

树

树

二叉树

- 94. 二叉树的中序遍历
- 144. 二叉树的前序遍历
- 145. 二叉树的后序遍历
- 114. 二叉树展开为链表
- 105. 从前序与中序遍历序列构造二叉树
- 106. 从中序与后序遍历序列构造二叉树
- 102. 二叉树的层序遍历
- 103. 二叉树的锯齿形层序遍历
- 107. 二叉树的层序遍历 II
- 116. 填充每个节点的下一个右侧节点指针
- 117. 填充每个节点的下一个右侧节点指针 II
- 100. 相同的树
- 101. 对称二叉树
- 226. 翻转二叉树
- 104. 二叉树的最大深度
- 111. 二叉树的最小深度
- 110. 平衡二叉树
- 112. 路径总和
- 113. 路径总和 II
- 257. 二叉树的所有路径
- 129. 求根节点到叶节点数字之和
- 199. 二叉树的右视图
- 513. 找树左下角的值
- 222. 完全二叉树的节点个数
- 236. 二叉树的最近公共祖先
- 437. 路径总和 III
- 508. 出现次数最多的子树元素和
- 863. 二叉树中所有距离为 K 的结点
- 1367. 二叉树中的列表

二叉搜索树

概念

参考

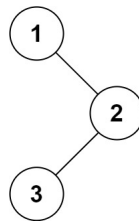
- 96. 不同的二叉搜索树
- 108. 将有序数组转换为二叉搜索树
- 95. 不同的二叉搜索树 II
- 98. 验证二叉搜索树
- 230. 二叉搜索树中第K小的元素
- 235. 二叉搜索树的最近公共祖先
- 449. 序列化和反序列化二叉搜索树
- 450. 删除二叉搜索树中的节点
- 501. 二叉搜索树中的众数

二叉树

94. 二叉树的中序遍历

给定一个二叉树的根节点 `root`，返回它的 中序 遍历。

示例 1：



输入: `root = [1,null,2,3]`

输出: `[1,3,2]`

示例 2：

输入: `root = []`

输出: `[]`

- 递归
 - 思路
 - 前序遍历：当前->左->右
 - 中序遍历：左->当前->右
 - 后序遍历：左->右->当前
 - 题目要求中序遍历，即按照 当前->左->右 的顺序打印即可，递归函数实现如下
 - 终止条件：当前节点为空时
 - 函数内：递归调用左节点，打印当前节点，递归调用右节点
 - 代码

```
class Solution {
public:
    void dfs(TreeNode *root, vector<int> &res) {
        if (root == nullptr) { return; }
        dfs(root->left, res);
        res.push_back(root->val);
        dfs(root->right, res);
    }
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        dfs(root, res);
        return res;
    }
};
```

- 复杂度
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(h)$, h 是树的高度
- 迭代

- 思路

递归是系统自动使用 **栈** 保存了调用函数，迭代就是需要手动使用栈来模拟系统栈的调用过程。如下为系统递归调用的过程

```
dfs(root.left)
```

```

dfs(root.left)
    为null返回
打印节点
dfs(root.right)
    dfs(root.left)
        dfs(root.left)
            ...

```

递归的调用过程就是 **不断往左走，当左边无法走时，打印节点，转向右边，然后右边继续这个过程**，迭代实现就是手动使用栈模拟这个调用过程

- 代码

```

class Solution {
public:
    // 迭代
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        // 2. 中序遍历-非递归方式
        if (root == nullptr) {
            return res;
        }
        std::stack<TreeNode*> s;
        while (root != nullptr || !s.empty()) {
            if (root != nullptr) { // 存在左子树，则一直向左走
                s.push(root);
                root = root->left;
            } else { // 左子树不存在，则开始处理 中-右
                // 取出 中 节点 处理
                root = s.top();
                s.pop();
                res.push_back(root->val);

                // 更新右子树，等待处理
                root = root->right;
            }
        }
        return res;
    }
};

```

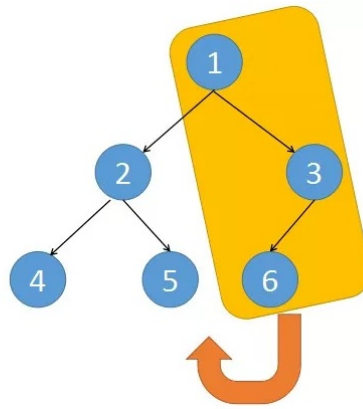
- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(h)$, h 是树的高度

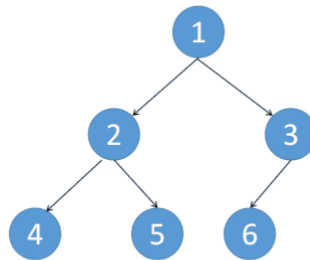
- Morris遍历

- 思路

用递归和迭代的方式都使用了辅助空间，而Morris遍历的优点是不使用任何辅助空间，缺点是改变了树结构，强行把一颗二叉树改为一段链表结构



如上图，将黄色区域部分挂到节点5的右子树上，接着把根节点为2的子树（节点2和节点5）挂到节点4的右子树上，这样整棵树基本上就成了一个链表了，后边就是链表遍历



◦ 代码

```
class Solution {
public:
    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> res;
        while (root != nullptr) {
            if (root->left != nullptr) {
                TreeNode *pre = root->left;
                while (pre->right != nullptr) {
                    pre = pre->right;
                }
                pre->right = root;
                TreeNode *tmp = root;
                root = root->left;
                tmp->left = nullptr;
            } else {
                res.push_back(root->val);
                root = root->right;
            }
        }
        return res;
    }
};
```

◦ 复杂度

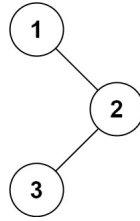
- 时间复杂度： $O(n)$ ，找到每个前驱结点的复杂度是 $O(n)$ ，因为 n 个节点的二叉树有 $n-1$ 条边，每条边只可能使用2次（一次定位到节点，一次找到前驱结点），故时间复杂度为 $O(n)$
- 空间复杂度： $O(1)$

- 参考
 - [动画演示+三种实现 94. 二叉树的中序遍历](#)

144. 二叉树的前序遍历

给你二叉树的根节点 `root`，返回它节点值的 **前序** 遍历。

示例 1:



输入: `root = [1,null,2,3]`

输出: `[1,2,3]`

示例 2:

输入: `root = []`

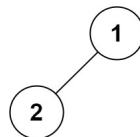
输出: `[]`

示例 3:

输入: `root = [1]`

输出: `[1]`

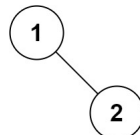
示例 4:



输入: `root = [1,2]`

输出: `[1,2]`

示例 5:

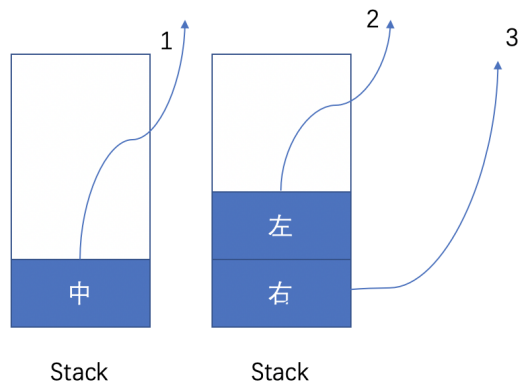


输入: `root = [1,null,2]`

输出: `[1,2]`

- 前言: 注意分清前、中、后遍历
 - 前序: 中、左、右
 - 中序: 左、中、右
 - 后续: 左、右、中
- 递归: 略, 比较简单
- 迭代
 - 思路
 - 迭代解法本质上是模拟递归, 因为递归解法使用了系统栈, 所以迭代解法中常用stack来模拟系统栈
 - 前序遍历
 - 创建一个 `stack` 用来存放节点, 首先要打印根节点数据, 需要将根节点压栈, 打印
 - 之后打印左子树、右子树, 所以按顺序将右子树、左子树压栈

输出的顺序就是 中 左 右，符合先序遍历



https://blog.csdn.net/weixin_42322309

◦ 代码

```
// https://leetcode-cn.com/problems/binary-tree-preorder-traversal/solution/leetcodesuan-fa-xiu-lian-dong-hua-yan-shi-xbian-2/
// 迭代解法
class Solution {
public:
    // 迭代
    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == nullptr) {
            return res;
        }
        stack<TreeNode*> s;
        while (root != nullptr || !s.empty()) {
            if (root != nullptr) {
                s.push(root);
                res.push_back(root->val);
                root = root->left;
            } else {
                root = s.top();
                s.pop();
                root = root->right;
            }
        }
        return res;
    }
};
```

◦ 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$
- morris遍历: 略吧, 后续有时间再看

145. 二叉树的后序遍历

给定一个二叉树，返回它的 后序 遍历。

示例:

输入: [1,null,2,3]

1
\
2
/
3

输出: [3,2,1]

进阶: 递归算法很简单, 你可以通过迭代算法完成吗?

- 递归: 略, 比较简单
- 迭代
 - 思路
 - 后序遍历的迭代方法同前、中序一样, 都是模拟系统栈
 - 需要注意, 后续遍历中, 根节点会入栈两次, 第一次是根节点->左子树入栈, 第二次是根节点->右子树入栈。所以相应的也存在两种情况弹出某根节点, 这种情况需要用 prev 指针标记, 只有当第二种情况下, 即上一个处理的节点是右子树, 才表示该根节点的左右子树都已遍历完, 根节点才能加入到结果集中
 - 一是其左子树遍历完, 会弹出根节点
 - 二是其右子树遍历完, 会弹出根节点、
 - 代码

```
// https://leetcode-cn.com/problems/binary-tree-postorder-traversal/solution/er-cha-shu-de-hou-xu-bian-li-by-leetcode-solution/
class Solution {
public:
    // 广搜
    vector<int> postorderTraversal(TreeNode* root) {
        vector<int> res;
        if (root == nullptr) { return res; }
        stack<TreeNode*> s;
        TreeNode *prev = nullptr;
        while (root != nullptr || !s.empty()) {
            if (root != nullptr) { // root、左子树入栈 存在左子树, 则一直向左走
                s.push(root);
                root = root->left;
            } else {
                root = s.top();
                s.pop();
                // 左、右子树已遍历 (右子树为空 || 右子树已遍历)
                if (root->right == nullptr || root->right == prev) {
                    res.push_back(root->val);
                    prev = root; // 表示右子树已遍历
                    root = nullptr; // 保证不重复入栈左子树
                } else { // root、右子树入栈 遍历右子树, 将root重新入栈
                    s.push(root);
                    root = root->right;
                }
            }
        }
        return res;
    }
};
```

- 复杂度
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(n)$

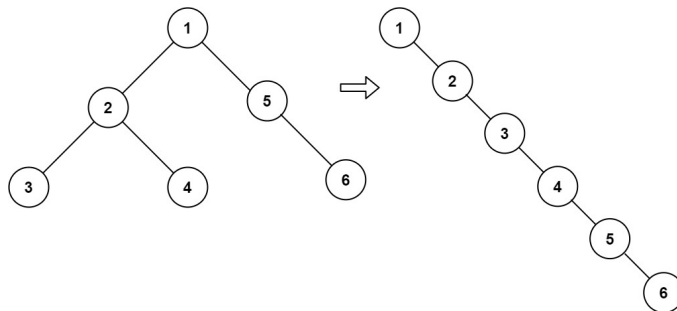
114. 二叉树展开为链表

给你二叉树的根结点 `root`，请你将它展开为一个单链表：

展开后的单链表应该同样使用 `TreeNode`，其中 `right` 子指针指向链表中下一个结点，而左子指针始终为 `null`。

展开后的单链表应该与二叉树 先序遍历 顺序相同。

示例 1：



输入: `root = [1,2,5,3,4,null,6]`

输出: `[1,null,2,null,3,null,4,null,5,null,6]`

示例 2：

输入: `root = []`

输出: `[]`

示例 3：

输入: `root = [0]`

输出: `[0]`

- 前驱结点
 - 思路
 - 将二叉树展开为单链表后，单链表中的节点顺序即为二叉树的前序遍历访问各节点的顺序。前序遍历顺序是 根节点、左子树、右子树。若某节点左子树为空，则该节点不需要展开。若某节点左子树不为空，则该节点的左子树中的最后一个节点（即左子树中最右边的节点，也就是前驱结点）被访问后，就该访问该节点的右子树。问题转化为 **寻找当前节点的前驱结点**
 - 具体做法
 - 对于当前节点，如果其左子树不为空，则在左子树中找到最右边的节点，作为前驱结点，
 - 将当前节点的右子节点赋给前驱结点的右子节点，
 - 然后将当前节点的左子节点赋给当前节点的右子节点。
 - 并将当前节点的左子节点设为空。
 - 对当前节点处理结束后继续处理链表中的下一个节点，直到所有节点都处理结束
 - 代码

```
// https://leetcode-cn.com/problems/flatten-binary-tree-to-linked-list/solution/er-cha-shu-zhan-kai-wei-lian-biao-by-leetcode-solu/  
class Solution {  
public:  
    void flatten(TreeNode* root) {
```



```

TreeNode *cur = root;
while (cur != nullptr) {
    if (cur->left != nullptr) {
        auto next = cur->left;
        auto pre = next;
        while (pre->right != nullptr) {
            pre = pre->right;
        }
        pre->right = cur->right;
        cur->left = nullptr;
        cur->right = next;
    }
    cur = cur->right;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。展开为单链表的过程中, 需要对每个节点访问一次
- 空间复杂度: $O(1)$

105. 从前序与中序遍历序列构造二叉树

根据一棵树的前序遍历与中序遍历构造二叉树。

注意:

你可以假设树中没有重复的元素。

例如, 给出

前序遍历 preorder = [3,9,20,15,7]

中序遍历 inorder = [9,3,15,20,7]

返回如下的二叉树:

```

3
 / \
9  20
 /  \
15  7

```

- 深搜

- 思路

对于任意一棵树而言, 前序遍历总是 根节点, [左子树前序遍历结果], [右子树遍历结果], 而中序遍历总是 [左子树中序遍历结果], 根节点, [右子树中序遍历结果]

只要在中序遍历中定位到根节点, 则可知道左右子树的节点数目, 并能对应到前序遍历中; 然后分别递归构造左子树和右子树, 再分别于当前节点的左右指针关联

- 细节

在中序遍历中对根节点进行定位时, 一种简单方法是直接扫描整个中序遍历的结果找到根节点, 这样时间复杂度较高。可以使用哈希表快速定位根节点, 只需 $O(1)$ 时间复杂度即可

- 代码

```

// 参考 https://leetcode-cn.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/solution/cong-qian-xu-yu-zhong-xu-bian-li-xu-lie-gou-zao-9/

```

```

class Solution {
public:
    unordered_map<int, int> lookup;

    TreeNode *helper(vector<int> &preorder, int pre_start, int pre_end,
vector<int> &inorder, int in_start, int in_end) {
        if (pre_start > pre_end) { return nullptr; }

        // 前序遍历中的第一个节点就是根节点
        int root_val = preorder[pre_start];

        // 在中序遍历中定位根节点所在下标
        int root_idx_inorder = lookup[root_val];

        // 建立根节点
        TreeNode *root = new TreeNode(root_val);

        // 计算左子树节点数目
        int left_nodes_num = root_idx_inorder - in_start;

        // 递归构造左子树，先序遍历中「从 左边界+1 开始的 left_nodes_num」个元素
        // 就对应了中序遍历中「从 左边界 开始到 根节点定位-1」的元素
        root->left = helper(preorder, pre_start + 1, pre_start +
left_nodes_num, inorder, in_start, root_idx_inorder - 1);

        // 递归构造右子树，先序遍历中「从 左边界+1+左子树节点数目 开始到 右边界」的
        // 元素就对应了中序遍历中「从 根节点定位+1 到 右边界」的元素
        root->right = helper(preorder, pre_start + left_nodes_num + 1,
pre_end, inorder, root_idx_inorder + 1, in_end);

        return root;
    }

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        // 构建 inorder中元素和下标的映射，方便后边快速查找
        for (int i = 0; i < inorder.size(); ++i) {
            lookup[inorder[i]] = i;
        }
        return helper(preorder, 0, preorder.size() - 1, inorder, 0,
inorder.size() - 1);
    }
};

```

◦ 复杂度

- 时间复杂度：\$O(n)\$，其中 \$n\$ 是树中的节点个数。
- 空间复杂度：\$O(n)\$，除去返回的答案需要的 \$O(n)\$ 空间之外，我们还需要使用 \$O(n)\$ 的空间存储哈希映射，以及 \$O(h)\$（其中 \$h\$ 是树的高度）的空间表示递归时栈空间。这里 \$h < n\$，所以总空间复杂度为 \$O(n)\$。

• 迭代（略）

106. 从中序与后序遍历序列构造二叉树

根据一棵树的中序遍历与后序遍历构造二叉树。

注意：

你可以假设树中没有重复的元素。

例如，给出

中序遍历 inorder = [9,3,15,20,7]
后序遍历 postorder = [9,15,7,20,3]
返回如下的二叉树:

```
3
 / \
9  20
 /  \
15  7
```

- 递归
 - 思路
 - 对于任意一棵树而言，后序遍历总是 [左子树前序遍历结果]，[右子树遍历结果]，根节点，而中序遍历总是 [左子树中序遍历结果]，根节点，[右子树中序遍历结果]
 - 只要在中序遍历中定位到根节点，则可知道左右子树的节点数目，并能对应到后序遍历中；然后分别递归构造左子树和右子树，再分别于当前节点的左右指针关联
 - 算法
 - 同题105，使用哈希表存储中序遍历 **元素：下标**，方便查找根节点在中序遍历数组中的位置
 - 定义递归函数 `helper(in_left, in_right)` 表示当前递归子树在中序遍历数组的左右边界，递归入口为 `helper(0, n - 1)`
 - 如果 `in_left > in_right`，说明子树为空，返回空节点
 - 选择后续遍历的最后一个节点作为根节点
 - 利用哈希表 `$O(1)` 查询当前根节点在中序遍历中的下标为 `index`，从 `in_left` 到 `index - 1` 为左子树，从 `index + 1` 到 `in_right` 为右子树
 - 根据后序遍历逻辑，递归创建右子树 `helper(index + 1, in_right)` 和左子树 `helper(in_left, index - 1)`。注意，这里真正构建的逻辑是先创建右子树，再创建左子树。
 - 代码

```
// 参考 https://leetcode-cn.com/problems/construct-binary-tree-from-inorder-and-postorder-traversal/solution/cong-zhong-xu-yu-hou-xu-bian-li-xu-lie-gou-zao-14/
class solution {
public:
    int root_idx_postorder;
    std::unordered_map<int, int> lookup;
    TreeNode *helper(int in_left, int in_right, vector<int> &inorder, vector<int> &postorder) {
        // 如果当前[in_left, in_right]范围为空，结束
        if (in_left > in_right) { return nullptr; }

        // 选择 root_idx_postorder位置的元素作为当前子树根节点
        int root_val = postorder[root_idx_postorder];
        // 往前移动根节点位置，供下一次递归找根节点
        --root_idx_postorder;

        TreeNode *root = new TreeNode(root_val);

        // 根据root位置分成左右两颗子树
        int root_idx_inorder = lookup[root_val];

        // 构建右子树
```

```

        root->right = helper(root_idx_inorder + 1, in_right, inorder,
postorder);
        // 构建左子树
        root->left = helper(in_left, root_idx_inorder - 1, inorder,
postorder);
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        // 构建中序遍历 [元素:下标] 哈希表
        for (int i = 0; i < inorder.size(); ++i) {
            lookup[inorder[i]] = i;
        }
        // 初始化根节点下标值
        root_idx_postorder = postorder.size() - 1;
        return helper(0, inorder.size() - 1, inorder, postorder);
    }
};

```

复杂度

- 时间复杂度: $O(n)$, 其中 n 是树中节点个数
- 空间复杂度: $O(n)$, 需要使用 $O(n)$ 空间存储哈希表, 以及 $O(h)$ 其中 h 是树的高度, 的空间表示递归时使用的占空间, 齐总 $h \leq n$, 所以总空间复杂度为 $O(n)$

102. 二叉树的层序遍历

给你一个二叉树, 请你返回其按 层序遍历 得到的节点值。 (即逐层地, 从左到右访问所有节点)。

示例:

二叉树: [3,9,20,null,null,15,7],

3

/ \

9 20

/ \

15 7

返回其层序遍历结果:

[

[3],

[9,20],

[15,7]

]

迭代

思路

- 修改广度优先搜索, 原先只取一个元素拓展, 这里需要取一层节点数目来拓展
 - 首先根节点入队
 - 当队列不为空
 - 求当前队列长度 s_i
 - 依次从队列中取 s_i 个节点, 并拓展其左右孩子, 进入下一次迭代

代码

```

class Solution {
public:
    // 广搜
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int n = q.size();
            vector<int> tmp;
            for (int i = 0; i < n; ++i) {
                TreeNode *cur = q.front();
                q.pop();

                if (cur->left != nullptr) {
                    q.push(cur->left);
                }

                if (cur->right != nullptr) {
                    q.push(cur->right);
                }

                tmp.push_back(cur->val);
            }
            res.push_back(tmp);
        }
        return res;
    }
};

```

- 复杂度
 - 时间复杂度：每个点进队出队各一次，故渐进时间复杂度为 $O(n)$ 。
 - 空间复杂度：队列中元素的个数不超过 n 个，故渐进空间复杂度为 $O(n)$ 。
- 参考
 - <https://leetcode-cn.com/problems/binary-tree-level-order-traversal/solution/er-cha-shu-de-ceng-xu-bian-li-by-leetcode-solution/>

103. 二叉树的锯齿形层序遍历

给定一个二叉树，返回其节点值的锯齿形层序遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：

给定二叉树 [3,9,20,null,null,15,7],

```

3
 / \
9  20
 /  \
15  7

```

返回锯齿形层序遍历如下：

```

[
  [3],
  [20,9],
  [15,7]
]

```

- 迭代

- 思路

类似于题102层次遍历，只不过本题要求锯齿形遍历，可以使用一个标志变量 `leftToRight` 表示该层是从左到右还是从右到左，并且对于每一层可使用双向队列 `deque` 来满足头插或尾插，遍历完该层后，将 `deque` 数据存到 `vector` 即可

- 代码

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if (root == nullptr) { return res; }
        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;
        while(!q.empty()) {
            int n = q.size();
            deque<int> dq;
            for (int i = 0; i < n; i++) {
                TreeNode *cur = q.front();
                q.pop();

                if (leftToRight) {
                    dq.push_back(cur->val);
                } else {
                    dq.push_front(cur->val);
                }

                if (cur->left != nullptr) {
                    q.push(cur->left);
                }

                if (cur->right != nullptr) {
                    q.push(cur->right);
                }
            }
            res.push_back(vector<int>(dq.begin(), dq.end()));
            leftToRight = !leftToRight;
        }
        return res;
    }
};
```

- 复杂度

- 时间复杂度： $O(N)$ ，其中 N 为二叉树的节点数。每个节点会且仅会被遍历一次。
- 空间复杂度： $O(N)$ 。我们需要维护存储节点的队列和存储节点值的双端队列，空间复杂度为 $O(N)$

- 参考

- <https://leetcode-cn.com/problems/binary-tree-zigzag-level-order-traversal/solution/bfshe-dfsliang-chong-jie-jue-fang-shi-by-184y/>

107. 二叉树的层序遍历 II

给定一个二叉树，返回其节点值自底向上的层序遍历。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

例如：

给定二叉树 [3,9,20,null,null,15,7],

```
3
 / \
9  20
 /  \
15  7
```

返回其自底向上的层序遍历为：

```
[
  [15,7],
  [9,20],
  [3]
]
```

- 广搜
 - 思路
 - 树的层次遍历可使用广搜实现，从根节点开始搜索，每次遍历同一层的全部节点，使用一个列表存储该层节点值
 - 若要求从上到下输出每层及该单支，做法是 每遍历完一层节点后，将存储该层节点值的列表添加到结果列表的尾部。
 - 而本题是要 **从下到上** 输出每层节点值，需要在遍历完一层节点后，将存储该层节点值的列表添加到结果列表的头部
 - C++可使用 `vector` 或 `list`，在使用尾部插入的方法得到从上到下的层次遍历列表后，再反转得到结果
 - 代码

```
// 参考 https://leetcode-cn.com/problems/binary-tree-level-order-traversal-ii/solution/er-cha-shu-de-ceng-ci-bian-li-ii-by-leetcode-solut/
class Solution {
public:
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> res;
        if (root == nullptr) { return res; }

        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int n = q.size();
            vector<int> level_nodes;
            for (int i = 0; i < n; ++i) {
                auto node = q.front();
                q.pop();

                level_nodes.push_back(node->val);
                if (node->left) { q.push(node->left); }
                if (node->right) { q.push(node->right); }
            }
            res.push_back(level_nodes);
        }
    }
};
```

```

    }
    reverse(res.begin(), res.end());
    return res;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 是树中节点个数
- 空间复杂度: $O(n)$, 其中 n 是树中节点个数, 空间复杂度取决于队列开销, 队列中的节点个数不会超过 $O(n)$

- 深搜

- 代码

```

class Solution {
public:
    // 深搜
    vector<vector<int>> levelOrderBottom(TreeNode* root) {
        vector<vector<int>> res;
        dfs(root, 0, res);
        return res;
    }
    void dfs(TreeNode *node, int depth, vector<vector<int>> &res) {
        if (node == nullptr) {
            return;
        }

        // 没出现过, 则前插一个空数组
        if (depth == res.size()) {
            res.insert(res.begin(), vector<int>());
        }
        res[res.size() - depth - 1].push_back(node->val);
        if (node->left) {
            dfs(node->left, depth + 1, res);
        }
        if (node->right) {
            dfs(node->right, depth + 1, res);
        }
    }
};

```

116. 填充每个节点的下一个右侧节点指针

给定一个完美二叉树, 其所有叶子节点都在同一层, 每个父节点都有两个子节点。二叉树定义如下:

```

struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}

```

填充它的每个 next 指针, 让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点, 则将 next 指针设置为 NULL。

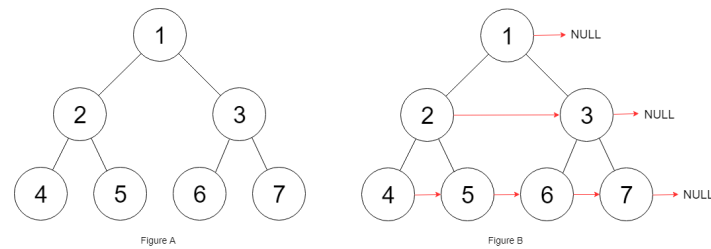
初始状态下，所有 next 指针都被设置为 NULL。

进阶：

你只能使用常量级额外空间。

使用递归解题也符合要求，本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例：



输入：root = [1,2,3,4,5,6,7]

输出：[1,#,2,3,#,4,5,6,7,#]

解释：给定二叉树如图 A 所示，你的函数应该填充它的每个 next 指针，以指向其下一个右侧节点，如图 B 所示。序列化的输出按层序遍历排列，同一层节点由 next 指针连接，'#' 标志着每一层的结束。

- 层次遍历

- 思路

- 题目本身希望我们将二叉树的每一层节点都连接起来形成一个链表，因此直观做法是对二叉树进行层次遍历，在层次遍历的过程中将二叉树每一层的节点拿出来遍历并连接
 - 层次遍历基于广搜，广搜每次只会取出一个节点来拓展，而层次遍历是取出一层所有元素来拓展，这样能保证每次从队列中拿出来遍历的元素都是属于同一层的，因此可以再遍历的过程中修改每个节点的next指针，同时拓展下一层的新队列

- 代码

```
// https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node/solution/tian-chong-mei-ge-jie-dian-de-xia-yi-ge-you-ce-2-4/
// 层次遍历
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) {
            return root;
        }

        queue<Node*> q;
        q.push(root);

        while (!q.empty()) {
            // 记录当前队列大小，表示当前层节点数目
            int n = q.size();
            for (int i = 0; i < n; ++i) {
                Node *node = q.front();
                q.pop();
                // 链接
                if (i < n - 1) {
                    node->next = q.front();
                }
                if (node->left != nullptr) {
                    q.push(node->left);
                }
            }
        }
    }
};
```

```

    }
    if (node->right != nullptr) {
        q.push(node->right);
    }
}
}
return root;
}
};

```

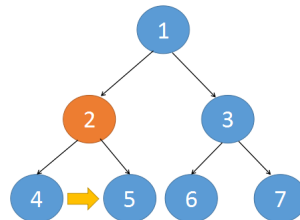
复杂度

- 时间复杂度: $O(n)$, 每个节点都会被访问一次且只有一次, 即从队列弹出, 建立next指针
- 空间复杂度: $O(n)$, 这是完美二叉树, 最后一层包含 $n / 2$ 个节点, 空间复杂度取决于一层上的最大元素数目

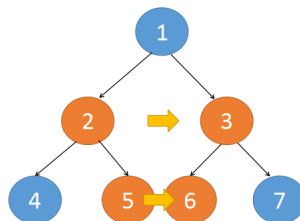
使用已建立的next指针

思路: 一棵树中, 存在两种类型的next指针

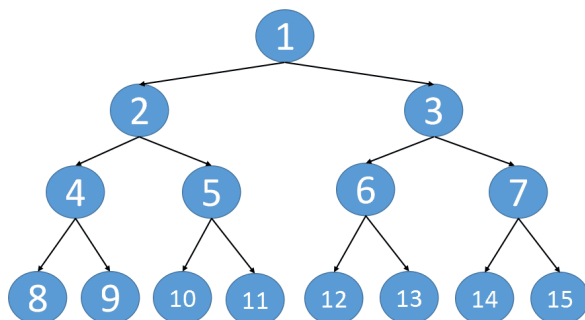
- 第一种情况是连接同一个父节点的两个子节点。它们可以通过同一个节点直接访问到, 因此执行 `node.left.next = node.right` 即可完成连接



- 第二种情况在不同父亲的子节点之间建立连接, 这种情况不能直接连接。需要借助上一层已连接好的节点, 通过父节点的 next 找到邻居 `node.right.next = node.next.left`



- 就是说, 在串联第 i 层节点时, 需要先完成第 $i - 1$ 层的节点串联。
 - 第一层最多只有一个节点, 不需要串联
 - 第二层最多只有两个节点, 借助根节点可完成串联
 - 第三层串联时, 第二层已串联完, 可借助第二层节点完成第三层串联, 后续同理



代码

```
// https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node/solution/tian-chong-mei-ge-jie-dian-de-xia-yi-ge-you-ce-2-4/
// https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node/solution/dong-hua-yan-shi-san-chong-shi-xian-116-tian-chong/
// 利用已建立好的next指针
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) { return root; }
        Node *cur = root;
        // 第一层while遍历层
        while (cur != nullptr) {
            Node dummy;
            Node *tail = &dummy;
            // 第二层while遍历层内节点
            while (cur != nullptr) {
                if (cur->left != nullptr) {
                    tail->next = cur->left;
                    tail = tail->next;
                }
                if (cur->right != nullptr) {
                    tail->next = cur->right;
                    tail = tail->next;
                }
                cur = cur->next;
            }
            cur = dummy.next;
        }
        return root;
    }
};
```

- 复杂度
 - 时间复杂度: $O(n)$, 每个节点都会被访问一次
- 空间复杂度: $O(1)$

117. 填充每个节点的下一个右侧节点指针 II

给定一个二叉树

```
struct Node {
    int val;
    Node *left;
    Node *right;
    Node *next;
}
```

填充它的每个 next 指针, 让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点, 则将 next 指针设置为 NULL。

初始状态下, 所有 next 指针都被设置为 NULL。

进阶:

你只能使用常量级额外空间。

使用递归解题也符合要求, 本题中递归程序占用的栈空间不算做额外的空间复杂度。

示例:

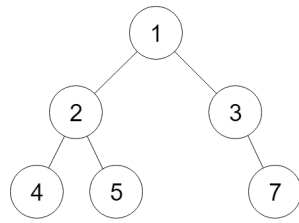


Figure A

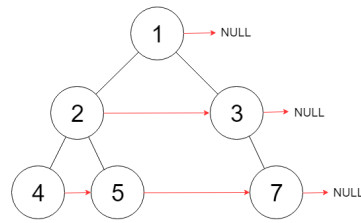


Figure B

输入: root = [1,2,3,4,5,null,7]

输出: [1,#,2,3,#,4,5,7,#]

解释: 给定二叉树如图 A 所示, 你的函数应该填充它的每个 next 指针, 以指向其下一个右侧节点, 如图 B 所示。序列化输出按层序遍历顺序 (由 next 指针连接), '#' 表示每层的末尾。

- 层次遍历
 - 同题116, 层次遍历解法同样适合于本题
- 使用已建立的next指针
 - 思路
 - 同题116, 可借助上一层已连接的next指针, 完成本层next指针的串联。
 - 从左向右看, 树的每层都是一个单链表, 通过外循环的cur跳进下一层, 内循环的cur遍历当前层节点
 - 在每层开始时, 创建一个虚拟头结点dummy, 用来保存下一层的起始节点。并使用tail指向当前要修改next指针的节点, 即实际操作next的指针
 - 第一层的根节点不需要任何修改, 直接操作第二层节点
 - 代码

```

// https://leetcode-cn.com/problems/populating-next-right-pointers-in-each-node-ii/solution/117tian-chong-jie-dian-de-xia-yi-ge-jie-dian-you-c/
// 利用已建立好的next指针
class Solution {
public:
    Node* connect(Node* root) {
        if (root == nullptr) { return root; }
        Node *cur = root;
        // 第一层while遍历层
        while (cur != nullptr) {
            Node dummy;
            Node *tail = &dummy;
            // 第二层while遍历层内节点
            while (cur != nullptr) {
                if (cur->left != nullptr) {
                    tail->next = cur->left;
                    tail = tail->next;
                }
                if (cur->right != nullptr) {
                    tail->next = cur->right;
                    tail = tail->next;
                }
                cur = cur->next;
            }
            cur = dummy.next;
        }
        return root;
    }
};

```

- 复杂度

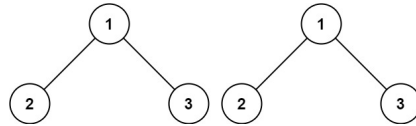
- 时间复杂度: $O(n)$, 每个节点都会被访问一次
- 空间复杂度: $O(1)$

100. 相同的树

给你两棵二叉树的根节点 p 和 q , 编写一个函数来检验这两棵树是否相同。

如果两个树在结构上相同, 并且节点具有相同的值, 则认为它们是相同的。

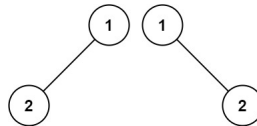
示例 1:



输入: $p = [1,2,3]$, $q = [1,2,3]$

输出: true

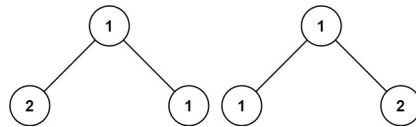
示例 2:



输入: $p = [1,2]$, $q = [1,null,2]$

输出: false

示例 3:



输入: $p = [1,2,1]$, $q = [1,1,2]$

输出: false

- 深搜

- 思路

- 如果两个二叉树都为空, 则相同
- 如果有且只有一个为空, 则不同
- 如果都不为空, 则首先判断根节点值是否相同, 若不相同则不同; 若相同, 则递归判断左右子树

- 代码

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (p == nullptr && q == nullptr) { return true; }
        if (p == nullptr || q == nullptr) { return false; }
        if (p->val != q->val) { return false; }
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

- 复杂度

- 时间复杂度： $O(\min(m, n))$ ，其中 m 和 n 分别是两个二叉树的节点数。对两个二叉树同时进行深度优先搜索，只有当两个二叉树中的对应节点都不为空时才会访问该节点，故访问节点数目不会超过较小的二叉树节点数
- 空间复杂度： $O(\min(m, n))$ ，其中 m 和 n 分别是两个二叉树的节点数。空间复杂度取决于递归调用的层数，递归调用的层数不会超过较小的二叉树的最大深度，最坏情况下，二叉树的高度等于节点数

- 广搜

- 思路

可使用广搜来判断，使用队列存储节点，将要比较的两个节点同时入队、出队，进行比较

- 代码

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        queue<std::pair<TreeNode*, TreeNode*>> q;
        q.push(std::make_pair(p, q));
        while (!q.empty()) {
            std::pair<TreeNode*, TreeNode*> cur = q.front();
            q.pop();
            TreeNode *tmp_p = cur.first, *tmp_q = cur.second;
            if (tmp_p == nullptr && tmp_q == nullptr) { continue; }
            if (tmp_p == nullptr || tmp_q == nullptr || tmp_p->val !=
tmp_q->val) {
                return false;
            }
            q.push(std::make_pair(tmp_p->left, tmp_q->left));
            q.push(std::make_pair(tmp_p->right, tmp_q->right));
        }
        return true;
    }
};
```

- 复杂度

- 时间复杂度： $O(\min(m, n))$ ，其中 m 和 n 分别是两个二叉树的节点数。对两个二叉树同时进行广度优先搜索，只有当两个二叉树中的对应节点都不为空时才会访问该节点，故访问节点数目不会超过较小的二叉树节点数
- 空间复杂度： $O(\min(m, n))$ ，其中 m 和 n 分别是两个二叉树的节点数。空间复杂度取决于队列中的元素个数，不会超过较小的二叉树的最大深度，最坏情况下，二叉树的高度等于节点数

- 参考

- <https://leetcode-cn.com/problems/same-tree/solution/xiang-tong-de-shu-by-leetcode-solution/>
- <https://leetcode-cn.com/problems/same-tree/solution/dai-ma-sui-xiang-lu-100-xiang-to-ng-de-sh-2k5k/>

101. 对称二叉树

给定一个二叉树，检查它是否是镜像对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```

    1
   / \
  2   2
 / \ / \
3  4 4  3
```

```
/ \ / \
3 4 4 3
```

但是下面这个 [1,2,2,null,3,null,3] 则不是镜像对称的:

```
1
/\
2 2
 \ \
 3 3
```

- 递归
 - 思路
 - 镜像对称的条件
 - 两个根节点具有相同值
 - 每个树的右子树都与另一个树的左子树镜像对称
 - 可实现这样递归函数，通过 **同步移动**两个指针的方法来遍历这棵树，p指针和q指针一开始指向根节点的左子树和右子树，随后同步移动，p指针指向左子树时，q指针指向右子树；反之亦然，每次都判断p节点和q节点的值是否相同，如果相同再递归判断p、q的左右子树

- 代码

```
class Solution {
public:
    bool helper(TreeNode *p, TreeNode *q) {
        if (p == nullptr && q == nullptr) { return true; }
        if (p == nullptr || q == nullptr) { return false; }
        if (p->val != q->val) { return false; }
        return helper(p->left, q->right) && helper(p->right, q->left);
    }
    bool isSymmetric(TreeNode* root) {
        return helper(root->left, root->right);
    }
};
```

- 复杂度
 - 时间复杂度： $O(n)$ ，遍历整棵树，渐进时间复杂度为 $O(n)$
 - 空间复杂度： $O(n)$ ，递归调用栈空间，调用层数不超过 n ，故渐进空间复杂度为 $O(n)$

- 迭代

- 思路

迭代方法使用队列判断镜像对称；初始时根节点左右子节点入队，每次提取两个节点比较，然后将两个节点的左右子节点按相反顺序入队。当队列为空时，或检测到树不对称时，结束

- 代码

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (root == nullptr) { return true; }
        queue<std::pair<TreeNode*, TreeNode*>> qp;
        qp.push(std::make_pair(root->left, root->right));
        while (!qp.empty()) {
```

```

        auto item = qp.front();
        qp.pop();
        TreeNode *cp = item.first, *cq = item.second;
        if (cp == nullptr && cq == nullptr) { continue; }
        if (cp == nullptr || cq == nullptr || cp->val != cq->val) {
            return false; }
        qp.push(std::make_pair(cp->left, cq->right));
        qp.push(std::make_pair(cp->right, cq->left));
    }
    return true;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$, 同「方法一」。
- 空间复杂度: 这里需要用一个队列来维护节点, 每个节点最多进队一次, 出队一次, 队列中最多不会超过 n 个点, 故渐进空间复杂度为 $O(n)$

- 参考

- <https://leetcode-cn.com/problems/symmetric-tree/solution/dui-cheng-er-cha-shu-by-lee-tcode-solution/>

226. 翻转二叉树

翻转一棵二叉树。

示例:

输入:

```

4
 / \
2   7
 / \ / \
1  3 6  9

```

输出:

```

4
 / \
7   2
 / \ / \
9  6 3  1

```

-

```

class Solution {
public:
    // 深搜
    // TreeNode* invertTree(TreeNode* root) {
    //     if (root == nullptr) { return root; }
    //     swap(root->left, root->right);
    //     invertTree(root->left);
    //     invertTree(root->right);
    //     return root;
    // }

    // 广搜
    TreeNode* invertTree(TreeNode* root) {
        if (root == nullptr) { return root; }
    }
};

```



```

std::queue<TreeNode*> q;
q.push(root);
while (!q.empty()) {
    TreeNode *node = q.front();
    q.pop();
    swap(node->left, node->right);
    if (node->left != nullptr) {
        q.push(node->left);
    }
    if (node->right != nullptr) {
        q.push(node->right);
    }
}
return root;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

104. 二叉树的最大深度

给定一个二叉树，找出其最大深度。

二叉树的深度为根节点到最远叶子节点的最长路径上的节点数。

说明: 叶子节点是指没有子节点的节点。

示例:

给定二叉树 [3,9,20,null,null,15,7],

```

3
 / \
9  20
 /  \
15  7

```

返回它的最大深度 3。

- 深搜

- 思路

若知道了左子树和右子树的最大深度 l 和 r ，那么该二叉树的最大深度即为 $\max(l, r) + 1$ 。而左右子树的最大深度又可以递归计算，故可使用深搜方法。即计算当前二叉树的最大深度时，先递归计算出其左右子树的最大深度，然后+1得到当前二叉树的最大深度，递归在访问空节点时推出

- 代码

```

class Solution {
public:
    // 深搜
    int maxDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        return 1 + max(maxDepth(root->left), maxDepth(root->right));
    }
};

```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 为二叉树节点的个数。每个节点在递归中只被遍历一次。
- 空间复杂度: $O(\text{height})$, 其中 height 表示二叉树的高度。递归函数需要栈空间, 而栈空间取决于递归的深度, 因此空间复杂度等价于二叉树的高度。

- 广搜

- 思路

同样可使用广搜解决, 此时广搜队列中存放的是 **当前层的所有节点**, 每次拓展下一层时, 需要将队列中的所有节点都拿出来拓展, 这样能保证每次拓展完的时候队列里存放的都是下一层的所有节点, 即一层层拓展, 最后使用一个变量 res 来维护拓展次数, 即为最大深度

- 代码

```

class Solution {
public:
    // 广搜
    int maxDepth(TreeNode* root) {
        if (root == nullptr) { return 0; }
        queue<TreeNode*> q;
        q.push(root);
        int depth = 0;
        while (!q.empty()) {
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                TreeNode *cur = q.front();
                q.pop();
                if (cur->left) { q.push(cur->left); }
                if (cur->right) { q.push(cur->right); }
            }
            ++depth;
        }
        return depth;
    }
};

```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 为二叉树的节点个数。与方法一同样的分析, 每个节点只会被访问一次。
- 空间复杂度: 此方法空间的消耗取决于队列存储的元素数量, 其在最坏情况下会达到 $O(n)$ 。

- 参考

- <https://leetcode-cn.com/problems/maximum-depth-of-binary-tree/solution/er-cha-shu-de-zui-da-shen-du-by-leetcode-solution/>

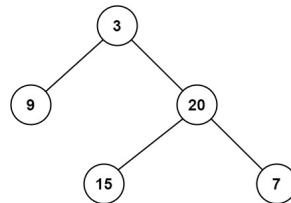
111. 二叉树的最小深度

给定一个二叉树，找出其最小深度。

最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

说明：叶子节点是指没有子节点的节点。

示例 1:



输入: root = [3,9,20,null,null,15,7]

输出: 2

示例 2:

输入: root = [2,null,3,null,4,null,5,null,6]

输出: 5

- 深搜

- 思路

遍历整棵树，记录最小深度。对于每一个非叶子节点，只需要分别计算其左右子树的最小叶子节点深度，这样就将一个大问题转化为了小问题，可以递归的解决该问题

- 代码

```
// https://leetcode-cn.com/problems/minimum-depth-of-binary-tree/solution/er-cha-shu-de-zui-xiao-shen-du-by-leetcode-solutio/
class Solution {
public:
    int minDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        if (root->left == nullptr && root->right == nullptr) {
            return 1;
        }

        int min_depth = INT_MAX;
        if (root->left != nullptr) {
            min_depth = min(minDepth(root->left), min_depth);
        }
        if (root->right != nullptr) {
            min_depth = min(minDepth(root->right), min_depth);
        }
        return min_depth + 1;
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。每个节点访问一次

- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于递归调用的层数, 层数不会超过 n

- 广搜

- 思路

同样可是使用广搜方法, 遍历整棵树。当我们找到一个叶子节点时, 直接返回这个叶子节点的深度, 广搜的性质保证了**最先搜索到的叶子节点的函数深度一定最小**

- 代码

```
// https://leetcode-cn.com/problems/minimum-depth-of-binary-tree/solution/er-cha-shu-de-zui-xiao-shen-du-by-leetcode-solutio/
class Solution {
public:
    int minDepth(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        queue<pair<TreeNode*, int>> q;
        q.emplace(root, 1);
        while (!q.empty()) {
            TreeNode *node = q.front().first;
            int depth = q.front().second;
            q.pop();
            if (node->left == nullptr && node->right == nullptr) {
                return depth;
            }
            if (node->left != nullptr) {
                q.emplace(node->left, depth + 1);
            }
            if (node->right != nullptr) {
                q.emplace(node->right, depth + 1);
            }
        }
        return 0;
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。每个节点访问一次
- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于队列的开销, 队列的元素个数不会超过树的节点数

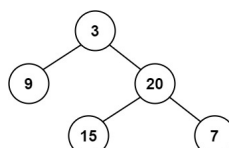
110. 平衡二叉树

给定一个二叉树, 判断它是否是高度平衡的二叉树。

本题中, 一棵高度平衡二叉树定义为:

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。

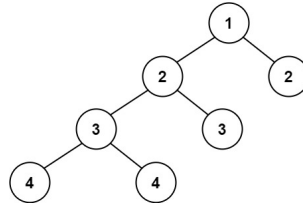
示例 1:



输入: root = [3,9,20,null,null,15,7]

输出: true

示例 2:



输入: root = [1,2,2,3,3,null,null,4,4]

输出: false

示例 3:

输入: root = []

输出: true

- 自顶向下的递归

- 思路

- 定义函数 `height`，用于计算二叉树中的任意一个节点 `p` 的高度

$$height(p) = \begin{cases} 0, & p \text{ 是空节点} \\ \max(height(p.left), height(p.right)) + 1, & p \text{ 是非空节点} \end{cases}$$

- 有了计算节点高度的函数，即可判断二叉树是否平衡。具体做法类似于二叉树的前序遍历，即对于当前遍历到的节点，首先计算左右子树的高度，如果左右子树的高度差不超过1，再分别递归地遍历左右子节点，并判断左子树和右子树是否平衡。这是自顶向下的递归过程

- 代码

```
// https://leetcode-cn.com/problems/balanced-binary-tree/solution/ping-heng-er-cha-shu-by-leetcode-solution/
class Solution {
public:
    int height(TreeNode *root) {
        if (root == nullptr) {
            return 0;
        }
        return max(height(root->left), height(root->right)) + 1;
    }
    bool isBalanced(TreeNode* root) {
        if (root == nullptr) { return true; }
        return abs(height(root->left) - height(root->right)) <= 1 &&
            isBalanced(root->left) && isBalanced(root->right);
    }
};
```

- 复杂度

- 时间复杂度: $O(n^2)$ ，其中 n 是二叉树中的节点个数。最坏情况下，二叉树是满二叉树，需要遍历二叉树中的所有节点，时间复杂度是 $O(n)$ 。对于节点 `p`，如果它的高度是 `d`，则 `height(p)` 最多会被调用 `d` 次（即遍历到它的每一个祖先节点时）。
 - 对于平均情况，一棵树的高度满足 $O(h) = O(\log n)$ ，因为 $d \leq h$ ，故总时间复杂度为 $O(n \log n)$ 。
 - 对于最坏情况，二叉树形成链式结构，高度为 $O(n)$ ，总时间复杂度为 $O(n^2)$

- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于递归调用的层数, 层数不会超过 n
- 自底向上的递归
 - 思路
 - 方法一由于是自顶向下递归, 故对同一节点, 函数 `height` 会被重复调用, 导致时间复杂度过高。如果使用自底向上的做法, 则对于每个节点, 函数 `height` 只会被调用一次
 - 自底向上递归类似于后序遍历, 对于当前遍历到的节点, 先递归的判断其左右子树是否平衡, 再判断以当前节点为根的子树是否平衡。如果一颗子树是平衡的, 则返回其高度 (高度一定是非负整数), 否则返回 -1。如果存在一颗子树不平衡, 则整个二叉树一定不平衡
 - 代码

```
// https://leetcode-cn.com/problems/balanced-binary-tree/solution/ping-heng-er-cha-shu-by-leetcode-solution/
class solution {
public:
    int height(TreeNode *root) {
        if (root == nullptr) {
            return 0;
        }
        int left_height = height(root->left);
        int right_height = height(root->right);
        if (left_height == -1 || right_height == -1 || abs(left_height - right_height) > 1) {
            return -1;
        }
        return max(left_height, right_height) + 1;
    }
    bool isBalanced(TreeNode* root) {
        return height(root) >= 0;
    }
};
```

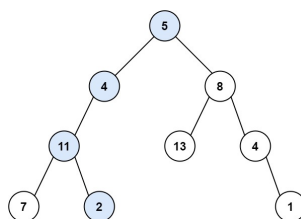
- 复杂度
 - 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。使用自底向上的递归, 每个节点的计算高度和判断是否平衡都只需要处理一次, 最坏情况下需要遍历二叉树中的所有节点, 因此时间复杂度是 $O(n)$
 - 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于递归调用的层数, 层数不会超过 n

112. 路径总和

给你二叉树的根节点 `root` 和一个表示目标和的整数 `targetSum`, 判断该树中是否存在 根节点到叶子节点 的路径, 这条路径上所有节点值相加等于目标和 `targetSum`。

叶子节点 是指没有子节点的节点。

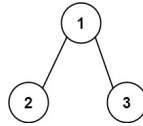
示例 1:



输入: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

输出: true

示例 2:



输入: root = [1,2,3], targetSum = 5

输出: false

示例 3:

输入: root = [1,2], targetSum = 0

输出: false

- 广搜
 - 思路
 - 核心思想是对树进行一次遍历，在遍历时记录从根节点到当前节点的路径和，以防重复计算
 - 首先可想到，利用广搜记录从根节点到当前节点的路径和，以防重复计算。可使用两个队列，分别存储要遍历的节点，以及根节点到这些节点的路径和即可
 - 代码

```
// https://leetcode-cn.com/problems/path-sum/solution/lu-jing-zong-he-by-leetcode-solution/
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (root == nullptr) {
            return false;
        }

        queue<TreeNode*> node_q;
        queue<int> val_q;
        node_q.push(root);
        val_q.push(root->val);

        while (!node_q.empty()) {
            TreeNode *cur_node = node_q.front();
            node_q.pop();
            int cur_val = val_q.front();
            val_q.pop();
            if (cur_node->left == nullptr && cur_node->right == nullptr)
            {
                if (cur_val == targetSum) {
                    return true;
                }
                continue;
            }
            if (cur_node->left != nullptr) {
                node_q.push(cur_node->left);
                val_q.push(cur_node->left->val + cur_val);
            }
            if (cur_node->right != nullptr) {
                node_q.push(cur_node->right);
                val_q.push(cur_node->right->val + cur_val);
            }
        }
    }
};
```

```

    }
    }
    return false;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。每个节点访问一次
- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于队列的开销, 队列的元素个数不会超过树的节点数

- 深搜

- 思路

- 假设从根节点到当前节点的值之和为 `val`, 可将该问题转化为小问题: 是否存在从当前节点的子节点到叶子的路径, 满足其路径和为 `sum - val`
- 不难发现, 可使用递归。若当前节点就是叶子节点, 则可直接判断 `sum == val?` (因为路径和已经确定, 就是当前节点的值, 只需要判断该路径和是否满足条件)。如当前节点不是叶子节点, 只需要递归询问它的子节点是否能满足条件即可

- 代码

```

// https://leetcode-cn.com/problems/path-sum/solution/lu-jing-zong-he-by-leetcode-solution/
class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        if (root == nullptr) {
            return false;
        }
        if (root->left == nullptr && root->right == nullptr) {
            return root->val == targetSum;
        }
        return hasPathSum(root->left, targetSum - root->val) ||
            hasPathSum(root->right, targetSum - root->val);
    }
};

```

- 复杂度

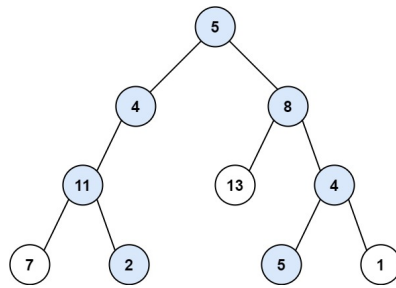
- 时间复杂度: $O(n)$, 其中 n 是二叉树中的节点个数。每个节点访问一次
- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于递归时栈空间的开销, 栈空间不会超过树的节点数

113. 路径总和 II

给你二叉树的根节点 `root` 和一个整数目标和 `targetSum`, 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。

叶子节点 是指没有子节点的节点。

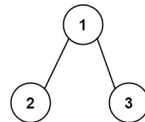
示例 1:



输入: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22

输出: [[5,4,11,2],[5,8,4,5]]

示例 2:



输入: root = [1,2,3], targetSum = 5

输出: []

示例 3:

输入: root = [1,2], targetSum = 0

输出: []

• 深搜+回溯

◦ 思路

- 注意本题要求是: 找到 **所有** 满足从 **根节点** 到某个 **叶子节点** 经过的路径上的节点之和等于目标和的路径。核心思想就是对树进行一次遍历, 在遍历时记录从根节点到当前节点的路径和, 以防止重复计算
- 可采用深搜方式, 枚举每一条从根节点到叶子节点的路径。当遍历到叶子节点, 且此时路径和恰为目标和时, 就找到了一条满足条件的路径

◦ 代码

```

// https://leetcode-cn.com/problems/path-sum-ii/solution/lu-jing-zong-he-ii-by-leetcode-solution/
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;

    void dfs(TreeNode *root, int targetSum) {
        if (root == nullptr) {
            return;
        }
        path.emplace_back(root->val);
        targetSum -= root->val;
        if (root->left == nullptr && root->right == nullptr && targetSum == 0) {
            res.emplace_back(path);
        }
        dfs(root->left, targetSum);
        dfs(root->right, targetSum);
        path.pop_back();
    }

    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        dfs(root, targetSum);
        return res;
    }
}
  
```

```
}  
};
```

- 复杂度

- 时间复杂度: $O(n^2)$, 其中 n 是二叉树中的节点个数。最坏情况下, 树的上半部分为链状, 下半部分为完全二叉树, 并且从根节点到每一个叶子节点的路径都符合题目要求, 此时路径的数目为 $O(n)$ 。并且每一条路径的节点个数也为 $O(n)$, 因此要将这些路径全部添加仅答案中, 时间复杂度为 $O(n^2)$, **耗时分析没看懂???**
- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于递归时栈空间的开销, 栈空间不会超过树的节点数

- 广搜

- 思路

- 也可采用广搜方式, 遍历这棵树, 当遍历到叶子节点, 且此时路径和恰为目标和时, 该路径满足条件
- 为了节省空间, 可使用哈希表记录树中的每一个节点的父节点, 每次找到一个满足条件的节点, 就从该节点出发不断向父节点迭代, 即可还原出从根节点到当前节点的路径

- 代码

```
// https://leetcode-cn.com/problems/path-sum-ii/solution/lu-jing-zong-he-ii-by-leetcode-solution/  
class Solution {  
public:  
    vector<vector<int>> res;  
    unordered_map<TreeNode*, TreeNode*> parents;  
  
    void getPath(TreeNode *node) {  
        vector<int> path;  
        while (node != nullptr) {  
            path.emplace_back(node->val);  
            node = parents[node];  
        }  
        reverse(path.begin(), path.end());  
        res.emplace_back(path);  
    }  
  
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {  
        if (root == nullptr) {  
            return res;  
        }  
        queue<TreeNode*> node_q;  
        queue<int> val_q;  
        node_q.emplace(root);  
        val_q.emplace(root->val);  
  
        while (!node_q.empty()) {  
            auto cur_node = node_q.front();  
            node_q.pop();  
            int cur_val = val_q.front();  
            val_q.pop();  
  
            if (cur_node->left == nullptr && cur_node->right == nullptr)  
{  
                if (cur_val == targetSum) {  
                    getPath(cur_node);  
                }  
            }  
        }  
    }  
};
```

```

    } else {
        if (cur_node->left != nullptr) {
            parents[cur_node->left] = cur_node;
            node_q.emplace(cur_node->left);
            val_q.emplace(cur_node->left->val + cur_val);
        }
        if (cur_node->right != nullptr) {
            parents[cur_node->right] = cur_node;
            node_q.emplace(cur_node->right);
            val_q.emplace(cur_node->right->val + cur_val);
        }
    }
}
return res;
}
};

```

复杂度

- 时间复杂度: $O(n^2)$, 其中 n 是二叉树中的节点个数。最坏情况下, 树的上半部分为链状, 下半部分为完全二叉树, 并且从根节点到每一个叶子节点的路径都符合题目要求, 此时路径的数目为 $O(n)$ 。并且每一条路径的节点个数也为 $O(n)$, 因此要将这些路径全部添加进答案中, 时间复杂度为 $O(n^2)$, **耗时分析没看懂???**
- 空间复杂度: $O(n)$, 其中 n 是二叉树中节点个数。空间复杂度主要取决于哈希表和队列的开销, 哈希表需要存储根节点外的每个节点的父节点, 队列中的元素个数不会超过树的节点数

257. 二叉树的所有路径

给定一个二叉树, 返回所有从根节点到叶子节点的路径。

说明: 叶子节点是指没有子节点的节点。

示例:

输入:

```

1
 / \
2   3
 \
  5

```

输出: ["1->2->5", "1->3"]

解释: 所有根节点到叶子节点的路径为: 1->2->5, 1->3

深搜

思路

深搜遍历二叉树, 需要考虑当前的节点以及其孩子节点

- 如果当前节点不是叶子节点, 则在当前的路径末位添加该节点, 并继续递归遍历节点的每个孩子节点
- 如果当前节点是叶子节点, 则在当前路径末位添加该节点后, 就得到一条从根节点到叶子节点的路径, 将其加入到结果集中

如此, 当遍历完整颗二叉树, 就得到结果了, 当前也可使用非递归完成

代码

```
// https://leetcode-cn.com/problems/binary-tree-paths/solution/er-shu-de-suo-you-lu-jing-by-leetcode-solution/
// 深搜
class Solution {
public:
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        dfs(root, "", res);
        return res;
    }

    void dfs(TreeNode* root, string path, vector<string> &res) {
        if (root == nullptr) { return; }
        path += std::to_string(root->val);
        if (root->left == nullptr && root->right == nullptr) {
            res.emplace_back(path);
            return;
        }
        path += "->";
        if (root->left != nullptr) {
            dfs(root->left, path, res);
        }
        if (root->right != nullptr) {
            dfs(root->right, path, res);
        }
    }
};
```

- 复杂度

- 时间复杂度: $O(n^2)$, 其中 n 表示节点数目。深搜中, 每个节点都会被访问一次, 并且每次都会对 $path$ 进行拷贝构造, 时间代价为 $O(n)$, 故总时间复杂度为 $O(n^2)$
- 空间复杂度: $O(n^2)$, 其中 n 表示节点数目。除答案数组外我们需要考虑递归调用的栈空间。在最坏情况下, 当二叉树中每个节点只有一个孩子节点时, 即整棵二叉树呈一个链状, 此时递归的层数为 NN , 此时每一层的 $path$ 变量的空间代价的总和为 $O(\sum_{i=1}^N i) = O(N^2)$ 空间复杂度为 $O(N^2)$ 。最好情况下, 当二叉树为平衡二叉树时, 它的高度为 $\log N$, 此时空间复杂度为 $O((\log N)^2)$

- 广搜

- 思路

维护两个队列, 分别存储节点以及跟节点到该节点的路径。一开始只有根节点, 迭代过程中, 取出队列的首节点。判断, 若是叶子节点, 则将对路径存到结果集中。若不是叶子节点, 则将其左右孩子加入到队列中

- 代码

```
// https://leetcode-cn.com/problems/binary-tree-paths/solution/er-shu-de-suo-you-lu-jing-by-leetcode-solution/
class Solution {
public:
    // 广搜
    vector<string> binaryTreePaths(TreeNode* root) {
        vector<string> res;
        if (root == nullptr) { return res; }
        std::queue<std::pair<TreeNode*, string>> qs;
        qs.emplace(root, "");
        while (!qs.empty()) {
```

```

        auto item = qs.front();
        qs.pop();
        TreeNode *node = item.first;
        string path = item.second;
        path += std::to_string(node->val);
        if (node->left == nullptr && node->right == nullptr) {
            res.emplace_back(path);
        }
        path += "->";
        if (node->left != nullptr) {
            qs.emplace(node->left, path);
        }
        if (node->right != nullptr) {
            qs.emplace(node->right, path);
        }
    }
    return res;
}

```

复杂度

- 时间复杂度： $O(n^2)$ ，其中 n 表示节点数目。深搜中，每个节点都会被访问一次，并且每次都会对`path`进行拷贝构造，时间代价为 $O(n)$ ，故总时间复杂度为 $O(n^2)$
- 空间复杂度： $O(n^2)$ ，其中 n 表示节点数目。最坏情况下，队列中存在 n 个节点。保存字符串的队列中每个节点的最大长度为 n ，故空间复杂度为 $O(n^2)$

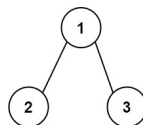
129. 求根节点到叶节点数字之和

给你一个二叉树的根节点 `root`，树中每个节点都存放有一个 0 到 9 之间的数字。
每条从根节点到叶节点的路径都代表一个数字：

例如，从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123。
计算从根节点到叶节点生成的 所有数字之和。

叶节点 是指没有子节点的节点

示例 1：



输入：root = [1,2,3]

输出：25

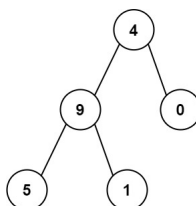
解释：

从根到叶子节点路径 1->2 代表数字 12

从根到叶子节点路径 1->3 代表数字 13

因此，数字总和 = 12 + 13 = 25

示例 2：



输入: root = [4,9,0,5,1]

输出: 1026

解释:

从根到叶子节点路径 4->9->5 代表数字 495

从根到叶子节点路径 4->9->1 代表数字 491

从根到叶子节点路径 4->0 代表数字 40

因此, 数字总和 = 495 + 491 + 40 = 1026

- 深搜

- 思路

从根节点开始, 遍历每个节点, 如果遇到叶子节点, 则将叶子节点对应的数字加到数字之和。
如果当前节点不是叶子节点, 则计算其子节点对应的数字, 然后对子节点递归遍历

- 代码

```
// https://leetcode-cn.com/problems/sum-root-to-leaf-  
numbers/solution/qiu-gen-dao-xie-zi-jie-dian-shu-zi-zhi-he-by-leetc/  
// 深搜  
class Solution {  
public:  
    int dfs(TreeNode *root, int pre_sum) {  
        if (root == nullptr) {  
            return 0;  
        }  
        int cur_sum = pre_sum * 10 + root->val;  
        if (root->left == nullptr && root->right == nullptr) {  
            return cur_sum;  
        } else {  
            return dfs(root->left, cur_sum) + dfs(root->right, cur_sum);  
        }  
    }  
    int sumNumbers(TreeNode* root) {  
        return dfs(root, 0);  
    }  
};
```

- 复杂度

- 时间复杂度: $O(n)$, 每个节点都会被访问一次
 - 空间复杂度: $O(n)$, 其中 n 是二叉树的节点个数, 空间复杂度主要取决于递归调用的栈空间, 递归栈的深度等于二叉树的高度, 最坏情况下, 二叉树的高度等于节点个数

- 广搜

- 思路

- 使用广搜, 需要维护两个队列, 分别存储节点和节点对应的数字
 - 初始时, 将根节点和根节点的值分别加入到两个队列。每次从这两个队列取出一个节点和一个值, 进行如下操作
 - 如果当前节点是叶子节点, 则将该节点对应的数字加到数字之和
 - 如果当前节点不是叶子节点, 则获得当前节点的非空子节点, 并根据当前节点对应的数字和子节点的值, 计算子节点对应的数字, 然后将子节点和子节点对应的数字分别加入到两个队列
 - 搜索结束后, 可得到所有叶子节点对应的数字之和

- 代码

```
// https://leetcode-cn.com/problems/sum-root-to-leaf-
// numbers/solution/qiu-gen-dao-xie-zi-jie-dian-shu-zi-zhi-he-by-leetc/
// 广搜
class solution {
public:
    int sumNumbers(TreeNode* root) {
        int res = 0;
        if (root == nullptr) {
            return res;
        }
        queue<std::pair<TreeNode*, int>> qp;
        qp.push(std::make_pair(root, 0));
        while (!qp.empty()) {
            auto item = qp.front();
            qp.pop();
            TreeNode *cur = item.first;
            if (cur == nullptr) { continue; }
            int cur_sum = item.second * 10 + cur->val;
            if (cur->left == nullptr && cur->right == nullptr) {
                res += cur_sum;
            }
            if (cur->left != nullptr) {
                qp.push(std::make_pair(cur->left, cur_sum));
            }
            if (cur->right != nullptr) {
                qp.push(std::make_pair(cur->right, cur_sum));
            }
        }
        return res;
    }
};
```

复杂度

- 时间复杂度: $O(n)$, 每个节点都会被访问一次
- 空间复杂度: $O(n)$, 其中 n 是二叉树的节点个数, 空间复杂度主要取决于队列, 队列中的元素不会超过 n

199. 二叉树的右视图

给定一棵二叉树, 想象自己站在它的右侧, 按照从顶部到底部的顺序, 返回从右侧所能看到的节点值。

示例:

输入: [1,2,3,null,5,null,4]

输出: [1, 3, 4]

解释:

```

1      <---
/  \
2    3    <---
\  \
5    4    <---
```

广搜

思路

- 层次遍历, 对于每层来说, 最右边的节点一定是在最后遍历到的。

- 代码

```
// https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-by-leetcode-solution/361141
// 广搜
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        if (root == nullptr) {
            return res;
        }
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int n = q.size();
            for (int i = 0; i < n; ++i) {
                auto cur = q.front();
                q.pop();
                if (cur->left) {
                    q.push(cur->left);
                }
                if (cur->right) {
                    q.push(cur->right);
                }
                if (i == n - 1) {
                    res.push_back(cur->val);
                }
            }
        }
        return res;
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

- 深搜

- 思路

按照 中、右、左 顺序访问，可保证每层都是最先访问最右边的节点；并且对每层来说，只有当

- 代码

```
// https://leetcode-cn.com/problems/binary-tree-right-side-view/solution/er-cha-shu-de-you-shi-tu-by-leetcode-solution/361141
// 深搜
class Solution {
public:
    void dfs(TreeNode *node, int depth, vector<int> &res) {
        if (node == nullptr) {
            return;
        }
        // 先访问 当前节点，再递归地访问 右子树 和 左子树。
    }
};
```



```

        // 如果当前节点所在深度还没有出现在res里，说明在该深度下当前节点是第一个被访问的节点，因此将当前节点加入res中。
        if (depth == res.size()) {
            res.push_back(node->val);
        }
        ++depth;
        dfs(node->right, depth, res);
        dfs(node->left, depth, res);
    }
    vector<int> rightSideView(TreeNode* root) {
        vector<int> res;
        // 从根节点开始访问，根节点深度是0
        dfs(root, 0, res);
        return res;
    }
};

```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

513. 找树左下角的值

给定一个二叉树，在树的最后一行找到最左边的值。

示例 1:

输入:

```

2
/\
1 3

```

输出:

```
1
```

示例 2:

输入:

```

1
/\
2 3
/ /\
4 5 6
/
7

```

输出:

```
7
```

- 前序遍历

- 思路

- 定义变量, `max_level` 表示已经遍历到的最大层
- 递归查询, 判断当前层是否初次到达, 初次则更新结果值和 `max_level`

- 代码

```

// 前序遍历
// 思路
// 定义变量, max_level 表示已经遍历到的最大层
// 递归查询, 判断当前层是否初次到达, 初次则更新结果值和 max_level
// 代码

```

```
// https://leetcode-cn.com/problems/find-bottom-left-tree-
// value/solution/javati-jie-olkong-jian-de-qian-xu-bian-l-t8dj/
// 前序遍历
class solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        int max_level = 0, res;
        dfs(root, 1, max_level, res);
        return res;
    }

    void dfs(TreeNode *node, int level, int &max_level, int &res) {
        if (node == nullptr) {
            return;
        }
        if (level > max_level) {
            max_level = level;
            res = node->val;
        }

        dfs(node->left, level + 1, max_level, res);
        dfs(node->right, level + 1, max_level, res);
    }
};
```

- 复杂度
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(n)$, 系统栈空间
- 层次遍历
 - 思路: 层次遍历, 将每层节点倒序放到队列中, 最后一个节点就是结果
 - 代码

```
// https://leetcode-cn.com/problems/find-bottom-left-tree-
// value/solution/zi-jie-ti-ku-513-zhong-deng-zhao-shu-zuo-t56y/
// 层次遍历
class solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        TreeNode *res = root;
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            res = q.front();
            q.pop();
            if (res->right != nullptr) {
                q.push(res->right);
            }
            if (res->left != nullptr) {
                q.push(res->left);
            }
        }
        return res->val;
    }
};
```

```
}  
};
```

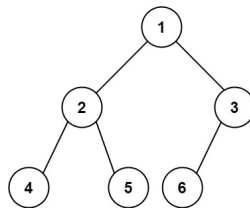
- 复杂度
 - 时间复杂度: $O(n)$
 - 空间复杂度: $O(n)$, 队列所占空间

222. 完全二叉树的节点个数

给你一棵 完全二叉树 的根节点 `root`，求出该树的节点个数。

完全二叉树的定义如下：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 $1 \sim 2^h$ 个节点。

示例 1：



输入: `root = [1,2,3,4,5,6]`

输出: 6

示例 2：

输入: `root = []`

输出: 0

示例 3：

输入: `root = [1]`

输出: 1

提示：

树中节点的数目范围是 $[0, 5 * 10^4]$

$0 \leq \text{Node.val} \leq 5 * 10^4$

题目数据保证输入的树是 完全二叉树

进阶：遍历树来统计节点是一种时间复杂度为 $O(n)$ 的简单解决方案。你可以设计一个更快的算法吗？

- 递归（遍历所有节点）
 - 思路
递归遍历所有节点，不过没有使用到完全二叉树的性质
 - 代码

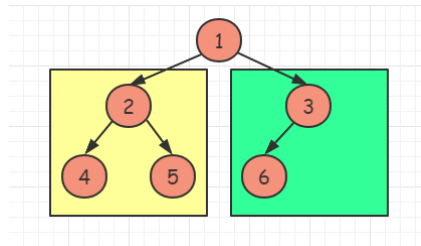
```
// https://leetcode-cn.com/problems/count-complete-tree-nodes/solution/c-san-chong-fang-fa-jie-jue-wan-quan-er-cha-shu-de/
// 递归
class Solution {
public:
    int countNodes(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }
        return countNodes(root->left) + countNodes(root->right) + 1;
    }
};
```

○ 复杂度


- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

● 利用完全二叉树性质

- 思路: 这是一颗完全二叉树, 除最后一层外, 其余层全部铺满; 且最后一层向左停靠
 - 如果根节点的左子树等于右子树深度, 则说明左子树为满二叉树



- 如果根节点的左子树深度大于右子树深度, 则说明右子树为满二叉树

 <https://pic.leetcode-cn.com/771abc84920a8ff8972a35cf690dae5f1a8b72ddfe513611ebb4ae07e6e7fa71> alt="left style="zoom:50%;>right" />

- 如果知道子树是满二叉树, 则可得到该子树的节点数目 $(1 \ll \text{depth}) - 1$, depth表示子树深度。使用移位操作符加快幂的运算速度
- 接着对另一棵子树递归即可

○ 代码

```
// https://leetcode.cn/problems/count-complete-tree-nodes/solutions/1812445/by-carlsun-2-bwlp/
// 利用完全二叉树性质
class Solution {
public:
    int countNodes(TreeNode* root) {
        if (root == nullptr) { return 0; }
        TreeNode *left = root, *right = root;
        int left_height = 0, right_height = 0;
        while (left != nullptr) {
            ++left_height;
            left = left->left;
        }

        while (right != nullptr) {
            ++right_height;
            right = right->right;
        }
    }
};
```

```

        if (left_height == right_height) {
            return (2 << left_height - 1) - 1; // 记住, 2 << 2 = 2 * 2 *
2 = 2^3次方, 所以应该是left_height - 1
        } else {
            return countNodes(root->left) + countNodes(root->right) + 1;
        }
    }
};

```

○ 复杂度

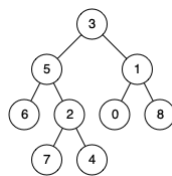
- 时间复杂度为 $O(\log^2 n)$,
- 空间复杂度为 $O(1)$ 【不考虑递归调用栈】
- 二分查找: 略, 有兴趣可阅读上述两种方法的参考链接和 [完全二叉树的节点个数](#)

236. 二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为: “对于有根树 T 的两个节点 p、q, 最近公共祖先表示为一个节点 x, 满足 x 是 p、q 的祖先且 x 的深度尽可能大 (一个节点也可以是它自己的祖先)。”

示例 1:

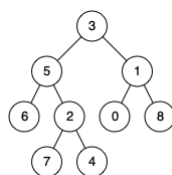


输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

输出: 3

解释: 节点 5 和节点 1 的最近公共祖先是节点 3。

示例 2:



输入: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

输出: 5

解释: 节点 5 和节点 4 的最近公共祖先是节点 5。因为根据定义最近公共祖先节点可以为节点本身。

示例 3:

输入: root = [1,2], p = 1, q = 2

输出: 1

- 自己ac

```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
    {

```

```

vector<TreeNode*> p_res, q_res, p_path, q_path;
dfs(root, p, p_path, p_res);
dfs(root, q, q_path, q_res);
TreeNode *res = nullptr;
int min_size = min(p_res.size(), q_res.size());
for (int i = 0; i < min_size; ++i) {
    if (p_res[i] == q_res[i]) {
        res = p_res[i];
    }
}
return res;
}

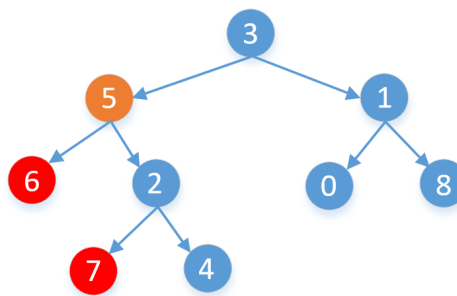
void dfs(TreeNode* root, TreeNode *node, vector<TreeNode*> &path,
vector<TreeNode*> &res) {
    if (root == nullptr || node == nullptr) { return; }
    path.push_back(root);
    if (root == node) {
        res = path;
        return;
    }
    dfs(root->left, node, path, res);
    dfs(root->right, node, path, res);
    path.pop_back();
}
};

```

- 迭代

- 思路

- 首先遍历树，找到节点p的祖先列表。然后遍历q，得到q的祖先列表。两个列表中第一个重合的祖先就是最近公共祖先



- 代码

```

// https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/javada-ma-di-gui-he-fei-di-gui-tu-wen-xiang-jie-b/
// 迭代
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
    TreeNode* q) {
        // 记录遍历到的每个节点的父节点
        unordered_map<TreeNode*, TreeNode*> parents;
        queue<TreeNode*> nq;
        // 根节点没有父节点
        parents[root] = nullptr;

```

```

nq.push(root);
// 知道两个节点都遍历完，将其父节点存放，就提前结束，不需要遍历完整棵树
while (parents.find(p) == parents.end() || parents.find(q) ==
parents.end()) {
    TreeNode *node = nq.front();
    nq.pop();
    // 处理左子树
    if (node->left != nullptr) {
        parents[node->left] = node;
        nq.push(node->left);
    }
    // 处理右子树
    if (node->right != nullptr) {
        parents[node->right] = node;
        nq.push(node->right);
    }
}
// 记录p和其祖先节点，从p到根节点的整个路径
unordered_set<TreeNode*> ancestors;
while (p != nullptr) {
    ancestors.insert(p);
    p = parents[p];
}
// 查看q和其祖先节点是否存在于p的祖先节点列表中
while (ancestors.find(q) == ancestors.end()) {
    q = parents[q];
}
return q;
}
};

```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

- 递归

- 思路

- 用递归解决是，需要明确，辅助函数的功能：在给定两个节点 `p` 和 `q` 时
 - 如果 `p` 和 `q` 都存在，则返回他们的公共祖先
 - 如果只存在一个，则返回存在的一个
 - 如果 `p` 和 `q` 都不存在，则返回 `nullptr`
- 具体思路
 - 如果当前节点 `root == nullptr`，则直接返回 `nullptr`
 - 如果 `root` 等于 `p` 或 `q`，则返回 `root`
 - 递归左右子树，因为是递归，使用函数后可认为左右子树已经得出结果，用 `left` 和 `right` 表示
 - 此时若 `left` 为空，则最终结果在 `right` 中。若 `right` 为空，则最终结果为 `left`
 - 如果 `left` 和 `right` 都非空，表示一边一个，`root` 就是最近公共祖先
 - 如果 `left` 和 `right` 都为空，则返回空

- 代码

```

// https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-tree/solution/c-jing-dian-di-gui-si-lu-fei-chang-hao-li-jie-shi/
// 递归

```

```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
    TreeNode* q) {
        if (root == nullptr) {
            return root;
        }
        if (root == p || root == q) {
            return root;
        }
        TreeNode *left = lowestCommonAncestor(root->left, p, q);
        TreeNode *right = lowestCommonAncestor(root->right, p, q);
        if (left == nullptr) {
            return right;
        }
        if (right == nullptr) {
            return left;
        }
        if (left != nullptr && right != nullptr) {
            return root;
        }
        return nullptr;
    }
};

```

○ 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$

437. 路径总和 III

给定一个二叉树，它的每个结点都存放着一个整数值。

找出路径和等于给定数值的路径总数。

路径不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

二叉树不超过1000个节点，且节点数值范围是 $[-1000000, 1000000]$ 的整数。

示例：

root = [10,5,-3,3,2,null,11,3,-2,null,1], sum = 8

```

    10
   / \
  5  -3
 / \  \
3  2  11
/\  \
3 -2 1

```

返回 3。和等于 8 的路径有：

1. 5 -> 3
2. 5 -> 2 -> 1
3. -3 -> 11

• 朴素dfs


```

class Solution {
public:
    // 朴素dfs
    int64_t pathSum(TreeNode* root, int targetSum) {
        if (root == nullptr) { return 0; }
        return dfs(root, targetSum) + pathSum(root->left, targetSum) +
            pathSum(root->right, targetSum);
    }
    int64_t dfs(TreeNode *root, int64_t targetSum) {
        if (root == nullptr) { return 0; }
        int64_t res = 0;
        targetSum -= root->val;
        if (targetSum == 0) {
            ++res;
        }
        return res + dfs(root->left, targetSum) + dfs(root->right,
            targetSum);
    }
};

```

- 前缀和解法

- 思路

- **前缀和定义**：一个节点的前缀和就是 **该节点到 根节点**之间的路径和

比如下图

节点 4 的前缀和为 $1 + 2 + 4 = 7$

节点 8 的前缀和为 $1 + 2 + 4 + 8 = 15$

节点 9 的前缀和为 $1 + 2 + 5 + 9 = 17$

- **前缀和对于本题的作用**：本题要求找出 **路径和等于 给定数值的 路径总数**，而 **两节点间的路径和=两节点间的前缀和之差**

比如下图，若题目给定数值为 5

节点 1 的前缀和为 1

节点 3 的前缀和为 $1 + 2 + 3 = 6$

$\text{prefix}(3) - \text{prefix}(1) = 5$ ，所以 节点 1 到节点 3 之间存在一条合格要求的路径
(节点2 --> 节点3)

理解上述计算之后，问题就得以简化：**只需要遍历整棵树一次，记录每个节点的前缀和，并查询该节点的祖先节点中符合条件的个数，将这个数目加到最终结果上**

- **HashMap存的是什么**：HashMap中的key为前缀和，value是该前缀和的节点数目，记录数目是因为有出现负数路径的可能

比如下图

前缀和为 1 的节点有两个：1, 0

所以路径和为 2 的路径数目有两条：0-->2, 2

- **恢复状态的意义**：由于题目要求 **路径方向必须是向下的（即只能从父节点到子节点）**。当讨论两个节点的前缀和差值时，有一个前提：**一个节点必须是另一个节点的祖先接线**，即当把某个节点的前缀和更新到 map 中，只能对该节点的子节点有效

比如下图，存在两个值为 2 的节点（A，B）。

当遍历到最右边的节点 6 时，对于它来说，此时的前缀和为 2 的节点只能有 B，因为从 A 节点无法达到节点 6（A 并不是节点 6 的祖先节点）。

若不做状态恢复，当遍历右子树时，左子树 A 的信息还保留在 map 中，那么此时节点 6 就会认为 A，B 节点都是可追溯的节点，从而产生错误

状态恢复代码的作用就是：在遍历完一个节点的所有子节点后，将其从 map 中删除，消除其影响

○ 代码

```
// 代码参考 https://leetcode-cn.com/problems/path-sum-iii/solution/qian-zhui-he-di-gui-hui-su-by-shi-huo-de-xia-tian/
// 代码参考 https://leetcode-cn.com/problems/path-sum-iii/solution/rang-ni-miao-dong-de-hui-su-qian-zhui-he-ou6t/
// 原理解析参考 https://leetcode-cn.com/problems/path-sum-iii/solution/dui-qian-zhui-he-jie-fa-de-yi-dian-jie-s-dey6/
class Solution {
public:
    int pathSum(TreeNode* root, int targetSum) {
        // 前缀和map, <前缀和, 出现次数>, 注意int会越界
        unordered_map<int64_t, int64_t> prefixSumCount;
        // 前缀和为0的路径只有一条, 即哪个节点都不选择
        prefixSumCount[0] = 1;
        // 前缀和递归回溯方法
        return dfs(root, prefixSumCount, targetSum, 0);
    }

    // 前缀和递归回溯方法
    int dfs(TreeNode *node, unordered_map<int64_t, int64_t>
    &prefixSumCount, int targetSum, int64_t currSum) {
        // 1. 递归终止条件
        if (node == nullptr) {
            return 0;
        }
        // 2. 本层要做的事情
        int res = 0;
        // 前缀和: 根节点到当前节点的和, 即路径和
        currSum += node->val;

        // 查询前缀和map, 确定路径中是否存在某个节点使得 其前缀和 + target 等于当前节点的前缀和 currSum
        // 比如, 若当前路径中存在节点A, 其前缀和 + target = 当前节点B的前缀和 currSum, 则说明 A节点->B节点的和为target, 就是答案路径之一
        if (prefixSumCount.find(currSum - targetSum) !=
        prefixSumCount.end()) {
            // 保存以当前节点为终点的路径上存在满足和为target的要求的路径数目
            res += prefixSumCount[currSum - targetSum];
        }
        // 更新路径上当前节点前缀和的个数
        ++prefixSumCount[currSum];
    }
};
```

```

        // 在左子树中递归寻找
        res += dfs(node->left, prefixSumCount, targetSum, currSum);
        // 在右子树中递归寻找
        res += dfs(node->right, prefixSumCount, targetSum, currSum);

        // 回溯，回到本层，恢复状态，去除当前节点的前缀和数量
        --prefixSumCount[currSum];
        return res;
    }
};

```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$, 前缀和 map

508. 出现次数最多的子树元素和

给你一个二叉树的根结点，请你找出出现次数最多的子树元素和。一个结点的「子树元素和」定义为以该结点为根的二叉树上所有结点的元素之和（包括结点本身）。

你需要返回出现次数最多的子树元素和。如果有多个元素出现的次数相同，返回所有出现次数最多的子树元素和（不限顺序）。

示例 1:

输入:

```

5
 / \
2  -3

```

返回 [2, -3, 4], 所有的值均只出现一次，以任意顺序返回所有值。

示例 2:

输入:

```

5
 / \
2  -5

```

返回 [2], 只有 2 出现两次，-5 只出现 1 次。

提示：假设任意子树元素和均可以用 32 位有符号整数表示。

- 深搜

- 思路

- 概括: dfs (自底向上) + hash
- 用深搜自底向上（后序遍历）计算所有子树元素和，同时在dfs过程中用hash表存储各元素和出现的次数，最终遍历该子树和map，取出出现次数最多的子树和即可
- 子树和计算规则: $sum\{root\} = node \rightarrow val + sum\{左子树\} + sum\{右子树\}$

- 代码

```

// 代码参考: https://leetcode-cn.com/problems/most-frequent-subtree-sum/solution/zhong-gui-zhong-ju-shu-de-hou-xu-bian-li-by-jyj4-2/
// 原理参考: https://leetcode-cn.com/problems/most-frequent-subtree-sum/solution/508-chu-xian-ci-shu-zui-duo-de-zi-shu-yuan-su-he-4/
class Solution {
public:

```

```

vector<int> findFrequentTreeSum(TreeNode* root) {
    // 出现次数map, <子树和, 出现次数>
    unordered_map<int, int> sum_freqs;
    // 最多出现的次数
    int max_freq = 0;
    vector<int> res;

    // 递归
    dfs(root, sum_freqs, max_freq);

    // 遍历 sum_freqs, 从中取出出现次数最多的子树和
    for (auto item : sum_freqs) {
        if (item.second == max_freq) {
            res.emplace_back(item.first);
        }
    }
    return res;
}

int dfs(TreeNode *node, unordered_map<int, int> &sum_freqs, int
&max_freq) {
    // 1. 递归出口: 节点为空
    if (node == nullptr) {
        return 0;
    }
    // 2. 本层要做的事情: 统计当前子树和
    int curr_sum = node->val + dfs(node->left, sum_freqs, max_freq)
+ dfs(node->right, sum_freqs, max_freq);
    // 更新当前子树和出现次数
    ++sum_freqs[curr_sum];

    // 更新最多出现的次数
    max_freq = max(max_freq, sum_freqs[curr_sum]);

    // 3. 返回当前子树和
    return curr_sum;
}
};

```

复杂度

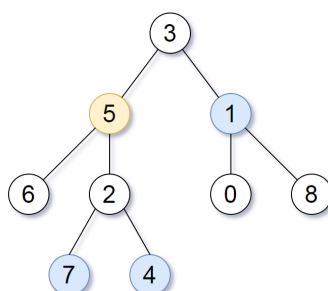
- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$, 子树和 map

863. 二叉树中所有距离为 K 的结点

给定一个二叉树（具有根结点 root），一个目标结点 target，和一个整数值 K。

返回到目标结点 target 距离为 K 的所有结点的值的列表。答案可以以任何顺序返回。

示例 1:



输入: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, K = 2

输出: [7,4,1]

解释:

所求结点为与目标结点 (值为 5) 距离为 2 的结点,
值分别为 7, 4, 以及 1

注意, 输入的 "root" 和 "target" 实际上是树上的结点。
上面的输入仅仅是对这些对象进行了序列化描述。

- 以 target 为原点, 向外扩散
 - 以target为原点, 向三个方向扩散: 每次距离加1, 正好为k的时候, 在queue里的节点就是答案。坑: 这里遍历会三个方向存在循环的问题, 所以需要记录哪些结点已经遍历了。
 - 父节点: 构建 node到parent的map, 根据map向外扩散
 - 左子节点
 - 右子节点
 - 代码

```
class Solution {
public:
    unordered_map<TreeNode*, TreeNode*> nodes_2_parent_;
    vector<int> distanceK(TreeNode* root, TreeNode* target, int k) {
        vector<int> res;
        // 1. 构建父子关系
        buildParent(root, NULL);

        // 2. 以target为原点, 准备从3个方向向外扩散
        unordered_set<TreeNode*> visited;
        queue<TreeNode*> q;
        q.push(target);
        visited.insert(target);
        int distance = 0;

        while (!q.empty()) {
            // 先校验距离k, 防止出现k=0情况
            if (distance == k) {
                while (!q.empty()) {
                    res.push_back(q.front()->val);
                    q.pop();
                }
                return res;
            }

            // 每扩散一层, distance+1
            int size = q.size();
            for (int i = 0; i < size; ++i) {
                TreeNode *cur = q.front();
                q.pop();

                if (cur->left != NULL && visited.find(cur->left) ==
visited.end()) {
                    q.push(cur->left);
                    visited.insert(cur->left);
                }

                if (cur->right != NULL && visited.find(cur->right) ==
visited.end()) {
```

```

        q.push(cur->right);
        visited.insert(cur->right);
    }

    TreeNode *parent = nodes_2_parent_[cur];
    if (parent != NULL && visited.find(parent) ==
visited.end()) {
        q.push(parent);
        visited.insert(parent);
    }

    ++distance;
}
return res;
}

void buildParent(TreeNode *cur, TreeNode *parent) {
    if (cur != NULL) {
        nodes_2_parent_[cur] = parent;
        buildParent(cur->left, cur);
        buildParent(cur->right, cur);
    }
}
};

```

○ 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(1)$

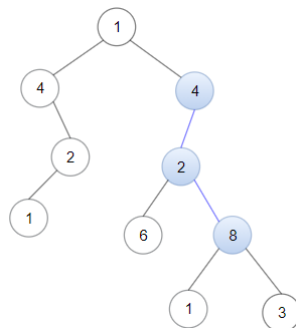
1367. 二叉树中的列表

给你一棵以 root 为根的二叉树和一个 head 为第一个节点的链表。

如果在二叉树中，存在一条一直向下的路径，且每个点的数值恰好一一对应以 head 为首的链表中每个节点的值，那么请你返回 True，否则返回 False。

一直向下的路径的意思是：从树中某个节点开始，一直连续向下的路径。

示例 1：

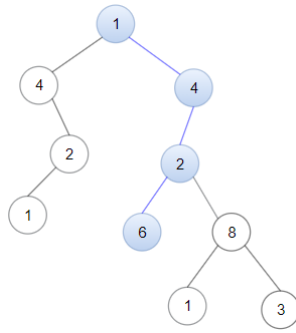


输入：head = [4,2,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,1,3]

输出：true

解释：树中蓝色的节点构成了与链表对应的子路径。

示例 2：



输入: head = [1,4,2,6], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]

输出: true

示例 3:

输入: head = [1,4,2,6,8], root = [1,4,4,null,2,2,null,1,null,6,8,null,null,null,null,1,3]

输出: false

解释: 二叉树中不存在——对应链表的路径。

• 深搜

◦ 思路:

- 枚举二叉树中的每个节点为起点往下的路径是否有链表相匹配的路径。为了判断是否匹配,可设计一个递归函数 `dfs(rt, head)`, 其中 `rt` 表示当前匹配到的二叉树节点, `head` 表示当前匹配到的链表节点, 整个函数返回布尔值表示是否有一条该节点往下的路径与 `head` 节点开始的链表匹配, 如匹配返回 `true`, 否则返回 `false`, 一共有四种情况
 - 链表已经全部匹配完, 匹配成功, 返回 `true`
 - 二叉树访问到了空节点, 匹配失败, `false`
 - 当前匹配的二叉树上节点值与链表节点值不相等, 匹配失败, 返回 `false`
 - 前三种情况都不满足, 说明匹配成功了一部分, 需要继续递归匹配。所以先调用函数 `dfs(rt->left, head->next)`, 其中 `rt->left` 表示该节点的左子节点, `head->next` 表示下一个链表节点, 如果返回的是 `false`, 说明没有找到相匹配的路径, 需要在右子树中匹配, 继续递归调用函数 `dfs(rt->right, head->next)` 去找是否有相匹配的路径, 其中 `rt->right` 表示右子节点, `head->next` 表示下一个链表节点
- 匹配函数确定了, 剩下只需要枚举即可。从根节点开始, 如果当前节点匹配成功就直接返回 `true`, 否则继续找其左儿子和右儿子是否满足, 即 `dfs(root, head) || isSubPath(head, root->left) || isSubPath(head, root->right)`, 不断递归调用即可

◦ 代码

```
// https://leetcode-cn.com/problems/linked-list-in-binary-tree/solution/er-cha-shu-zhong-de-lie-biao-by-leetcode-solution/
// 深搜
class Solution {
public:
    bool isSubPath(ListNode* head, TreeNode* root) {
        if (root == nullptr) {
            return false;
        }
        return dfs(root, head) || isSubPath(head, root->left) ||
isSubPath(head, root->right);
    }
    bool dfs(TreeNode *root, ListNode *head) {
        // 链表已经全部匹配玩, 匹配成功
```

```

    if (head == nullptr) {
        return true;
    }
    // 二叉树访问到了空节点，匹配失败
    if (root == nullptr) {
        return false;
    }
    // 当前匹配的二叉树上节点的值与链表节点的值不相等，匹配失败
    if (root->val != head->val) {
        return false;
    }
    return dfs(root->left, head->next) || dfs(root->right, head->next);
}
};

```

复杂度

- 时间复杂度：\$O(n \times \min(2^{\text{len} + 1}, n))\$。最坏情况下需要对所有节点进行匹配。假设一共有 n 个节点，对于一个节点为根的子树，如果是满二叉树，且每次匹配均为链表的最后一个节点的时候匹配失败，那么一共被遍历到的节点数为 $2^{\text{len} + 1} - 1$ ，即这个节点为根的子树往下 len 层的满二叉树的节点数，其中 len 为链表的长度，而二叉树总结点数最多 n 个，所以枚举节点最多匹配 $\min(2^{\text{len} + 1}, n)$ 次，最坏情况下需要 $O(n \times \min(2^{\text{len} + 1}, n))$ 的时间复杂度
- 空间复杂度：\$O(\text{height})\$ 递归最多不超过二叉树的高度

二叉搜索树

概念

参考

- <https://leetcode-cn.com/problems/same-tree/solution/xie-shu-suan-fa-de-tao-lu-kuang-jia-by-wei-lai-bu/>

96. 不同的二叉搜索树

给定一个整数 n ，求以 $1 \dots n$ 为节点组成的二叉搜索树有多少种？

示例：

输入：3

输出：5

解释：

给定 $n = 3$ ，一共有 5 种不同结构的二叉搜索树：

```

    1   3   3   2   1
     \  /  /  /\  \
    3  2  1  1  3  2
     /  /  \      \
    2  1   2      3

```

- 动态规划
 - 思路

- 思路：给定一个有序序列 $1 \dots n$ ，为了构建出一颗二叉搜索树，我们可以 **遍历每个数组 i ，将该数字作为树根节点，将 $1 \dots (i - 1)$ 序列作为左子树，将 $(i + 1) \dots n$ 序列作为右子树。接着按照同样方式递归构建左子树和右子树**。在上述构建过程中，由于根节点不同，因此能保证每棵二叉搜索树是唯一的。

由此可见，原问题可被分解为规模较小的两个字问题，且子问题的解可以复用。因此可使用动态规划解决本问题

- 算法：题目要求计算不同二叉搜索树的个数，为此定义两个函数
 - $G(n)$: 长度为 n 的序列能构成的不同二叉搜索树的个数
 - $F(i, n)$: 以 i 为根、序列长度为 n 的不同二叉搜索树个数 $(1 \leq i \leq n)$ 。

可见， $G(n)$ 是我们要求解的函数，又由后面可知， $G(n)$ 可从 $F(i, n)$ 得到，而 $F(i, n)$ 又递归依赖 $G(n)$

- 首先由上可知，不同二叉搜索树的总数 $G(n)$ ，是对遍历所有 $(1 \leq i \leq n)$ 的 $F(i, n)$ 之和，即

$$G(n) = \sum_{i=1}^n F(i, n) \quad (1)$$

- 对于边界情况，当序列长度为 1 （只有跟）或为 0 （空树）时，只有一种情况，即 $G(0) = 1$, $G(1) = 1$

给定序列 $1 \dots n$ ，若选择数字 i 为根，则根为 i 的所有二叉搜索树的集合是左子树集合和右子树集合的 **笛卡尔积**，对于笛卡尔积中的每个元素，加上根节点之后形成完成的二叉搜索树，如下图所示



举例而言，创建以 3 为根、长度为 7 的不同二叉搜索树，整个序列是 $[1, 2, 3, 4, 5, 6, 7]$ ，我们需要从左子序列 $[1, 2]$ 构建左子树，从右子序列 $[4, 5, 6, 7]$ 构建右子树，然后将它们组合（即笛卡尔积）。

对于这个例子，不同二叉搜索树的个数为 $F(3, 7)$ 。我们将 $[1, 2]$ 构建不同左子树的数量表示为 $G(2)$ ，从 $[4, 5, 6, 7]$ 构建不同右子树的数量表示为 $G(4)$ ，注意到 $G(n)$ 和序列的内容无关，只和序列的长度有关。于是， $F(3, 7) = G(2) \cdot G(4)$ 。因此，我们可以得到以下公式：

$$F(i, n) = G(i-1) \cdot G(n-i) \quad \text{tag2}$$

将上述公式 $(1), (2)$ 结合，可得到 $G(n)$ 的递归表达式

$$G(n) = \sum_{i=1}^n G(i-1) \cdot G(n-i) \quad \text{tag3}$$

故，只需从从小到计算 G 函数即可

- 代码

```
class Solution {
public:
    // 动态规划
    int numTrees(int n) {
        vector<int> G(n + 1, 0);
        G[0] = 1;
        G[1] = 1;
        // G[i]表示长度为i的二叉搜索树的个数
        for (int i = 2; i <= n; ++i) {
            // j表示以j为根节点分隔 长度为i的序列
            for (int j = 1; j <= i; ++j) {
                G[i] += G[j - 1] * G[i - j];
            }
        }
        return G[n];
    }
};
```

- 复杂度

- 时间复杂度: $O(n^2)$, 其中 n 表示二叉搜索树的节点个数, $G(n)$ 函数一共有 n 个值需要求解, 每次求解需要 $O(n)$ 的时间复杂度, 故总时间复杂度为 $O(n^2)$
 - 空间复杂度: $O(n)$, 需要 $O(n)$ 的空间存储 G 数组

- 卡特兰数 (数学方法)

- 思路

事实上动态规划方法推导出的 $G(n)$ 函数的值在数学上被称为 [卡特兰数 \$C_n\$](#) 卡特兰数更便于计算的定义如下

$$C_0 = 1, C_{n+1} = \frac{2(2n+1)}{n+2} C_n$$

证明过程在以上链接

- 代码

```
class Solution {
public:
    // 卡特兰数
    int numTrees(int n) {
        long long c = 1;
        for (int i = 0; i < n; ++i) {
            c = c * 2 * (2 * i + 1) / (i + 2);
        }
        return (int)c;
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$, 其中 n 表示二叉搜索树的节点个数, 只需要循环遍历一次即可
- 空间复杂度: $O(1)$, 只需要常数空间存放若干变量

- 参考

- <https://leetcode-cn.com/problems/unique-binary-search-trees/solution/bu-tong-de-er-c-ha-sou-suo-shu-by-leetcode-solution/>

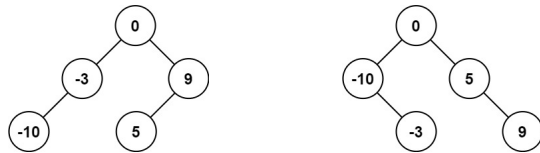
- <https://leetcode-cn.com/problems/unique-binary-search-trees/solution/shou-hua-tu-jie-san-chong-xie-fa-dp-di-gui-ji-yi-h/>

108. 将有序数组转换为二叉搜索树

给你一个整数数组 `nums`，其中元素已经按升序排列，请你将其转换为一棵高度平衡二叉搜索树。

高度平衡二叉树是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1」的二叉树。

示例 1：

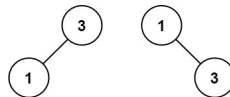


输入：nums = [-10,-3,0,5,9]

输出：[0,-3,9,-10,null,5]

解释：[0,-10,5,null,-3,null,9] 也将被视为正确答案：

示例 2：



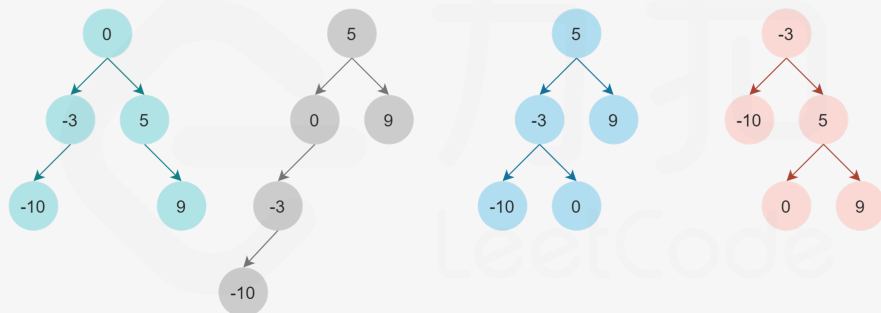
输入：nums = [1,3]

输出：[3,1]

解释：[1,3] 和 [3,1] 都是高度平衡二叉搜索树。

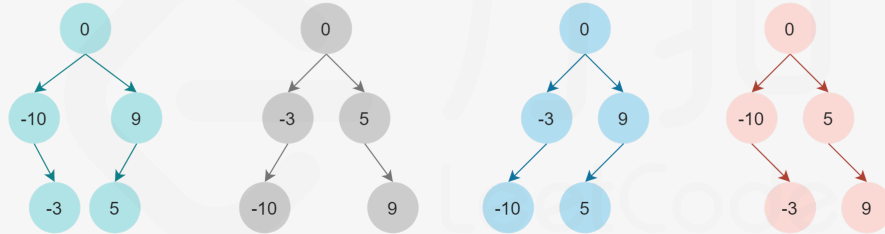
- 递归
 - 思路
 - 二叉搜索树的中序遍历是升序序列，题目给定的数组是按照升序排序的有序数组，可确保数组是二叉搜索树的中序遍历序列
 - 给定二叉搜索树的中序遍历，是否可确定唯一的二叉搜索树？答案是 **否定的**，如果没有要求强制平衡，则任何一个数字都可作为二叉搜索树的根节点，故可能的二叉搜索树有多个

以下 BST 的中序遍历结果均为 [-10, -3, 0, 5, 9]



- 若要求二叉搜索树的高度平衡，也不能确定唯一的二叉搜索树

以下 BST 的中序遍历结果均为 $[-10, -3, 0, 5, 9]$



- 构建高度平衡二叉搜索树的思路：直观地看，可选择中间数字作为二叉搜索树的根节点，这样左右子树的元素个数相同或仅差一个，可使得树平衡。若数组长度为偶数，则可选择中间位置左边或右边的数字作为根节点；确定根节点后，则数组分为左右子树，后续使用递归方式创建即可
- 递归的出口：当前数组方位 $[\text{left}, \text{right}]$ 为空，即 $\text{left} > \text{right}$

○ 代码

```
// 参考 https://leetcode-cn.com/problems/convert-sorted-array-to-binary-search-tree/solution/jiang-you-xu-shu-zu-zhuan-huan-wei-er-cha-sou-s-33/
class Solution {
public:
    TreeNode* helper(vector<int> &nums, int left, int right) {
        // 递归出口
        if (left > right) { return nullptr; }

        // 总数选择数组中间位置左边的元素作为根节点
        int mid = (left + right) / 2;

        // 创建当前子树根节点
        TreeNode *root = new TreeNode(nums[mid]);
        // 创建左子树
        root->left = helper(nums, left, mid - 1);
        // 创建右子树
        root->right = helper(nums, mid + 1, right);
        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return helper(nums, 0, nums.size() - 1);
    }
};
```

○ 复杂度

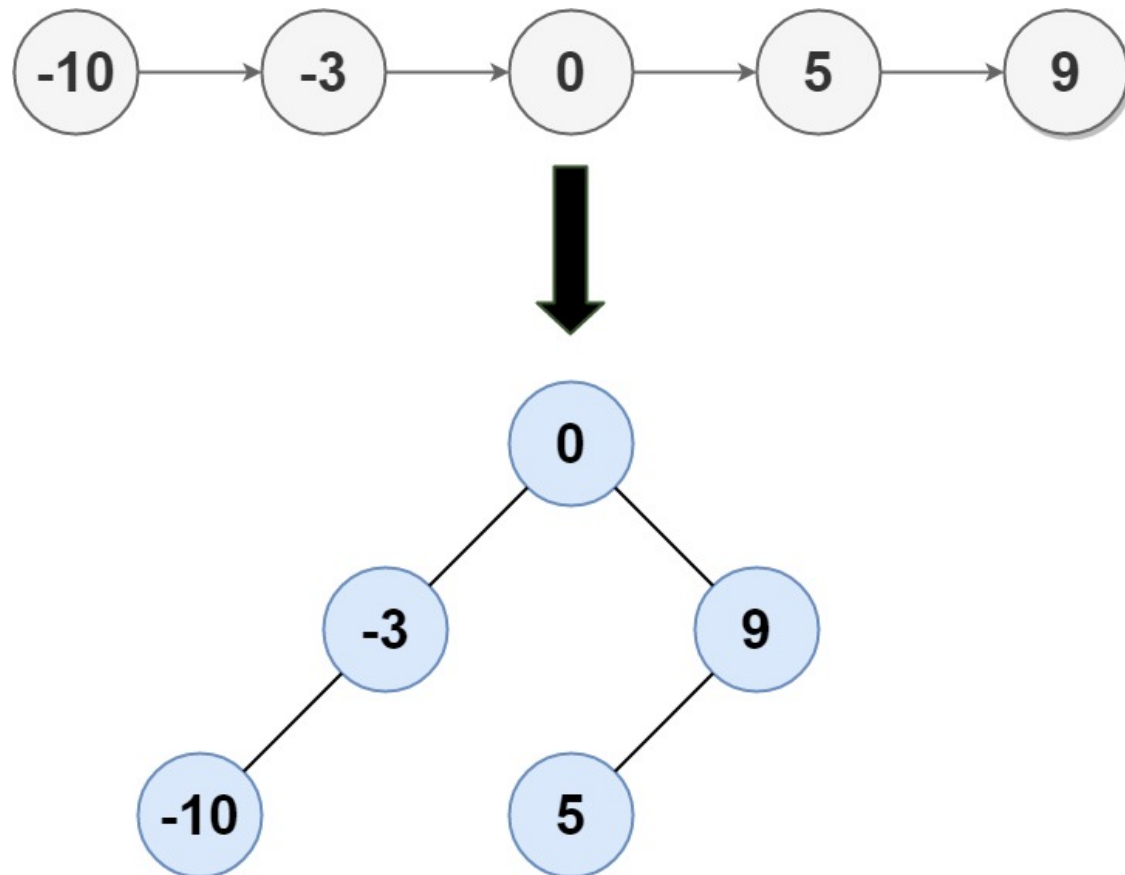
- 时间复杂度： $O(n)$ ，其中 n 为数组长度。每个数字只访问一次
- 空间复杂度： $O(\log n)$ ，其中 n 为数组长度，空间复杂度不考虑返回值，因此空间复杂度主要取决于递归栈的深度，递归栈的深度是 $O(\log n)$

109. 有序链表转换二叉搜索树

给定一个单链表的头节点 `head`，其中的元素 **按升序排序**，将其转换为高度平衡的二叉搜索树。

本题中，一个高度平衡二叉树是指一个二叉树每个节点的左右两个子树的高度差不超过 1。

示例 1:



输入: head = [-10,-3,0,5,9]

输出: [0,-3,9,-10,null,5]

解释: 一个可能的答案是[0, -3, 9, -10, null, 5], 它表示所示的高度平衡的二叉搜索树。

示例 2:

输入: head = []

输出: []

提示:

- head 中的节点数在 $[0, 2 * 10^4]$ 范围内
 - $-105 \leq \text{Node.val} \leq 105$
- 递归: 每次构造节点, 都获取当前链表的中间节点, 填充值

```
class Solution {
public:
    ListNode* getMedian(ListNode* left, ListNode* right) {
        ListNode* fast = left;
        ListNode* slow = left;
        while (fast != right && fast->next != right) {
            fast = fast->next;
            fast = fast->next;
            slow = slow->next;
        }
        return slow;
    }
}
```

```

TreeNode* buildTree(ListNode* left, ListNode* right) {
    if (left == right) {
        return nullptr;
    }
    ListNode* mid = getMedian(left, right);
    TreeNode* root = new TreeNode(mid->val);
    root->left = buildTree(left, mid);
    root->right = buildTree(mid->next, right);
    return root;
}

TreeNode* sortedListToBST(ListNode* head) {
    return buildTree(head, nullptr);
}
};

```

- 递归（优化版）：先构造节点，进行中序遍历。后填充值

```

class Solution {
public:
    TreeNode* sortedListToBST(ListNode* head) {
        int length = getLength(head);
        TreeNode *root = dfs(head, 0, length - 1); // 这里使用dfs(head, 1, length);也行
        return root;
    }
    int getLength(ListNode* head) {
        int length = 0;
        while (head != nullptr) {
            ++length;
            head = head->next;
        }
        return length;
    }
    TreeNode* dfs(ListNode* &head, int start, int end) {
        if (start > end) { return nullptr; }
        int mid = (start + end) / 2;
        TreeNode* root = new TreeNode();
        root->left = dfs(head, start, mid - 1);

        root->val = head->val;
        head = head->next;

        root->right = dfs(head, mid + 1, end);
        return root;
    }
};

```

95. 不同的二叉搜索树 II

给定一个整数 n ，生成所有由 $1 \dots n$ 为节点所组成的 二叉搜索树。

示例：

输入：3

输出：

```
[
  [1,null,3,2],
  [3,2,null,1],
  [3,1,null,null,2],
  [2,1,3],
  [1,null,2,null,3]
]
```

解释：

以上的输出对应以下 5 种不同结构的二叉搜索树：

```

1      3 3  2  1
 \    / /  /\  \
 3  2 1  1 3  2
 /  /  \      \
2  1    2      3
```

- 递归
 - 思路
 - **构建一棵二叉搜索树**：只需要选择一个根节点，然后递归去创建左右子树

```
TreeNode *helper(int start, int end) {
    if (start > end) { return nullptr; }
    // 此处可选择[start, end]区间中任意值作为根节点，这里选择中点，则构建出来的是一棵平衡二叉搜索树
    int val = (start + end) / 2;
    TreeNode root = new TreeNode(val);
    root->left = helper(1, val - 1);
    root->right = helper(val + 1, end);
    return root;
}

TreeNode *createBinaryTree(int n) {
    return helper(1, n);
}
```

- **构建多颗二叉搜索树**：选择不同根节点，构建不同树和子树，存储到数组中，在上面代码中，可修改选择根节点代码
 - 遍历序列，以每个节点作为根节点，然后递归构建左右子树，注意一个节点为根节点的二叉搜索树个数为 **左子树个数*右子树个数**
 - 递归构建左、右子树，可拿到左子树所有可能的根节点列表left、right
 - 需要注意构建root节点的实际
- 代码

```
class Solution {
public:
    vector<TreeNode*> helper(int start, int end) {
        vector<TreeNode*> res;
```

```

        if (start > end) {
            // 加nullptr是为了防止左子树或右子树列表为空的情况下，另一个无法遍历的
情况
            res.push_back(nullptr);
            return res;
        }
        for (int i = start; i <= end; ++i) {
            // 不能放在这里，否则以root为根的所有子树共用同一个root，后边遍历左右
子树的时候，只会修改left、right，所有树都是同一个了
            // TreeNode *root = new TreeNode(i);
            vector<TreeNode*> left = helper(start, i - 1);
            vector<TreeNode*> right = helper(i + 1, end);

            // 从左子树集合中选出一棵左子树，从右子树集合中选出一棵右子树，拼接到根
节点上
            for (TreeNode *l : left) {
                for (TreeNode *r : right) {
                    TreeNode *root = new TreeNode(i);
                    root->left = l;
                    root->right = r;
                    res.push_back(root);
                }
            }
            return res;
        }
        vector<TreeNode*> generateTrees(int n) {
            vector<TreeNode*> res = helper(1, n);
            return res;
        }
    };

```

复杂度

- 时间复杂度： $O(\frac{4^n}{n^{\frac{1}{2}}})$ ，整个算法时间复杂度取决于 **可行二叉搜索树的个数**，对于 n 个点生成的二叉搜索树刷领等价于数学上第 n 个 **卡特兰数** $G(n)$ 。生成一棵二叉搜索树需要 $O(n)$ 的复杂度，一共有 $G(n)$ 棵，即 $O(nG(n))$ 。而卡特兰数以 $O(\frac{4^n}{n^{\frac{1}{2}}})$ 增长，故总时间复杂度为 $O(\frac{4^n}{n^{\frac{1}{2}}})$
- 空间复杂度： $O(\frac{4^n}{n^{\frac{1}{2}}})$ ， n 个点平衡的二叉搜索树有 $G(n)$ 棵，每棵有 n 个节点，因此存储的空间需要 $O(nG(n)) = O(\frac{4^n}{n^{\frac{1}{2}}})$ ，递归函数需要 $O(n)$ 的栈空间，故总空间复杂度为 $O(\frac{4^n}{n^{\frac{1}{2}}})$

参考

- <https://leetcode-cn.com/problems/unique-binary-search-trees-ii/solution/cong-gou-jian-dan-ke-shu-dao-gou-jian-suo-you-shu/>

98. 验证二叉搜索树

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

节点的左子树只包含小于当前节点的数。

节点的右子树只包含大于当前节点的数。

所有左子树和右子树自身必须也是二叉搜索树。

示例 1：

输入:

2

/ \

1 3

输出: true

示例 2:

输入:

5

/ \

1 4

/ \

3 6

输出: false

解释: 输入为: [5,1,4,null,null,3,6]。

根节点的值为 5，但是其右子节点值为 4。

- 递归

- 思路

- 二叉搜索树性质

- 若左子树不为空，则左子树上的所有节点的值均小于它的根节点的值
 - 若右子树不为空，则右子树上的所有节点的值均大于它的根节点的值
 - 左右子树也都为二叉搜索树

- 由上性质，可设计一个递归函数 `helper(root, lower, upper)` 来递归判断，函数表示以 `root` 为根的子树，判断子树中所有节点的值是否都在 (l, r) 范围内 **注意开区间**。如果不在则说明不满足二叉搜索树，直接返回。否则继续递归调用判断其左右子树

- 故由上，递归调用左子树时，上界 `upper` 要修改为 `root->val`，因为左子树中的所有节点的值都小于根节点的值。递归调用右子树时，下界 `lower` 要修改为 `root->val`，因为右子树中的所有节点的值都大于根节点的值。

- 代码

```
class Solution {
public:
    bool helper(TreeNode *root, long long lower, long long upper) {
        if (root == nullptr) { return true; }
        if (root->val <= lower || root->val >= upper) { return false; }
        return helper(root->left, lower, root->val) && helper(root->right, root->val, upper);
    }
    bool isValidBST(TreeNode *root) {
        return helper(root, LONG_MIN, LONG_MAX);
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$ ，其中 n 为二叉树节点个数，递归调用时每个节点最多被访问一次
 - 空间复杂度: $O(n)$ ，其中 n 为二叉树节点个数。递归函数在递归过程中需要为每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归深度，即二叉树高度。最坏情况下二叉树为一条链，树的高度为 n ，递归最深达到 n 层，故最坏情况下空间复杂度为 $O(n)$

- 中序遍历 (迭代)

- 思路

基于上边方法提到的性质，可知二叉搜索树的 **中序遍历** 得到的值的序列一定是升序的，故也可在中序遍历的时候实时检查当前节点的值是否大于前一个中序遍历到的节点的值即可。如果大于说明是升序的，整棵树是二叉搜索树，否则不是。下面用栈模拟中序遍历

- 代码

```
// 参考 https://leetcode-cn.com/problems/validate-binary-search-tree/solution/yan-zheng-er-cha-sou-suo-shu-by-leetcode-solution/
class Solution {
public:
    bool isValidBST(TreeNode *root) {
        stack<TreeNode*> s;
        long long inorder = LONG_MIN;
        while (root != nullptr || !s.empty()) {
            if (root != nullptr) {
                s.push(root);
                root = root->left;
            } else {
                TreeNode *cur = s.top();
                s.pop();
                if (cur->val <= inorder) {
                    return false;
                }
                inorder = cur->val;
                root = cur->right;
            }
        }
        return true;
    }
};
```

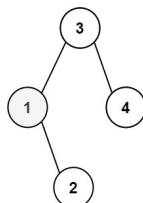
- 复杂度

- 时间复杂度： $O(n)$ ，其中 n 为二叉树的节点个数。二叉树的每个节点最多被访问一次，因此时间复杂度为 $O(n)$ 。
- 空间复杂度： $O(n)$ ，其中 n 为二叉树的节点个数。栈最多存储 n 个节点，因此需要额外的 $O(n)$ 的空间。

230. 二叉搜索树中第K小的元素

给定一个二叉搜索树的根节点 $root$ ，和一个整数 k ，请你设计一个算法查找其中第 k 个最小元素（从 1 开始计数）。

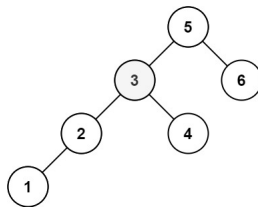
示例 1:



输入: $root = [3,1,4,null,2]$, $k = 1$

输出: 1

示例 2:



输入: root = [5,3,6,2,4,null,null,1], k = 3

输出: 3

- 中序遍历 (可递归、可迭代)

- 思路

通过构造bst的中序遍历序列, 则第 $k-1$ 个元素就是第 k 小的元素

- 代码

```
// https://leetcode-cn.com/problems/kth-smallest-element-in-a-bst/solution/er-cha-sou-suo-shu-zhong-di-kxiao-de-yuan-su-by-le/
// 递归
class solution {
public:
    void dfs(TreeNode *root, vector<int> &res) {
        if (root == nullptr) {
            return;
        }
        dfs(root->left, res);
        res.push_back(root->val);
        dfs(root->right, res);
    }
    int kthSmallest(TreeNode* root, int k) {
        vector<int> res;
        dfs(root, res);
        return res[k - 1];
    }
};
```

- 复杂度

- 时间复杂度: $O(n)$
- 空间复杂度: $O(n)$, 用了一个数组存储中序序列

- 记录子树节点数目

- 思路

二叉搜索树性质 (左子树值 < root节点 < 右子树值)

- 代码

```
class solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        if (root == nullptr) { return 0; }
        int left_nodes_num = getNodesNum(root->left);
        int right_nodes_num = getNodesNum(root->right);
        if (left_nodes_num + 1 == k) {
            return root->val;
        } else if (left_nodes_num >= k) {
            return kthSmallest(root->left, k);
        } else {
```

```

        return kthSmallest(root->right, k - left_nodes_num - 1);
    }
    return 0;
}
int getNodesNum(TreeNode* root) {
    if (root == nullptr) { return 0; }
    return 1 + getNodesNum(root->left) + getNodesNum(root->right);
}
};

```

○ 复杂度

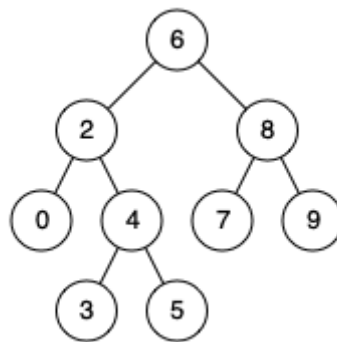
- 时间复杂度：当树是平衡树时，时间复杂度取得最小值 $O(\log N)$ $O(\log N)$ $O(\log N)$ ；当树是线性树时，时间复杂度取得最大值 $O(N)$ $O(N)$ $O(N)$
- 空间复杂度： $O(1)$ 。

235. 二叉搜索树的最近公共祖先

给定一个二叉搜索树，找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个结点 p 、 q ，最近公共祖先表示为一个结点 x ，满足 x 是 p 、 q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

例如，给定如下二叉搜索树：root = [6,2,8,0,4,7,9,null,null,3,5]



示例 1:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

输出: 6

解释: 节点 2 和节点 8 的最近公共祖先是 6。

示例 2:

输入: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4

输出: 2

解释: 节点 2 和节点 4 的最近公共祖先是 2, 因为根据定义最近公共祖先节点可以为节点本身。

• 递归

```

class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
    {
        if (root == NULL) { return NULL; }
        if (root == p || root == q) { return root; }
        if (root->val > p->val && root->val > q->val) {
            return lowestCommonAncestor(root->left, p, q);
        }
    }
};

```

```

    } else if (root->val < p->val && root->val < q->val) {
        return lowestCommonAncestor(root->right, p, q);
    } else {
        return root;
    }
    return NULL;
}
};

```

- 迭代

- 思路：注意是 **二叉搜索树**

- 如果两个节点值都小于根节点，说明都在左子树上
- 如果两个节点值都大于根节点，说明都在右子树上
- 如果一个节点值大于根节点，另一个小于根节点，说明两个节点分别位于两个子树上。那么当前根节点就是他们的最近公共祖先节点

- 代码

```

// https://leetcode-cn.com/problems/lowest-common-ancestor-of-a-binary-
// search-tree/solution/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian-26/
// 迭代
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p,
    TreeNode* q) {
        TreeNode *ancestor = root;
        while (true) {
            if (p->val < ancestor->val && q->val < ancestor->val) {
                ancestor = ancestor->left;
            } else if (p->val > ancestor->val && q->val > ancestor->val)
            {
                ancestor = ancestor->right;
            } else {
                break;
            }
        }
        return ancestor;
    }
};

```

- 复杂度

- 时间复杂度： $O(n)$
- 空间复杂度： $O(1)$

449. 序列化和反序列化二叉搜索树

序列化是将数据结构或对象转换为一系列位的过程，以便它可以存储在文件或内存缓冲区中，或通过网络连接链路传输，以便稍后在同一个或另一个计算机环境中重建。

设计一个算法来序列化和反序列化 二叉搜索树 。 对序列化/反序列化算法的工作方式没有限制。 您只需确保二叉搜索树可以序列化为字符串，并且可以将该字符串反序列化为最初的二叉搜索树。

编码的字符串应尽可能紧凑。

示例 1:

输入: root = [2,1,3]

输出: [2,1,3]

示例 2:

输入: root = []

输出: []

注意: 不要使用类成员/全局/静态变量来存储状态。你的序列化和反序列化算法应该是无状态的。

- 深搜
 - 思路
 - 题目其实就是把二叉树转化成一个字符串，并且能通过字符串还原成原来的二叉树即可
 - 注意，二叉搜索树可使用前序遍历或后序遍历恢复，二叉树不行喔
 - 代码

```
class Codec {
public:

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        string data;
        if (root == nullptr) { return data; }
        preorder(root, data);
        return data;
    }
    void preorder(TreeNode* root, string &data) {
        if (root == nullptr) {
            data += "# ";
            return;
        }
        data += std::to_string(root->val) + " ";
        preorder(root->left, data);
        preorder(root->right, data);
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if (data.empty()) { return nullptr; }
        stringstream ss(data);
        return decoder(ss);
    }

    TreeNode* decoder(stringstream &ss) {
        string t;
        ss >> t;
        if (t == "#") {
            return nullptr;
        }
        TreeNode* root = new TreeNode(stoi(t));
        root->left = decoder(ss);
        root->right = decoder(ss);
        return root;
    }
};
```

- 复杂度
 - 时间复杂度： $O(n)$ ，其中 n 为树节点数目
 - 空间复杂度： $O(n)$ ，其中 n 为树节点数目

450. 删除二叉搜索树中的节点

给定一个二叉搜索树的根节点 `root` 和一个值 `key`，删除二叉搜索树中的 `key` 对应的节点，并保证二叉搜索树的性质不变。返回二叉搜索树（有可能被更新）的根节点的引用。

一般来说，删除节点可分为两个步骤：

首先找到需要删除的节点；

如果找到了，删除它。

说明：要求算法时间复杂度为 $O(h)$ ， h 为树的高度。

示例：

`root = [5,3,6,2,4,null,7]`

`key = 3`

```

5
 / \
3   6
 / \ \
2  4 7

```

给定需要删除的节点值是 3，所以我们首先找到 3 这个节点，然后删除它。

一个正确的答案是 `[5,4,6,2,null,null,7]`，如下图所示。

```

5
 / \
4   6
 /  \
2    7

```

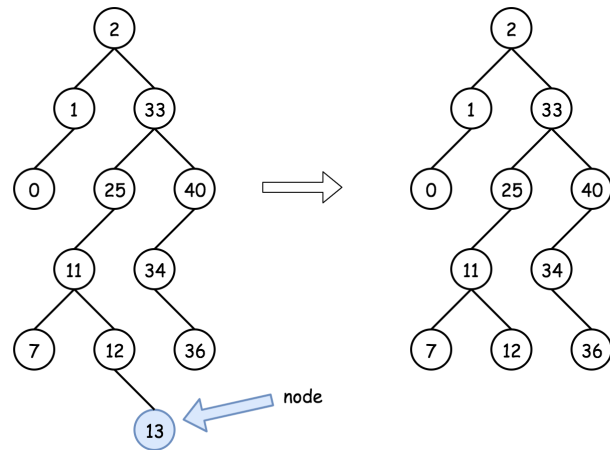
另一个正确答案是 `[5,2,6,null,4,null,7]`。

```

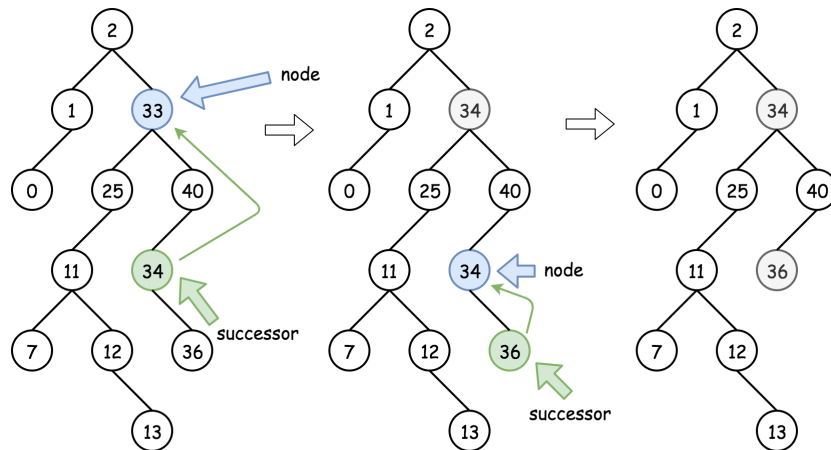
5
 / \
2   6
 \  \
  4  7

```

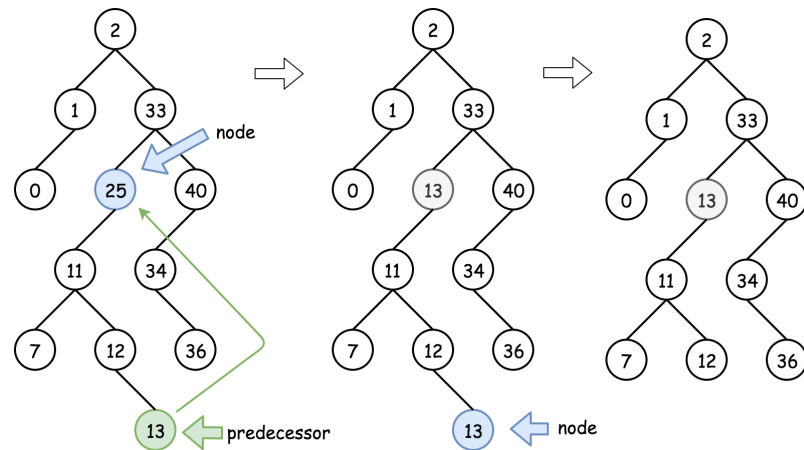
- 递归
 - 思路：删除存在三种情况
 - 要删除的节点为叶子节点，可直接删除



- 要删除的节点不是叶子节点，但有右子节点，则该节点可由其后继节点替代。后继节点位于右子树中较低的位置，然后递归删除其后继节点



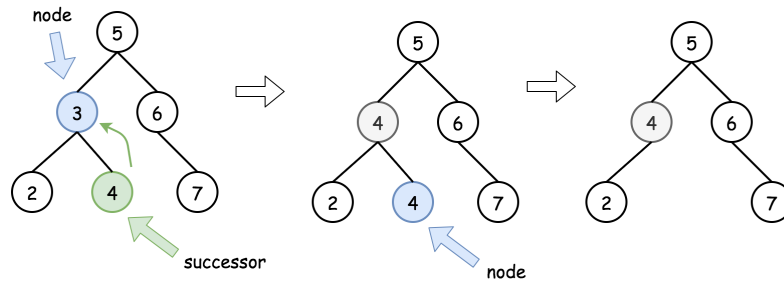
- 要删除的节点不是叶子节点，但有左子节点，且没有右子节点。可由前驱结点替代，进而递归删除其前驱结点



○ 算法

- 如果 $key > root \rightarrow val$ ，则说明要删除的节点在右子树， $root \rightarrow right = deleteNode(root \rightarrow right, key)$
- 如果 $key < root \rightarrow val$ ，则说明要删除的节点在左子树， $root \rightarrow left = deleteNode(root \rightarrow left, key)$
- 如果 $key == root \rightarrow val$ ，则说明该节点就是要删除的节点，则
 - 如果该节点是叶子节点，直接删除： $root == null$
 - 如果该节点不是叶子节点，且有右子节点，则用后继节点的值替代 $root \rightarrow val = successor \rightarrow val$ ，然后删除后继节点

- 如果该节点不是叶子节点，且只有左节点，则用其前驱结点的替代 `root->val = predecessor->val`，然后删除前驱结点
- 返回 `root`



代码

// <https://leetcode-cn.com/problems/delete-node-in-a-bst/solution/shan-chu-er-cha-sou-suo-shu-zhong-de-jie-dian-by-l/>

// 递归

```
class Solution {
public:
    int predecessor(TreeNode* root) {
        root = root->left;
        while (root->right != nullptr) {
            root = root->right;
        }
        return root->val;
    }
    int successor(TreeNode* root) {
        root = root->right;
        while (root->left != nullptr) {
            root = root->left;
        }
        return root->val;
    }
    TreeNode* deleteNode(TreeNode* root, int key) {
        if (root == nullptr) {
            return nullptr;
        }

        // key在右子树中
        if (key > root->val) {
            root->right = deleteNode(root->right, key);
        } else if (key < root->val) {
            // key在左子树中
            root->left = deleteNode(root->left, key);
        } else {
            // 本节点就是待删除节点
            // 叶子节点
            if (root->left == nullptr && root->right == nullptr) {
                root = nullptr;
            } else if (root->right != nullptr) {
                // 非叶子节点，有右子节点，使用后继节点值替换待删除节点
                root->val = successor(root);
                // 递归删除后继节点
                root->right = deleteNode(root->right, root->val);
            } else {
                // 非叶子节点，无右子节点，有左子节点，使用前驱节点替换待删除节点
```

```

        root->val = predecessor(root);
        // 递归删除前驱结点
        root->left = deleteNode(root->left, root->val);
    }
}
return root;
}
};

```

复杂度

- 时间复杂度： $O(\log n)$ ，在算法的执行过程中，我们一直在树上向左或向右移动。首先用 $O(H_1)$ 的时间找到要删除的节点， H_1 指的是从根节点到要删除节点的高度。然后删除节点需要 $O(H_2)$ 的时间， H_2 指的是从要删除节点到替换节点的高度。由于 $O(H_1 + H_2) = O(H)$ ， H 是树的高度，若树是一个平衡树则 $H = \log N$ 。
- 空间复杂度： $O(H)$ ，递归时堆栈使用的空间， H 是树的高度。

501. 二叉搜索树中的众数

给定一个有相同值的二叉搜索树（BST），找出 BST 中的所有众数（出现频率最高的元素）。

假定 BST 有如下定义：

结点左子树中所含结点的值小于等于当前结点的值

结点右子树中所含结点的值大于等于当前结点的值

左子树和右子树都是二叉搜索树

例如：

给定 BST [1,null,2,2],

```

1
 \
  2
 /
2

```

返回[2].

提示：如果众数超过1个，不需考虑输出顺序

进阶：你可以不使用额外的空间吗？（假设由递归产生的隐式调用栈的开销不被计算在内）

中序遍历

思路

- 最朴素的方法是使用哈希表，保存每个元素出现的次数
- 如何不使用哈希表呢？本题是二叉搜索树，而二叉搜索树的中序遍历序列是一个非递减的有序序列。这样，重复出现的数字一定是连续出现的。
- 则可以顺序遍历中序序列，用 `val` 记录当前数字，`count` 记录当前数字重复次数，用 `max_count` 记录目前出现重复数字最多的次数，用 `res` 记录已出现的最多次数的众数。
- 那么每遍历到一个新元素，则
 - 首先更新 `val` 和 `count`
 - 如果该元素和 `base` 相等，那么 `++count`
 - 不相等，则更新 `base`，`count=1`
 - 然后更新 `max_count`

- 如果 `count == max_count`，表示当前数字 `base` 出现的次数等于当前众数出现的次数，将 `base` 加入到 `res` 数组中
- 如果 `count > max_count`，表示当前数字 `base` 出现的次数大于当前众数出现的次数，更新 `max_count = count`，清空 `res` 数组后，将 `base` 加入到 `res` 数组中

○ 代码

```
// https://leetcode-cn.com/problems/find-mode-in-binary-search-tree/solution/er-cha-sou-suo-shu-zhong-de-zhong-shu-by-leetcode-/  
// 中序遍历  
class Solution {  
public:  
    vector<int> res;  
    int val = 0;  
    int count = 0;  
    int max_count = 0;  
    vector<int> findMode(TreeNode* root) {  
        dfs(root);  
        return res;  
    }  
  
    void dfs(TreeNode* root) {  
        if (root == nullptr) { return; }  
        dfs(root->left);  
  
        if (root->val == val) {  
            ++count;  
        } else {  
            val = root->val;  
            count = 1;  
        }  
  
        if (count == max_count) {  
            res.push_back(val);  
        }  
        if (count > max_count) {  
            res.clear();  
            res.push_back(val);  
            max_count = count;  
        }  
  
        dfs(root->right);  
    }  
};
```

○ 复杂度

- 时间复杂度： $O(n)$ ，遍历整棵树
- 空间复杂度： $O(n)$ ，递归栈空间