curr->left;
            while(next->right){
                if(next->right == curr)    break;
                next = next->right;
            }
            if(next->right == curr){
                next->right = NULL;
                if(!isFirst && curr->val <= last){
                    ret = false;
                }
                if(isFirst)
                    isFirst = false;
                last = curr->val;
                curr = curr->right;
            }else{
                next->right = curr;
                curr = curr->left;
            }
        }
        return ret;
}
```

有了*Mirror*算法，是不是你已经爱上它了，再也不用栈这么麻烦了,不过有一点需要注意的是一旦你使用*Mirror*算法，那么必须保证把整个树全遍历一遍，不能中途退出，因为那样树的结构被改变了

2.3.2 Symmetric Tree

问题: Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center). *(leetcode 101)*
这是求证树是不是自身Mirror(成镜像).

队列：$\boxed{\text{时间复杂度O(n)，空间复杂度O(w), w为树的最大宽度}}$

```
bool isSymmetric(TreeNode* root){
    if(!root) return true;
    deque<TreeNode*> left(1, root->left), right(1, root
        ->right);
    TreeNode *l, *r;
    while(!left.empty() && !right.empty()){
        l = left.front();
        r = right.front();
        left.pop_front();
        right.pop_front();
        if(!l && !r)    continue;
        if(!l || !r || l->val != r->val)    return false
            ;
            left.push_back(l->left);
            left.push_back(l->right);
            right.push_back(r->right);
            right.push_back(r->left);
        }
        return true;
}
```

递归： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
这里是把一棵树的对称问题看成两棵树的对称问题

```
bool recursion(TreeNode* root, TreeNode* symm){
    if(!root && !symm)
        return true;
    if(!root || !symm)    return false;
    if(root->val != symm->val)    return false;
    if(root == symm)    return recursion(root->left,
        symm->right);
    return recursion(root->left, symm->right) &&
        recursion(root->right, symm->left);
}

bool isSymmetric(TreeNode* root){
    if(!root) return true;
    return recursion(root, root);
}
```

这里还可以延伸出一个问题： 求一个二叉树的镜像树

2.3.3 Same Tree

问题: Given a binary tree, determine if it is height-balanced. *(leetcode 110)*

递归： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
先判断左子树是否高度平衡并返回左子树高度，再判断右子树是否高度平衡，再返回右子树高度，根据左右子树高度再判断当前树是否平衡.

```
bool dfs(TreeNode *root, int &hight){
    if(!root){
        hight = 0;
        return true;
    }
    int left, right;
    bool is_left = dfs(root->left, left);
    bool is_right = dfs(root->right, right);
    hight = left > right? left + 1 : right + 1;
    return is_left && is_right && (abs(left - right) <
        2);
}

bool isBalanced(TreeNode *root) {
    int hight;
    return dfs(root, hight);
}
```

2.3.4 Maximum Depth of Binary Tree

问题: Given a binary tree, find its maximum depth.*(leetcode 104)*
从根节点来看，它的深度就是左右子树深度较大的那个+1,所以很自然的想到递归

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
递归代码十分简洁

```
int maxDepth(TreeNode *root){
    if(!root)   return 0;
    int left = maxDepth(root->left);
    int right = maxDepth(root->right);
    return left < right? right + 1 : left + 1;
}
```

除了递归，其实这道题能不能用迭代的做法呢？答案是肯定的，最初你可能会想到用两个栈，一个栈存放节点，一个栈存放深度，其实可以把这个两者打包成一个pair，使用一个栈就可以啦

迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
int maxDepth(TreeNode *root) {
    if(!root)    return 0;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, 1));
    pair<TreeNode*, int> curr;
    int result = INT_MIN;
    while(!s.empty()){
        curr = s.top();
        s.pop();
        if(!curr.first->left && !curr.first->right){
            if(result < curr.second)
                result = curr.second;
            continue;
        }
        if(curr.first->left){
            s.push(make_pair(curr.first->left, curr.
                second + 1));
        }
        if(curr.first->right){
            s.push(make_pair(curr.first->right, curr.
                second + 1));
        }
    }
    return result;
}
```

这种自上而下的过程(比如节点深度)，用迭代实现就很简单，但是如果是自下而上的过程呢？好像很复杂，你有什么好的想法？

2.3.5 Minimum Depth of Binary Tree

问题: Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node. *(leetcode 111)*

递归: $\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$
自下而上的递归，非常的简单

```
int minDepth(TreeNode *root) {
    if(!root)    return 0;
    if(!root->left && !root->right)    return 1;
    if(!root->left)
        return minDepth(root->right) + 1;
    if(!root->right)
        return minDepth(root->left) + 1;
    return min(minDepth(root->left), minDepth(root->
        right)) + 1;
}
```

DFS: $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
这也是递归，但是是一种自上而下的递归，可以进行剪枝而不必把整个树都访问一遍

```
void dfs(TreeNode *root, int &result, int depth){
    if(result < depth + 1) return;
    if(!root->left && !root->right){
        result = depth + 1;
        return;
    }
    if(root->left)
        dfs(root->left, result, depth + 1);
    if(root->right)
        dfs(root->right, result, depth + 1);
}
```

```

    int minDepth(TreeNode *root) {
    if(!root)    return 0;
    int result = INT_MAX;
    dfs(root, result, 0);
    return result;
}
```

迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

同样我们也可以剪枝

```
int minDepth(TreeNode *root) {
    if(!root)    return 0;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, 1));
    int     result = -((1<<31) + 1);
    TreeNode *node;
    int depth;
    while(!s.empty()){
        node = s.top().first;
        depth = s.top().second;
        s.pop();
        if(result < depth)    continue;
        if(!node->left && !node->right)
            result = depth;

        if(node->left )
            s.push(make_pair(node->left, depth + 1));
        if(node->right)
            s.push(make_pair(node->right, depth + 1));
    }
    return result;
}
```

2.3.6 Path Sum

问题: Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. *(leetcode 112)*

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

先减去当前节点的值，剩余的值再分别递归求解左右子树

```
bool hasPathSum(TreeNode *root, int sum) {
    if(root == NULL) return false;
    int val = root->val;
    sum = sum - val;
    if(sum == 0 && !root->left && !root->right)
        return true;

    bool left = false;
    if(root->left){
        left = hasPathSum(root->left, sum);
    }
    if(left) return true;
    if(root->right)
        return hasPathSum(root->right, sum);
    return false;
}
```

仿照最大/小深度问题，可以在迭代栈加上附加路径和这个信息，那么实现起来就非常简单了

迭代： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
bool hasPathSum(TreeNode *root, int sum) {
    if(!root)    return false;
    stack<pair<TreeNode*, int> > s;
    s.push(make_pair(root, root->val));
    TreeNode *node;
    int e;
    while(!s.empty()){
        node = s.top().first;
        e = s.top().second;
        s.pop();
        if(!node->left && !node->right && e == sum)
            return true;
        if(node->left)
            s.push(make_pair(node->left, e + node->left
                ->val));
        if(node->right)
            s.push(make_pair(node->right, e + node->
                right->val));
    }
    return false;
}
```

可以看得出，迭代栈加入附件信息是一个常用的技巧，需要深刻理解和掌握. 另外，这道题还可以变形为求二叉树最远两个节点距离的问题，这就相当于把每个节点的权值都设为1，方法和这道题一样，而且还要比它简单.

2.3.7 Path Sum II

问题: Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum. *(leetcode 113)*

这题是Path Sum问题的延续，解法其实是一模一样的.

递归： $\boxed{\text{时间复杂度O(n)，空间复杂度O(n)}}$
这里有个技巧需要注意的就是，可以使用一个引用参数cur来记录跟节点到当前节点这条路径中的所有值，我们在进入某个节点后cur要push这个节点，在离开这个节点后就要pop这个节点

```
vector<vector<int> > pathSum(TreeNode *root, int sum){
    vector<int> cur;
    vector<vector<int> > result;
    recursion(root, cur, result, sum);
    return result;
}

void recursion(TreeNode *root, vector<int> &cur, vector<
    vector<int> > &result, int sum){
    if(!root)    return;
    cur.push_back(root->val);
    if(!root->left && !root->right && root->val == sum)
        result.push_back(cur);
    if(root->left)
        recursion(root->left, cur, result, sum - root->
            val);
    if(root->right)
        recursion(root->right, cur, result, sum - root->
            val);
    cur.pop_back();
}
```

我们知道Path Sum有迭代的做法，但是那种通过栈附加信息的做法对于我们这题还要求求出路径的问题不是很适合，所以就没有写出这种做法了，你有什么好想法么?

2.3.8 Binary Tree Maximum Path Sum

问题: Given a binary tree, find the maximum path sum.The path may start and end at any node in the tree. *(leetcode 124)*

递归： $\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
这里先说约定一些叫法:

1 最大路径和: max{树中任意两节点之间的路径和}

2 最大到根路径和: max{树中任意节点到根节点的路径和}

这里我们知道，这个路径肯定是有个拐点，那么这个拐点肯定是某个子树的根节点，所以我们只要递归的求每个子树的过根最大路径和，然后在这些路径和中选出最大的那个就可以了.

过根最大路径和怎么求呢？可以先分别求出左右子树的最大到根路径和，根据这个两个路径和与根节点则可以求出当前树的最大路径和.

```
int dfs(TreeNode *root, int &result){
    if(!root)    return 0;
    if(!root->left && !root->right){
        if(result < root->val)
            result = root->val;
        return root->val > 0? root->val : 0;
    }
    int left = 0, right = 0;
    if(root->left){
        left = dfs(root->left, result);
    }
    if(root->right){
        right = dfs(root->right, result);
    }
    int cur = root->val + left + right;
    if(result < cur)
        result = cur;
    int add = left > right ? left : right;
    if(add < 0)    return root->val > 0? root->val : 0;
    return root->val + add > 0 ? root->val + add : 0;
}

int maxPathSum(TreeNode *root) {
    int result = INT_MIN;
    dfs(root, result);
    return result;
}
```

2.3.9 Sum Root to Leaf Numbers

问题: Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.An example is the root-to-leaf path 1-¿2-¿3 which represents the number 123.Find the total sum of all root-to-leaf numbers. *(leetcode 129)*
其实这还是属于那种自上而下带有附加信息过来的问题，和最大/小深度问题是一样的.

递归: $\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$
从上面传来上面路径产生的数字，类似于传来上面路径的深度(最大深度问题)和上面路径的和(Path Sum问题)

```
void dfs(TreeNode *root, int track, int &sum){
    if(!root)    return;
    track = track*10 + root->val;
    if(!root->left && !root->right){
        sum = sum + track;
    }
    if(root->left)
        dfs(root->left, track, sum);
    if(root->right)
        dfs(root->right, track, sum);
}

int sumNumbers(TreeNode *root){
    int sum = 0;
    int track = 0;
    dfs(root, track, sum);
    return sum;
}
```

同样，类似与Path Sum这类问题，可以常用附加路径信息的栈实现迭代，具体做法就是当前节点左右孩子入栈时候当前节点的附加信息值*10加左右孩子节点的值然后作为左右孩子的附加信息值，这里就不再写出.

2.3.10 Least Common Ancest

问题: Given a binary tree, return the least common ancest of two nodes.

递归： $\boxed{\text{时间复杂度O(n), 空间复杂度O}(lgn)}$

```
TreeNode * NearestCommAncestor(TreeNode *root, TreeNode
   *node1, TreeNode *node2){
    if(!root || !node1 || !node2)  return NULL;
    if(node1 == root || node2 == root)
        return root;
    if(!root->left)
        return NearestCommAncestor(root->right, node1,
           node2);
    if(!root->right)
        return NearestCommAncestor(root->left, node1,
           node2);
    TreeNode *left, *right;
    left = NearestCommAncestor(root->left, node1, node2)
       ;
    right = NearestCommAncestor(root->right, node1,
       node2);
    if(left && right)  return root;
    return left? left : right;
}
```

2.4 其他

2.4.1 Flatten Binary Tree to Linked List

问题: Given a binary tree, flatten it to a linked list in-place by the pre-order. *(leetcode 114)*
这是一个基于先序遍历的问题，所以可以使用递归和迭代的方法.

递归：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$

```
void flatten(TreeNode *root) {
    TreeNode *tail;
    recursion(root, tail);
}

TreeNode* recursion(TreeNode *root, TreeNode* &tail){
    if(!root)   return NULL;
    TreeNode *next = NULL;
    tail = root;
    if(root->left)
        next = recursion(root->left, tail);
    if(root->right)
        tail->right = recursion(root->right, tail);
    root->left = NULL;
    if(next)
        root->right = next;
    return root;
}
```

迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O}(lgn)}$
这里就是完完全全的迭代版前序遍历,这里使用了栈，同样你也可以使用Mirror算法.

```
void flatten(TreeNode *root) {
    if(!root)   return;
    stack<TreeNode*> s;
    s.push(root);
    TreeNode *last = NULL, *cur;
    while(!s.empty()){
        cur = s.top();
        s.pop();
        if(last)
            last->right = cur;
        if(cur->right)
            s.push(cur->right);
        if(cur->left)
            s.push(cur->left);
        cur->left = NULL;
        last = cur;
    }
    last->right = NULL;
}
```

2.4.2 Populating Next Right Pointers in Each Node

问题: Given a binary tree:

```
struct TreeLinkNode {
    TreeLinkNode *left;
    TreeLinkNode *right;
    TreeLinkNode *next;
}
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.
Note:
You may only use constant extra space.
You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children). *(leetcode 116)*
其实就是一个很简单的BFS过程.

迭代: $\boxed{\text{时间复杂度O(n), 空间复杂度O(1)}}$
因为是满二叉树，所以每次在上一层建立这一层的next，然后再到这一层来,这样就不需要队列，使用常数的空间复杂度.

```
void connect(TreeLinkNode *root) {
    if(!root)    return;
    TreeLinkNode *cur = root, *next;
    cur->next = NULL;
    while(cur->left){
        next = cur->left;
        while(cur){
            cur->left->next = cur->right;
            cur->right->next = cur->next? cur->next->
                left : NULL;
            cur = cur->next;
        }
        cur = next;
    }
}
```

2.4.3 Populating Next Right Pointers in Each Node II

问题: Follow up for problem "Populating Next Right Pointers in Each Node".What if the given tree could be any binary tree? Would your previous solution still work? **Note**: You may only use constant extra space. *(leetcode 117)*

迭代：$\boxed{\text{时间复杂度O(n)，空间复杂度O(1)}}$

这里其实和上一题一样，只不过多了一些判断条件.

```
void connect(TreeLinkNode *root) {
    if(!root)    return;
    TreeLinkNode *cur = root, *next, *last;
    cur->next = NULL;
    do{
        next = NULL;
        while(cur){
            if(cur->left){
                if(next){
                    last->next = cur->left;
                    last = last->next;
                }
                else{
                    last = cur->left;
                    next = last;
                }
            }
            if(cur->right){
                if(next){
                    last->next = cur->right;
                    last = last->next;
                }
                else{
                    last = cur->left;
                    next = last;
                }
            }
            cur = cur->next;
        }
        last->next = NULL;
        cur = next;
    }while(cur);
}
```