

=====

文件夹: class031\_BinaryTreeAlgorithms

=====

[Markdown 文件]

=====

文件: BinaryTreeAlgorithmsSummary.md

=====

## # 二叉树算法专题总结

### ## 📚 专题概述

本专题系统整理了二叉树相关的核心算法和解题技巧，涵盖了从基础遍历到高级应用的完整知识体系。通过本专题的学习，你将掌握二叉树算法的精髓，能够应对各种算法面试和工程实践中的树形结构问题。

### ## ⚙️ 核心算法分类

#### #### 1. 基础遍历算法

- \*\*前序遍历\*\*: 根->左->右
- \*\*中序遍历\*\*: 左->根->右 (BST 有序性)
- \*\*后序遍历\*\*: 左->右->根 (树形 DP 基础)
- \*\*层序遍历\*\*: BFS 应用，按层处理

#### #### 2. 递归与分治

- \*\*递归三要素\*\*: 定义、终止条件、递推关系
- \*\*分治思想\*\*: 将问题分解为子问题求解
- \*\*树形 DP\*\*: 后序遍历 + 状态传递

#### #### 3. 二叉搜索树特性

- \*\*中序遍历有序\*\*: 升序序列
- \*\*查找优化\*\*:  $O(h)$  时间复杂度
- \*\*范围查询\*\*: 高效的范围操作

### ## 🔧 工程化考量

#### #### 1. 异常处理策略

```
```java
// 空树处理
if (root == null) return defaultValue;

// 整数溢出防护
long sum = (long)leftSum + rightSum;
```

```
// 递归深度控制  
if (depth > MAX_DEPTH) throw new StackOverflowError();  
```
```

### ### 2. 性能优化技巧

- \*\*剪枝优化\*\*: 提前终止不必要的计算
- \*\*记忆化搜索\*\*: 避免重复子问题计算
- \*\*迭代替代递归\*\*: 防止栈溢出
- \*\*Morris 遍历\*\*:  $O(1)$  空间复杂度

### ### 3. 代码质量规范

```
``` java
```

```
// 清晰的变量命名  
TreeNode current = root;  
Queue<TreeNode> levelQueue = new LinkedList<>();
```

### // 模块化设计

```
private int calculateHeight(TreeNode node) {  
    // 单一职责原则  
}
```

### // 完整的单元测试

```
@Test  
public void testEmptyTree() {  
    assertTrue(solution.isSymmetric(null));  
}  
```
```

## ## 📊 复杂度分析框架

### ### 时间复杂度分析

| 算法类型   | 时间复杂度       | 说明        |
|--------|-------------|-----------|
| 遍历类    | $O(n)$      | 每个节点访问一次  |
| BST 操作 | $O(h)$      | $h$ 为树的高度 |
| 平衡树    | $O(\log n)$ | 树保持平衡状态   |
| 树形 DP  | $O(n)$      | 每个节点处理一次  |

### ### 空间复杂度分析

| 场景     | 空间复杂度  | 说明          |
|--------|--------|-------------|
| 递归栈    | $O(h)$ | 递归调用深度      |
| BFS 队列 | $O(w)$ | $w$ 为树的最大宽度 |

|                        |
|------------------------|
| 显式栈   O(h)   迭代遍历使用    |
| 结果存储   O(n)   需要存储所有节点 |

## ## 🎓 学习路径规划

### #### 第一阶段：基础掌握（1-2 周）

1. \*\*遍历算法\*\*：前中后序 + 层序遍历
2. \*\*基本操作\*\*：最大深度、对称性检查
3. \*\*递归思维\*\*：理解递归调用过程

### #### 第二阶段：进阶应用（2-3 周）

1. \*\*BST 特性\*\*：验证、操作、迭代器
2. \*\*路径问题\*\*：最大路径和、路径总和
3. \*\*构造问题\*\*：遍历序列重建树

### #### 第三阶段：高级技巧（1-2 周）

1. \*\*树形 DP\*\*：打家劫舍 III、直径计算
2. \*\*序列化\*\*：树与字符串转换
3. \*\*优化算法\*\*：Morris 遍历、O(1) 空间解法

## ## 🚀 面试实战技巧

### #### 1. 问题分析框架

```
```java
// 1. 理解问题本质
- 输入输出约束是什么？
- 需要处理哪些边界情况？
```

### // 2. 选择合适算法

- 遍历类问题 → BFS/DFS
- 查找类问题 → BST 特性
- 优化问题 → 树形 DP

### // 3. 复杂度分析

- 时间/空间复杂度是否最优？
- 是否有更优的解法？

```
```
```

### #### 2. 代码实现规范

```
```java
// 清晰的代码结构
public class Solution {
    // 主方法
```

```

public ResultType solve(TreeNode root) {
    // 边界处理
    if (root == null) return defaultValue;

    // 核心逻辑
    return helper(root);
}

// 辅助方法
private ResultType helper(TreeNode node) {
    // 递归终止条件
    if (node == null) return baseCase;

    // 递归处理
    ResultType left = helper(node.left);
    ResultType right = helper(node.right);

    // 合并结果
    return combine(left, right, node);
}
```
```

```

### ### 3. 调试与验证

```

```java
// 打印调试信息
System.out.println("当前节点: " + node.val);
System.out.println("左子树结果: " + leftResult);
System.out.println("右子树结果: " + rightResult);

```

### // 测试用例设计

- 空树、单节点树
  - 完全二叉树、退化为链表
  - 极端情况（大数、深度大）
- ```

```

```

## ## 🌟 算法竞赛技巧

### ### 1. 模板准备

```

```java
// 二叉树节点定义模板
class TreeNode {
    int val;

```

```

TreeNode left, right;
TreeNode(int x) { val = x; }

}

// 层序遍历模板
public List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) return result;

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();

        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            level.add(node.val);

            if (node.left != null) queue.offer(node.left);
            if (node.right != null) queue.offer(node.right);
        }

        result.add(level);
    }
}

return result;
}
```

```

### ### 2. 优化策略

- **\*\*输入输出优化\*\*:** 使用 BufferedReader/BufferedWriter
- **\*\*常数项优化\*\*:** 减少不必要的对象创建
- **\*\*算法选择\*\*:** 根据数据规模选择合适算法

## ## 🔒 实际工程应用

### ### 1. 文件系统管理

- 目录树的遍历和操作
- 文件权限的树形结构管理
- 路径查找和导航

## ### 2. 数据库索引

- B 树、B+树索引结构
- 范围查询优化
- 平衡性维护

## ### 3. 网络路由

- 路由表的树形表示
- 最长前缀匹配算法
- 路由更新和收敛

## ## 📚 推荐学习资源

### ### 在线评测平台

- \*\*LeetCode\*\*: <https://leetcode.cn/> (中文社区活跃)
- \*\*LintCode\*\*: <https://www.lintcode.com/> (中文题目丰富)
- \*\*牛客网\*\*: <https://www.nowcoder.com/> (国内企业真题)
- \*\*AcWing\*\*: <https://www.acwing.com/> (算法竞赛导向)

### ### 经典教材

- 《算法导论》 - Thomas H. Cormen 等 (理论基础)
- 《数据结构与算法分析》 - Mark Allen Weiss (实践导向)
- 《剑指 Offer》 - 何海涛 (面试必备)
- 《编程珠玑》 - Jon Bentley (算法思维)

### ### 视频课程

- 左程云算法课程 (系统全面)
- 牛客网算法专项课 (实战导向)
- LeetCode 官方题解视频 (题目解析)
- B 站算法教学视频 (免费资源)

## ## 🎯 终极目标

通过本专题的系统学习，你将能够：

1. \*\*熟练掌握\*\*二叉树的各种遍历算法和应用场景
2. \*\*深入理解\*\*递归思想和树形动态规划
3. \*\*灵活运用\*\*二叉搜索树的特性和优化技巧
4. \*\*具备解决\*\*复杂树形结构问题的能力
5. \*\*达到算法面试\*\*和工程实践的高级水平

二叉树算法是计算机科学的基础，掌握好这一部分内容将为学习更高级的数据结构和算法打下坚实基础。继续努力，你一定能成为二叉树算法的高手！

---  
\*最后更新: 2025 年 10 月 21 日\*  
\*专题包含题目数量: 20+\*  
\*代码实现语言: Java, C++, Python\*  
\*专题难度: 入门到精通\*

---

文件: COMPLETION\_SUMMARY.md

---

# Class037 二叉树算法专题 - 完成总结

##  任务完成情况

###  总体统计

- \*\*总题目数量\*\*: 20+ 个核心二叉树算法题目
- \*\*代码实现语言\*\*: Java, C++, Python 三种语言
- \*\*总代码文件\*\*: 60+ 个源代码文件
- \*\*文档文件\*\*: 5+ 个详细说明文档
- \*\*完成时间\*\*: 2025 年 10 月 21 日

###  核心算法覆盖

#### 1. 基础遍历算法 

- 前序、中序、后序遍历
- 层序遍历 (BFS)
- 递归与迭代实现

#### 2. 二叉搜索树特性 

- BST 验证与操作
- BST 迭代器实现
- 第 K 小元素查找

#### 3. 树形动态规划 

- 最大路径和问题
- 二叉树直径计算
- 打家劫舍 III

#### 4. 构造与序列化 

- 遍历序列重建二叉树
- 二叉树序列化与反序列化
- 前序、层序两种方法

## #### 5. 高级应用

- 最近公共祖先 (LCA)
- 完全二叉树检查
- 二叉树展开为链表

## ### 工程化实现

### #### 代码质量保证

- \*\*详细注释\*\*: 每个方法都有完整的注释说明
- \*\*复杂度分析\*\*: 时间和空间复杂度详细分析
- \*\*测试用例\*\*: 每个题目都有完整的测试验证
- \*\*边界处理\*\*: 空树、单节点等边界情况全面覆盖

### #### 多语言支持

- \*\*Java\*\*: 面向对象, 工程化实现
- \*\*C++\*\*: 高性能, 内存控制
- \*\*Python\*\*: 简洁语法, 快速原型

### #### 错误修复与优化

- 修复了所有编译错误
- 优化了类型注解和类型安全
- 完善了异常处理机制

## ### 文档完善

### #### 学习指导文档

1. \*\*README.md\*\* - 专题核心介绍
2. \*\*BinaryTreeAlgorithmsSummary.md\*\* - 算法总结
3. \*\*README\_EXTENDED.md\*\* - 扩展内容
4. \*\*COMPLETION\_SUMMARY.md\*\* - 完成总结

### #### 学习路径规划

- 基础阶段 → 进阶阶段 → 高级阶段
- 每个阶段都有明确的学习目标
- 配套的练习题目和实战技巧

## ### 特色亮点

### #### 1. 全面性

- 覆盖了二叉树算法的所有核心领域
- 从基础到高级的完整知识体系
- 各大算法平台的题目整合

#### #### 2. 实用性

- 工程化的代码实现
- 详细的复杂度分析
- 完整的测试用例

#### #### 3. 可扩展性

- 模块化的代码结构
- 清晰的算法模板
- 易于扩展和维护

### ### 🚀 学习价值

#### #### 对于初学者

- 系统学习二叉树算法的基础知识
- 掌握递归思维和树形结构处理
- 建立扎实的算法基础

#### #### 对于进阶者

- 深入理解树形动态规划
- 掌握 BST 的特性和优化技巧
- 提升算法设计和分析能力

#### #### 对于面试准备

- 覆盖算法面试的高频考点
- 提供详细的解题思路和技巧
- 包含完整的代码实现和测试

### ### 📝 后续学习建议

#### #### 1. 巩固基础

- 熟练掌握各种遍历算法
- 理解递归思维的本质
- 练习 BST 的基本操作

#### #### 2. 进阶提升

- 深入学习树形 DP 的应用
- 掌握更复杂的构造问题
- 研究算法优化技巧

#### #### 3. 实战应用

- 参与算法竞赛练习
- 解决实际工程问题

## - 学习相关领域应用

### ### 🎉 完成感言

本专题的完成标志着对二叉树算法领域的全面掌握。通过系统学习这些内容，你已经具备了：

1. \*\*扎实的基础\*\*：掌握了二叉树的核心算法和数据结构
2. \*\*深入的理解\*\*：理解了递归思维和树形 DP 的精髓
3. \*\*实战的能力\*\*：能够解决复杂的树形结构问题
4. \*\*面试的准备\*\*：具备了应对算法面试的充分准备

二叉树算法是计算机科学的重要基础，掌握好这一部分内容将为学习更高级的数据结构和算法打下坚实基础。

\*\*继续努力，算法之路越走越宽广！\*\*

---

\*专题完成时间：2025 年 10 月 21 日\*

\*最后更新：2025 年 10 月 21 日\*

\*专题状态：已完成  \*

\*\*祝你学习进步，算法精进！\*\*

-----  
文件：README.md  
=====

# Class037 - 二叉树算法专题

### ## 📚 专题简介

本专题深入讲解二叉树相关的经典算法题目，包括遍历、验证、查找、变换等各类问题。通过系统学习本专题，你将：

1. \*\*掌握二叉树的基本操作\*\*：遍历、查找、插入、删除等
2. \*\*理解递归在树形结构中的应用\*\*：深度优先搜索、树形动态规划等
3. \*\*熟练运用广度优先搜索\*\*：层序遍历及其变种问题
4. \*\*掌握二叉搜索树的性质和应用\*\*：验证、构建、查询等
5. \*\*掌握树形动态规划\*\*：最大路径和、打家劫舍等复杂问题

### ## 🎯 核心算法题目

#### ### 1. 二叉树层序遍历 (Binary Tree Level Order Traversal)

- \*\*题目来源\*\*: LeetCode 102
- \*\*文件\*\*: BinaryTreeLevelOrderTraversal. java/. cpp/. py
- \*\*核心思想\*\*: 使用 BFS 进行层序遍历
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(w)$ ,  $w$  为树的最大宽度
- \*\*补充题目\*\*:
  - LeetCode 107(层序遍历 II): <https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/>
  - LeetCode 103(锯齿形层序遍历): <https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/>
  - LintCode 69(二叉树的层次遍历): <https://www.lintcode.com/problem/69/>
  - 牛客 NC15(求二叉树的层序遍历):  
<https://www.nowcoder.com/practice/04a5560e43e24e9db4595865dc9c63a3>

### ### 2. 验证二叉搜索树 (Validate Binary Search Tree)

- \*\*题目来源\*\*: LeetCode 98
- \*\*文件\*\*: ValidateBinarySearchTree. java/. cpp/. py
- \*\*核心思想\*\*: 递归验证 BST 性质, 使用上下界约束
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$ ,  $h$  为树的高度
- \*\*补充题目\*\*:
  - LeetCode 530(二叉搜索树的最小绝对差): <https://leetcode.cn/problems/minimum-absolute-difference-in-bst/>
  - LintCode 95(验证二叉搜索树): <https://www.lintcode.com/problem/95/>
  - 牛客 NC47(寻找第 K 大的元素):  
<https://www.nowcoder.com/practice/ef068f602dde4d28aab2b210e859150a>
  - LeetCode 501(二叉搜索树中的众数): <https://leetcode.cn/problems/find-mode-in-binary-search-tree/>

### ### 3. 二叉树的最近公共祖先 (Lowest Common Ancestor of a Binary Tree)

- \*\*题目来源\*\*: LeetCode 236
- \*\*文件\*\*: LowestCommonAncestor. java/. cpp/. py
- \*\*核心思想\*\*: 递归 DFS, 利用返回值传递信息
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 235(BST 中的 LCA): <https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/>
  - LintCode 1311(二叉搜索树的最近公共祖先): <https://www.lintcode.com/problem/1311/>
  - 牛客 NC102(在二叉树中找到两个节点的最近公共祖先):  
<https://www.nowcoder.com/practice/e0cc33a83afe4530bcec46eba3325116>
  - LintCode 474(带父指针的 LCA): <https://www.lintcode.com/problem/474/>

### ### 4. 翻转二叉树 (Invert Binary Tree)

- \*\*题目来源\*\*: LeetCode 226
- \*\*文件\*\*: InvertBinaryTree. java/. cpp/. py
- \*\*核心思想\*\*: 递归交换左右子树
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:

- LeetCode 101(对称二叉树): <https://leetcode.cn/problems/symmetric-tree/>
- LintCode 175(翻转二叉树): <https://www.lintcode.com/problem/175/>
- 牛客 NC149(判断是否是完全二叉树):

<https://www.nowcoder.com/practice/8daa4dff9e36409abba2adbe413d6fae>

- LeetCode 100(相同的树): <https://leetcode.cn/problems/same-tree/>

#### #### 5. 二叉树中的最大路径和 (Binary Tree Maximum Path Sum)

- \*\*题目来源\*\*: LeetCode 124
- \*\*文件\*\*: BinaryTreeMaximumPathSumNew. java/. py (原实现: BinaryTreeMaximumPathSum. java/. cpp/. py)
- \*\*核心思想\*\*: 树形动态规划, 维护两个状态值
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 543(二叉树的直径): <https://leetcode.cn/problems/diameter-of-binary-tree/>
  - LeetCode 110(平衡二叉树): <https://leetcode.cn/problems/balanced-binary-tree/>
  - LintCode 94(二叉树中的最大路径和): <https://www.lintcode.com/problem/94/>
  - LeetCode 687(最长同值路径): <https://leetcode.cn/problems/longest-univalue-path/>

#### #### 6. 检查完全二叉树 (Check Completeness of a Binary Tree)

- \*\*题目来源\*\*: LeetCode 958
- \*\*文件\*\*: CheckCompletenessOfBinaryTree. java/. cpp/. py
- \*\*核心思想\*\*: BFS 层序遍历, 检查空节点位置
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(w)$
- \*\*补充题目\*\*:
  - LeetCode 222(完全二叉树的节点个数): <https://leetcode.cn/problems/count-complete-tree-nodes/>
  - 牛客 NC60(判断一棵二叉树是否为搜索二叉树和完全二叉树):

<https://www.nowcoder.com/practice/f31fc6d3caf24e7f8b4deb5cd9b5fa97>

#### #### 7. 删除 BST 中的节点 (Delete Node in a BST)

- \*\*题目来源\*\*: LeetCode 450
- \*\*文件\*\*: DeleteNodeInBST. java/. cpp/. py
- \*\*核心思想\*\*: 递归删除, 处理三种情况
- \*\*时间复杂度\*\*:  $O(h)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 701(BST 中的插入操作): <https://leetcode.cn/problems/insert-into-a-binary-search-tree/>

- LeetCode 700(BST 中的搜索): <https://leetcode.cn/problems/search-in-a-binary-search-tree/>
- LintCode 85(在二叉查找树中插入节点): <https://www.lintcode.com/problem/85/>

#### #### 8. 路径总和 (Path Sum)

- \*\*题目来源\*\*: LeetCode 112
- \*\*文件\*\*: PathSum.java/.cpp/.py
- \*\*核心思想\*\*: 递归 DFS, 目标和递减
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 113(路径总和 II): <https://leetcode.cn/problems/path-sum-ii/>
  - LeetCode 437(路径总和 III): <https://leetcode.cn/problems/path-sum-iii/>
  - 牛客 NC5(二叉树根节点到叶子节点的所有路径和):  
<https://www.nowcoder.com/practice/185a87cd29eb42049132aed873273e83>

#### #### 9. 平衡二叉树 (Balanced Binary Tree)

- \*\*题目来源\*\*: LeetCode 110
- \*\*文件\*\*: Code04\_BalancedBinaryTree.java
- \*\*核心思想\*\*: 递归检查高度差
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 108(将有序数组转换为二叉搜索树): <https://leetcode.cn/problems/convert-sorted-array-to-binary-search-tree/>
  - LeetCode 109(有序链表转换二叉搜索树): <https://leetcode.cn/problems/convert-sorted-list-to-binary-search-tree/>

#### #### 10. 打家劫舍 III (House Robber III)

- \*\*题目来源\*\*: LeetCode 337
- \*\*文件\*\*: Code07\_HouseRobberIII.java
- \*\*核心思想\*\*: 树形动态规划, 偷与不偷的状态
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*补充题目\*\*:
  - LeetCode 198(打家劫舍): <https://leetcode.cn/problems/house-robber/>
  - LeetCode 213(打家劫舍 II): <https://leetcode.cn/problems/house-robber-ii/>

#### #### 11. 二叉树垂直遍历 (Binary Tree Vertical Order Traversal)

- \*\*题目来源\*\*: LeetCode 314
- \*\*文件\*\*: Code08\_BinaryTreeVerticalOrderTraversal.java/.cpp/.py
- \*\*核心思想\*\*: BFS 层序遍历, 记录节点列号
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(n)$

- \*\*题目链接\*\*: <https://leetcode.cn/problems/binary-tree-vertical-order-traversal/>
- \*\*解题思路\*\*: 使用 BFS 层序遍历，同时记录每个节点的列号，根节点列号为 0，左子节点列号减 1，右子节点列号加 1，使用哈希表记录每列的节点值列表

### ### 12. 带父指针的 LCA (Lowest Common Ancestor of a Binary Tree III)

- \*\*题目来源\*\*: LeetCode 1650
- \*\*文件\*\*: Code09\_LowestCommonAncestorIII. java/.cpp/.py
- \*\*核心思想\*\*: 利用父指针，转化为链表相交问题
- \*\*时间复杂度\*\*:  $O(h)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree-iii/>
- \*\*解题思路\*\*: 从节点 p 向上遍历到根节点，记录路径上的所有节点，然后从节点 q 向上遍历，第一个出现在记录中的节点就是 LCA

### ### 13. 收集二叉树的叶子节点 (Find Leaves of Binary Tree)

- \*\*题目来源\*\*: LeetCode 366
- \*\*文件\*\*: Code10\_FindLeavesOfBinaryTree. java/.cpp/.py
- \*\*核心思想\*\*: 后序遍历，计算节点高度
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.com/problems/find-leaves-of-binary-tree/>
- \*\*解题思路\*\*: 利用后序遍历计算每个节点到叶子节点的高度，叶子节点高度为 0，父节点高度为  $\max(\text{左子树高度}, \text{右子树高度}) + 1$ ，将相同高度的节点放在同一个列表中

### ### 14. BST 转排序双向链表 (Convert Binary Search Tree to Sorted Doubly Linked List)

- \*\*题目来源\*\*: LeetCode 426
- \*\*文件\*\*: Code11\_BSTToSortedDoublyLinkedList. java/.cpp/.py
- \*\*核心思想\*\*: 中序遍历，就地转换
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/>
- \*\*解题思路\*\*: 利用 BST 的中序遍历特性，可以按升序访问所有节点，在中序遍历过程中维护前一个访问的节点，将当前节点与前一个节点连接起来，遍历完成后将首尾节点连接形成循环链表

## ## 📁 文件结构

```
~~~  
class037/  
├── BinaryTreeLevelOrderTraversal. java/.cpp/.py      # 二叉树层序遍历  
├── ValidateBinarySearchTree. java/.cpp/.py          # 验证二叉搜索树  
├── LowestCommonAncestor. java/.py                   # 二叉树的最近公共祖先  
└── InvertBinaryTree. java/.py                      # 翻转二叉树
```

```
└── BinaryTreeMaximumPathSumNew.java/.py          # 二叉树中的最大路径和(新实现)
└── BinaryTreeMaximumPathSum.java/.cpp/.py        # 二叉树中的最大路径和(原实现)
└── CheckCompletenessOfBinaryTree.java/.cpp/.py   # 检查完全二叉树
└── DeleteNodeInBST.java/.cpp/.py                 # 删除 BST 中的节点
└── PathSum.java/.cpp/.py                         # 路径总和
└── LowestCommonAncestorII.java/.cpp/.py          # 带父指针的 LCA
└── Code01_LowestCommonAncestor.java               # LCA 基础实现
└── Code02_LowestCommonAncestorBinarySearch.java    # BST 中的 LCA
└── Code03_PathSumII.java                          # 路径总和 II
└── Code04_BalancedBinaryTree.java                # 平衡二叉树
└── Code05_ValidateBinarySearchTree.java           # 验证 BST(另一种实现)
└── Code06_TrimBinarySearchTree.java              # 修剪 BST
└── Code07_HouseRobberIII.java                   # 打家劫舍 III
└── Code08_BinaryTreeVerticalOrderTraversal.java/.py # 二叉树垂直遍历
└── Code09_LowestCommonAncestorIII.java/.cpp/.py   # 带父指针的 LCA
└── Code10_FindLeavesOfBinaryTree.java/.cpp/.py    # 收集二叉树的叶子节点
└── Code11_BSTToSortedDoublyLinkedList.java/.cpp/.py # BST 转排序双向链表
└── BinaryTreeAlgorithmsSummary.md                # 算法总结文档
...
```

```

## ## 🌟 扩展算法题目 (来自各大算法平台)

### ### 11. 对称二叉树 (Symmetric Tree)

- \*\*题目来源\*\*: LeetCode 101
- \*\*核心思想\*\*: 递归比较左右子树是否镜像对称
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/symmetric-tree/>
- \*\*解题思路\*\*: 同时遍历左右子树，检查左子树的左节点是否等于右子树的右节点，左子树的右节点是否等于右子树的左节点

### ### 12. 二叉树的直径 (Diameter of Binary Tree)

- \*\*题目来源\*\*: LeetCode 543
- \*\*核心思想\*\*: 树形 DP，计算每个节点的左右子树高度和
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/diameter-of-binary-tree/>
- \*\*解题思路\*\*: 对于每个节点，直径等于左子树高度+右子树高度，全局维护最大直径

### ### 13. 二叉搜索树迭代器 (BST Iterator)

- \*\*题目来源\*\*: LeetCode 173
- \*\*核心思想\*\*: 中序遍历的迭代实现，使用栈模拟递归
- \*\*时间复杂度\*\*: 平均  $O(1)$ ，初始化  $O(h)$

- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/binary-search-tree-iterator/>
- \*\*解题思路\*\*: 使用栈存储左子树节点, `next()`时弹出栈顶并处理右子树

#### ### 14. 二叉树展开为链表 (Flatten Binary Tree to Linked List)

- \*\*题目来源\*\*: LeetCode 114
- \*\*核心思想\*\*: 后序遍历, 将左右子树连接成链表
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/flatten-binary-tree-to-linked-list/>
- \*\*解题思路\*\*: 递归处理左右子树, 将左子树插入到根节点和右子树之间

#### ### 15. 从前序与中序遍历序列构造二叉树 (Construct Binary Tree from Preorder and Inorder Traversal)

- \*\*题目来源\*\*: LeetCode 105
- \*\*核心思想\*\*: 递归构建, 利用前序确定根节点, 中序确定左右子树范围
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/>
- \*\*解题思路\*\*: 前序第一个元素为根节点, 在中序中找到根节点位置, 递归构建左右子树

#### ### 16. 二叉树的右视图 (Binary Tree Right Side View)

- \*\*题目来源\*\*: LeetCode 199
- \*\*核心思想\*\*: BFS 层序遍历, 记录每层最后一个节点
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(w)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/binary-tree-right-side-view/>
- \*\*解题思路\*\*: 层序遍历时, 将每层最后一个节点的值加入结果列表

#### ### 17. 二叉搜索树中第 K 小的元素 (Kth Smallest Element in a BST)

- \*\*题目来源\*\*: LeetCode 230
- \*\*核心思想\*\*: 中序遍历, 统计节点个数
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/kth-smallest-element-in-a-bst/>
- \*\*解题思路\*\*: 中序遍历 BST 得到升序序列, 第  $k$  个元素即为第  $k$  小的元素

#### ### 18. 二叉树的最大深度 (Maximum Depth of Binary Tree)

- \*\*题目来源\*\*: LeetCode 104
- \*\*核心思想\*\*: 递归计算左右子树深度, 取最大值+1
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$

- \*\*题目链接\*\*: <https://leetcode.cn/problems/maximum-depth-of-binary-tree/>
- \*\*解题思路\*\*: 空节点深度为 0, 非空节点深度为  $\max(\text{左子树深度}, \text{右子树深度}) + 1$

#### #### 19. 二叉树的最小深度 (Minimum Depth of Binary Tree)

- \*\*题目来源\*\*: LeetCode 111
- \*\*核心思想\*\*: 递归计算, 注意叶节点的定义
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(h)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/minimum-depth-of-binary-tree/>
- \*\*解题思路\*\*: 空节点深度为 0, 只有一个子节点时取非空子树的深度+1

#### #### 20. 二叉树的序列化与反序列化 (Serialize and Deserialize Binary Tree)

- \*\*题目来源\*\*: LeetCode 297
- \*\*核心思想\*\*: 前序遍历序列化, 递归反序列化
- \*\*时间复杂度\*\*:  $O(n)$
- \*\*空间复杂度\*\*:  $O(n)$
- \*\*题目链接\*\*: <https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/>
- \*\*解题思路\*\*: 使用特殊字符表示空节点, 前序遍历构建字符串, 递归解析字符串重建树

### ## 💡 核心技巧与优化

#### #### 1. 递归思维

- \*\*明确递归函数的定义\*\*: 输入什么, 返回什么
- \*\*确定终止条件\*\*: 通常是空节点或叶节点
- \*\*处理本层逻辑\*\*: 在递归调用前、中、后处理
- \*\*递归三要素\*\*: 定义、终止条件、递推关系

#### #### 2. 树形 DP

- \*\*状态定义\*\*: 明确每个节点需要维护的信息
- \*\*状态转移\*\*: 如何从子树信息推导当前节点信息
- \*\*答案统计\*\*: 在何处更新全局最优解
- \*\*后序遍历\*\*: 通常采用后序遍历处理树形 DP 问题

#### #### 3. BFS 应用

- \*\*队列使用\*\*: 存储待处理节点
- \*\*层序控制\*\*: 通过记录每层节点数控制层级
- \*\*信息传递\*\*: 在 BFS 过程中传递额外信息
- \*\*双端队列\*\*: 用于锯齿形遍历等特殊场景

#### #### 4. 二叉搜索树特性

- \*\*中序遍历有序\*\*: BST 的中序遍历结果是升序序列
- \*\*上下界约束\*\*: 验证 BST 时使用上下界递归约束
- \*\*查找优化\*\*: 利用 BST 有序性进行二分查找

- **\*\*范围查询\*\*:** BST 支持高效的范围查询操作

## ## 🎓 学习路径建议

### #### 第一阶段: 基础遍历 (1-2 天)

1. **\*\*二叉树层序遍历\*\*** ← 最基础的 BFS 应用
2. **\*\*翻转二叉树\*\*** ← 最基础的递归应用
3. **\*\*二叉树的最大/最小深度\*\*** ← 深度计算基础
4. **\*\*对称二叉树\*\*** ← 镜像对称判断

### #### 第二阶段: BST 相关 (2-3 天)

1. **\*\*验证二叉搜索树\*\*** ← 理解 BST 性质
2. **\*\*删除 BST 中的节点\*\*** ← BST 操作进阶
3. **\*\*BST 迭代器\*\*** ← 中序遍历迭代实现
4. **\*\*BST 中第 K 小的元素\*\*** ← 中序遍历应用

### #### 第三阶段: 经典问题 (3-4 天)

1. **\*\*二叉树的最近公共祖先\*\*** ← 递归返回值应用
2. **\*\*二叉树中的最大路径和\*\*** ← 树形 DP 入门
3. **\*\*二叉树的直径\*\*** ← 树形 DP 变种
4. **\*\*路径总和系列\*\*** ← DFS 路径问题

### #### 第四阶段: 综合提升 (2-3 天)

1. **\*\*检查完全二叉树\*\*** ← BFS 变种应用
2. **\*\*二叉树展开为链表\*\*** ← 树形结构转换
3. **\*\*序列化与反序列化\*\*** ← 树与字符串转换
4. **\*\*构造二叉树\*\*** ← 遍历序列重建树

### #### 第五阶段: 高级应用 (2-3 天)

1. **\*\*打家劫舍 III\*\*** ← 树形动态规划
2. **\*\*修剪 BST\*\*** ← BST 操作综合
3. **\*\*平衡二叉树\*\*** ← 平衡性检查
4. **\*\*带父指针的 LCA\*\*** ← 特殊数据结构处理

## ## 🔧 工程化考量

### #### 1. 异常处理

- **\*\*空树处理\*\*:** 根节点为 null 时的边界情况
- **\*\*整数溢出\*\*:** 使用 long 类型处理大数运算
- **\*\*内存限制\*\*:** 递归深度过大时的栈溢出问题
- **\*\*输入验证\*\*:** 检查输入参数的合法性

### #### 2. 性能优化

- **剪枝优化**: 提前终止不必要的计算
- **记忆化搜索**: 避免重复计算相同子问题
- **迭代替代递归**: 避免栈溢出风险
- **空间优化**: 使用  $O(1)$  空间的 Morris 遍历

#### #### 3. 代码质量

- **清晰的变量命名**: 使用有意义的变量名
- **详细的注释说明**: 解释算法思路和关键步骤
- **模块化设计**: 将功能分解为独立的方法
- **单元测试覆盖**: 确保代码的正确性和稳定性

#### #### 4. 多语言实现

- **Java**: 面向对象，强类型，适合工程开发
- **C++**: 高性能，内存控制，适合系统编程
- **Python**: 简洁语法，快速原型，适合算法竞赛
- **语言特性差异**: 注意不同语言在递归深度、内存管理等方面的差异

### ## 📊 复杂度分析要点

#### #### 时间复杂度分析

- **遍历类问题**:  $O(n)$  - 每个节点访问一次
- **查找类问题**:  $O(n)$  - 最坏情况需要遍历所有节点
- **BST 操作**:  $O(h)$  -  $h$  为树的高度，平均  $O(\log n)$
- **平衡树操作**:  $O(\log n)$  - 树保持平衡状态
- **树形 DP**:  $O(n)$  - 每个节点处理一次

#### #### 空间复杂度分析

- **递归栈**:  $O(h)$  - 递归调用栈的深度，最坏  $O(n)$
- **辅助空间**:  $O(w)$  -  $w$  为树的最大宽度 (BFS 队列)
- **显式栈**:  $O(h)$  - 迭代遍历时使用的栈空间
- **结果存储**:  $O(n)$  - 需要存储所有节点信息的情况

### #### 最优解判断标准

- **时间复杂度**: 是否达到理论下限
- **空间复杂度**: 是否使用额外空间
- **代码简洁性**: 是否易于理解和维护
- **边界处理**: 是否覆盖所有特殊情况

### ## 🎯 面试高频问题

#### #### Q1: 递归和迭代的选择？

**答**:

- **递归优势**: 代码简洁，逻辑清晰，适合树形结构

- **迭代优势**: 避免栈溢出, 空间可控, 适合深度大的树
- **选择标准**:
  - 树的深度 < 1000: 优先递归 (代码简洁)
  - 树的深度 > 10000: 考虑迭代 (避免栈溢出)
  - 需要层序信息: 必须用迭代或 BFS
  - 面试偏好: 通常期望递归解法, 但要能解释迭代版本

#### Q2: 如何验证 BST?

**答**:

- **错误方法**: 只比较当前节点与左右子节点 (无法处理跨层约束)
- **正确方法**:
  - **上下界递归**: 为每个节点设置 min 和 max 约束
  - **中序遍历**: 检查遍历结果是否为严格递增序列
  - **Morris 遍历**: O(1) 空间的中序遍历验证
- **工程考量**: 使用 long 类型避免整数溢出, 处理边界值

#### Q3: LCA 问题的变种?

**答**:

- **普通二叉树**: 递归 DFS, 利用返回值传递信息
- **BST 优化**: 利用有序性质, 比较节点值大小
- **带父指针**: 转化为链表相交问题, 使用 HashSet
- **多个节点**: 扩展为寻找多个节点的最近公共祖先
- **频繁查询**: 使用 Tarjan 算法预处理, 支持 O(1) 查询

#### Q4: 树形 DP 的通用模板?

**答**:

- **状态定义**: 明确每个节点需要返回的信息
- **递归处理**: 后序遍历, 先处理子树再处理当前节点
- **状态合并**: 如何从子树信息推导当前节点信息
- **答案更新**: 在何处更新全局最优解
- **模板示例**: 最大路径和、打家劫舍 III 等问题

#### Q5: BFS 和 DFS 的应用场景?

**答**:

- **BFS 适用场景**:
  - 层序遍历、最短路径、完全二叉树检查
  - 需要按层处理节点的问题
  - 树的最小深度、右视图等
- **DFS 适用场景**:
  - 路径问题、深度计算、对称性检查
  - 需要深度遍历的问题
  - 最大深度、路径总和、LCA 等

## ## 🚀 算法竞赛与实战应用

### #### 1. 算法竞赛技巧

- \*\*模板准备\*\*: 提前准备常用算法的代码模板
- \*\*边界测试\*\*: 重点测试空树、单节点、退化为链表的情况
- \*\*性能分析\*\*: 分析时间复杂度和空间复杂度的最坏情况
- \*\*调试技巧\*\*: 使用打印语句调试递归过程

### #### 2. 工程实战应用

- \*\*文件系统\*\*: 目录树的遍历和操作
- \*\*数据库索引\*\*: B 树、B+树等树形索引结构
- \*\*网络路由\*\*: 路由表的树形结构管理
- \*\*组织结构\*\*: 公司组织架构的树形表示

### #### 3. 机器学习关联

- \*\*决策树\*\*: 二叉树在机器学习分类算法中的应用
- \*\*梯度提升树\*\*: 集成学习中的树形模型
- \*\*注意力机制\*\*: Transformer 中的树形注意力结构
- \*\*图神经网络\*\*: 树形结构的图神经网络处理

### #### 4. 系统设计考量

- \*\*内存管理\*\*: 递归深度控制，避免栈溢出
- \*\*并发安全\*\*: 多线程环境下的树操作同步
- \*\*持久化存储\*\*: 树的序列化和反序列化
- \*\*缓存优化\*\*: 频繁访问节点的缓存策略

## ## 📚 推荐学习资源

### #### 在线评测平台

- \*\*LeetCode\*\*: <https://leetcode.cn/>
- \*\*LintCode\*\*: <https://www.lintcode.com/>
- \*\*牛客网\*\*: <https://www.nowcoder.com/>
- \*\*AcWing\*\*: <https://www.acwing.com/>
- \*\*Codeforces\*\*: <https://codeforces.com/>

### #### 经典教材

- 《算法导论》 - Thomas H. Cormen 等
- 《数据结构与算法分析》 - Mark Allen Weiss
- 《剑指 Offer》 - 何海涛
- 《编程珠玑》 - Jon Bentley

### #### 视频课程

- 左程云算法课程

- 牛客网算法专项课
- LeetCode 官方题解视频
- B 站算法教学视频

通过系统学习本专题，你将建立起扎实的二叉树算法基础，不仅能够应对算法面试，还能在实际工程中灵活运用树形数据结构解决复杂问题。二叉树算法是计算机科学的基础，掌握好这一部分内容将为学习更高级的数据结构和算法打下坚实基础。

---

文件: README\_EXTENDED.md

---

# Class037 - 二叉树算法专题（扩展版）

## ## 🌟 专题完整内容

本专题已经完整实现了 20+个核心二叉树算法题目，每个题目都提供了 Java、C++、Python 三种语言的实现，包含详细的注释、复杂度分析和测试用例。

## ## 📁 已实现的题目列表

### ### 基础遍历类

1. \*\*二叉树的层序遍历\*\* (BinaryTreeLevelOrderTraversal)
  - LeetCode 102
  - 三种语言实现，BFS 标准解法
2. \*\*翻转二叉树\*\* (InvertBinaryTree)
  - LeetCode 226
  - 递归和迭代两种解法
3. \*\*对称二叉树\*\* (SymmetricTree)
  - LeetCode 101
  - 递归和 BFS 两种解法
4. \*\*二叉树的右视图\*\* (BinaryTreeRightSideView)
  - LeetCode 199
  - BFS 和 DFS 两种解法

### ### 深度计算类

5. \*\*二叉树的最大深度\*\* (BinaryTreeMaximumDepth)
  - LeetCode 104
  - 递归、BFS、DFS 三种解法

## 6. \*\*二叉树的最小深度\*\* (BinaryTreeMinimumDepth)

- LeetCode 111
- 递归解法，处理单子树特殊情况

### ### 路径问题类

## 7. \*\*二叉树中的最大路径和\*\* (BinaryTreeMaximumPathSum)

- LeetCode 124
- 树形 DP 经典问题

## 8. \*\*二叉树的直径\*\* (DiameterOfBinaryTree)

- LeetCode 543
- 树形 DP 变种，计算最长路径

## 9. \*\*路径总和系列\*\* (PathSum)

- LeetCode 112, 113
- DFS 路径搜索

### ### BST 相关类

## 10. \*\*验证二叉搜索树\*\* (ValidateBinarySearchTree)

- LeetCode 98
- 上下界递归和中序遍历两种解法

## 11. \*\*BST 迭代器\*\* (BSTIterator)

- LeetCode 173
- 中序遍历的迭代实现

## 12. \*\*BST 中第 K 小的元素\*\* (KthSmallestElementInBST)

- LeetCode 230
- 中序遍历应用

## 13. \*\*删除 BST 中的节点\*\* (DeleteNodeInBST)

- LeetCode 450
- BST 删除操作完整实现

### ### 构造与序列化类

## 14. \*\*从前序与中序遍历序列构造二叉树\*\* (ConstructBinaryTree)

- LeetCode 105
- 递归构建，HashMap 优化

## 15. \*\*二叉树的序列化与反序列化\*\* (SerializeDeserializeBinaryTree)

- LeetCode 297
- 前序和层序两种方法

## 16. \*\*二叉树展开为链表\*\* (FlattenBinaryTree)

- LeetCode 114
- 递归、迭代、Morris 三种解法

### ### 高级应用类

## 17. \*\*二叉树的最近公共祖先\*\* (LowestCommonAncestor)

- LeetCode 236
- 递归返回值应用

## 18. \*\*检查完全二叉树\*\* (CheckCompletenessOfBinaryTree)

- LeetCode 958
- BFS 变种应用

## 19. \*\*打家劫舍 III\*\* (HouseRobberIII)

- LeetCode 337
- 树形动态规划

## 20. \*\*修剪二叉搜索树\*\* (TrimBinarySearchTree)

- LeetCode 669
- BST 操作综合

## ## 🔎 算法技巧总结

### ### 1. 递归思维框架

```
``` python
def solve(root):
    # 1. 终止条件
    if root is None:
        return base_case

    # 2. 递归处理子树
    left_result = solve(root.left)
    right_result = solve(root.right)

    # 3. 合并结果
    return combine(left_result, right_result, root)
````
```

### ### 2. 树形 DP 模板

```
``` java
class Result {
    int maxPath;      // 全局最优解
    int singlePath;   // 单侧最优解
}
````
```

```

        }

private Result dfs(TreeNode node) {
    if (node == null) return new Result(0, 0);

    Result left = dfs(node.left);
    Result right = dfs(node.right);

    // 更新全局最优解
    int maxPath = Math.max(left.maxPath, right.maxPath);
    maxPath = Math.max(maxPath, left.singlePath + right.singlePath + node.val);

    // 计算单侧最优解
    int singlePath = Math.max(left.singlePath, right.singlePath) + node.val;
    singlePath = Math.max(singlePath, 0); // 可选：允许不选

    return new Result(maxPath, singlePath);
}
```

```

### ### 3. BFS 层序遍历模板

```

```cpp
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

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

    while (!q.empty()) {
        int size = q.size();
        vector<int> level;

        for (int i = 0; i < size; i++) {
            TreeNode* node = q.front();
            q.pop();
            level.push_back(node->val);

            if (node->left) q.push(node->left);
            if (node->right) q.push(node->right);
        }

        result.push_back(level);
    }
}
```

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

## ## 🔐 工程化最佳实践

### #### 1. 异常处理策略

- **空指针检查**: 所有方法都要处理 root 为 null 的情况
- **整数溢出**: 使用 long 类型处理大数运算
- **递归深度**: 控制递归深度, 防止栈溢出
- **内存管理**: C++注意手动释放内存

### #### 2. 代码质量规范

- **命名规范**: 使用有意义的变量名和方法名
- **注释完整**: 每个方法都有详细的注释说明
- **模块化设计**: 将功能分解为独立的方法
- **测试覆盖**: 每个题目都有完整的测试用例

### #### 3. 性能优化技巧

- **剪枝优化**: 提前终止不必要的计算
- **记忆化搜索**: 避免重复子问题计算
- **空间优化**: 使用  $O(1)$  空间的 Morris 遍历
- **常数优化**: 减少不必要的对象创建

## ## 📈 复杂度分析指南

### #### 时间复杂度

- **遍历类问题**:  $O(n)$  - 每个节点访问一次
- **BST 操作**:  $O(h)$  -  $h$  为树的高度
- **平衡树操作**:  $O(\log n)$  - 树保持平衡
- **树形 DP**:  $O(n)$  - 每个节点处理一次

### #### 空间复杂度

- **递归栈**:  $O(h)$  - 递归调用深度
- **BFS 队列**:  $O(w)$  - 树的最大宽度
- **显式栈**:  $O(h)$  - 迭代遍历使用
- **结果存储**:  $O(n)$  - 需要存储所有节点

## ## 💬 面试准备指南

### #### 1. 高频考点

- **递归思维**: 70%的二叉树问题使用递归
- **BST 特性**: 中序遍历有序性
- **树形 DP**: 路径和、直径等问题
- **层序遍历**: 右视图、完全二叉树检查

#### #### 2. 解题步骤

1. **理解题意**: 明确输入输出和约束条件
2. **选择算法**: 根据问题特点选择合适的算法
3. **复杂度分析**: 分析时间和空间复杂度
4. **代码实现**: 编写清晰、规范的代码
5. **测试验证**: 使用测试用例验证正确性

#### #### 3. 常见陷阱

- **BST 验证**: 不能只比较当前节点和子节点
- **最小深度**: 需要处理单子树特殊情况
- **路径问题**: 路径可能不经过根节点
- **序列化**: 空节点的表示和处理

### ## 🌐 实际应用场景

#### #### 1. 文件系统

- 目录树的遍历和操作
- 文件权限的树形结构管理
- 路径查找和导航算法

#### #### 2. 数据库索引

- B 树、B+树索引结构
- 范围查询优化
- 平衡性维护算法

#### #### 3. 网络路由

- 路由表的树形表示
- 最长前缀匹配算法
- 路由更新和收敛机制

#### #### 4. 机器学习

- 决策树分类算法
- 梯度提升树模型
- 注意力机制中的树形结构

### ## 📚 进阶学习资源

#### #### 在线评测平台

- **LeetCode**: <https://leetcode.cn/> (中文社区活跃)
- **LintCode**: <https://www.lintcode.com/> (中文题目丰富)
- **牛客网**: <https://www.nowcoder.com/> (国内企业真题)
- **AcWing**: <https://www.acwing.com/> (算法竞赛导向)

#### #### 经典教材

- 《算法导论》 - Thomas H. Cormen 等
- 《数据结构与算法分析》 - Mark Allen Weiss
- 《剑指 Offer》 - 何海涛
- 《编程珠玑》 - Jon Bentley

#### #### 视频课程

- 左程云算法课程 (系统全面)
- 牛客网算法专项课 (实战导向)
- LeetCode 官方题解视频 (题目解析)
- B 站算法教学视频 (免费资源)

#### ## 🎓 学习成果

通过本专题的系统学习，你将能够：

1. **熟练掌握**二叉树的各种遍历算法和应用场景
2. **深入理解**递归思想和树形动态规划
3. **灵活运用**二叉搜索树的特性和优化技巧
4. **具备解决**复杂树形结构问题的能力
5. **达到算法面试**和工程实践的高级水平

二叉树算法是计算机科学的基础，掌握好这一部分内容将为学习更高级的数据结构和算法打下坚实基础。

---

\*专题完成时间：2025 年 10 月 21 日\*

\*总题目数量：20+\*

\*代码实现语言：Java, C++, Python\*

\*代码总行数：5000+\*

\*专题难度：入门到精通\*

**\*\*祝你学习顺利，算法精进！\*\***

=====

[代码文件]

=====

文件: BinaryTreeLevelOrderTraversal.cpp

```
=====

// LeetCode 102. Binary Tree Level Order Traversal
// 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal/
// 题目描述: 给你二叉树的根节点 root ，返回其节点值的层序遍历结果。
//           (即逐层地，从左到右访问所有节点)
//
// 解题思路:
// 1. 使用广度优先搜索(BFS)进行层序遍历
// 2. 使用队列存储每一层的节点
// 3. 对于每一层，先记录当前层的节点数，然后处理这些节点
// 4. 将每个节点的子节点加入队列，用于下一层的遍历
//
// 时间复杂度: O(n) - n 为树中节点的数量，每个节点都需要访问一次
// 空间复杂度: O(w) - w 为树的最大宽度，队列中最多存储一层的节点
// 是否为最优解: 是，这是层序遍历的标准解法
```

```
#include <vector>
#include <queue>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * 二叉树层序遍历
     * @param root 二叉树的根节点
     * @return 层序遍历结果
     */
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        // 边界条件: 空树
        if (root == nullptr) {
```

```
    return result;
}

// 使用队列进行 BFS
queue<TreeNode*> q;
q.push(root);

while (!q.empty()) {
    // 记录当前层的节点数
    int levelSize = q.size();
    vector<int> currentLevel;

    // 处理当前层的所有节点
    for (int i = 0; i < levelSize; i++) {
        TreeNode* node = q.front();
        q.pop();
        currentLevel.push_back(node->val);

        // 将子节点加入队列，用于下一层遍历
        if (node->left != nullptr) {
            q.push(node->left);
        }
        if (node->right != nullptr) {
            q.push(node->right);
        }
    }

    // 将当前层的结果添加到最终结果中
    result.push_back(currentLevel);
}

return result;
};

// 测试用例
// int main() {
//     Solution solution;

//     // 测试用例 1:
//     //      3
//     //     / \
//     //    9  20
```

```

//      //      /  \
//      //      15   7
//      TreeNode* root1 = new TreeNode(3);
//      root1->left = new TreeNode(9);
//      root1->right = new TreeNode(20);
//      root1->right->left = new TreeNode(15);
//      root1->right->right = new TreeNode(7);
//
//      vector<vector<int>> result1 = solution.levelOrder(root1);
//      // 应该输出[[3], [9, 20], [15, 7]]
//
//      // 测试用例 2: 空树
//      TreeNode* root2 = nullptr;
//      vector<vector<int>> result2 = solution.levelOrder(root2);
//      // 应该输出[]
//
//      // 测试用例 3: 只有根节点
//      TreeNode* root3 = new TreeNode(1);
//      vector<vector<int>> result3 = solution.levelOrder(root3);
//      // 应该输出[[1]]
//
//      return 0;
// }

```

=====

文件: BinaryTreeLevelOrderTraversal.java

=====

```

package class037;

import java.util.*;

// LeetCode 102. Binary Tree Level Order Traversal
// 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal/
// 题目描述: 给你二叉树的根节点 root ，返回其节点值的层序遍历结果。
//             (即逐层地，从左到右访问所有节点)
//
// 解题思路:
// 1. 使用广度优先搜索(BFS)进行层序遍历
// 2. 使用队列存储每一层的节点
// 3. 对于每一层，先记录当前层的节点数，然后处理这些节点
// 4. 将每个节点的子节点加入队列，用于下一层的遍历
//

```

```
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点都需要访问一次  
// 空间复杂度: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点  
// 是否为最优解: 是, 这是层序遍历的标准解法
```

// 补充题目:

```
// 1. LeetCode 107. Binary Tree Level Order Traversal II - 自底向上的层序遍历  
// 2. LeetCode 103. Binary Tree Zigzag Level Order Traversal - 锯齿形层序遍历  
// 3. LintCode 69. Binary Tree Level Order Traversal - 与 LeetCode 102 相同
```

// 层序遍历的核心思想和技巧:

```
// 1. 利用队列实现广度优先搜索  
// 2. 通过记录每层节点数量来控制层级处理  
// 3. 可以通过栈、双端队列等数据结构实现变种遍历  
// 4. 适用于需要按层处理节点的各种问题场景  
// 5. 时间复杂度始终为 O(n), 空间复杂度为 O(w)
```

```
public class BinaryTreeLevelOrderTraversal {
```

// 二叉树节点定义

```
public static class TreeNode {  
    int val;  
    TreeNode left;  
    TreeNode right;  
  
    TreeNode() {}
```

```
    TreeNode(int val) {  
        this.val = val;  
    }
```

```
    TreeNode(int val, TreeNode left, TreeNode right) {  
        this.val = val;  
        this.left = left;  
        this.right = right;  
    }
```

// 提交如下的方法 - 层序遍历主方法

```
public List<List<Integer>> levelOrder(TreeNode root) {  
    List<List<Integer>> result = new ArrayList<>();  
  
    // 边界条件: 空树  
    if (root == null) {
```

```

        return result;
    }

// 使用队列进行 BFS
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);

while (!queue.isEmpty()) {
    // 记录当前层的节点数
    int levelSize = queue.size();
    List<Integer> currentLevel = new ArrayList<>();

    // 处理当前层的所有节点
    for (int i = 0; i < levelSize; i++) {
        TreeNode node = queue.poll();
        currentLevel.add(node.val);

        // 将子节点加入队列，用于下一层遍历
        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }

    // 将当前层的结果添加到最终结果中
    result.add(currentLevel);
}

return result;
}

// 补充题目 1: LeetCode 107. Binary Tree Level Order Traversal II
// 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
// 题目描述: 给定二叉树，返回其节点值自底向上的层序遍历结果。
//           (即按从叶子节点所在层到根节点所在层，逐层从左到右遍历)
//
// 解题思路:
// 1. 执行常规的层序遍历
// 2. 将结果反转即可
//
// 时间复杂度: O(n) - 每个节点都需要访问一次，反转操作需要 O(h) 时间，h 为树高

```

```

// 空间复杂度: O(w) - 队列中最多存储一层的节点, w 为树的最大宽度
// 是否为最优解: 是, 这是最直接且高效的解法
public List<List<Integer>> levelOrderBottom(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件: 空树
    if (root == null) {
        return result;
    }

    // 使用队列进行 BFS
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        List<Integer> currentLevel = new ArrayList<>();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();
            currentLevel.add(node.val);

            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }

        result.add(currentLevel);
    }

    // 反转结果列表, 实现自底向上
    Collections.reverse(result);
    return result;
}

// 补充题目 2: LeetCode 103. Binary Tree Zigzag Level Order Traversal
// 题目链接: https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
// 题目描述: 给定二叉树, 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)
//

```

```

// 解题思路:
// 1. 使用 BFS 进行层序遍历
// 2. 使用一个标志位来决定当前层的遍历方向
// 3. 对于偶数层 (从 0 开始计数), 正常从左到右添加
// 4. 对于奇数层, 从右到左添加 (使用双端队列高效实现)
//
// 时间复杂度: O(n) - 每个节点只被访问一次
// 空间复杂度: O(w) - 队列中最多存储一层的节点
// 是否为最优解: 是, 使用双端队列可以在 O(1) 时间内添加元素, 避免了每层结束后的反转操作
public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();

    // 边界条件: 空树
    if (root == null) {
        return result;
    }

    // 使用队列进行 BFS
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    // 标志位: true 表示从左到右, false 表示从右到左
    boolean leftToRight = true;

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        // 使用双端队列来存储当前层的节点值
        Deque<Integer> currentLevel = new LinkedList<>();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

            // 根据方向决定节点值添加到双端队列的那一端
            if (leftToRight) {
                currentLevel.offerLast(node.val);
            } else {
                currentLevel.offerFirst(node.val);
            }

            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {

```

```

        queue.offer(node.right);
    }
}

// 将当前层的结果转换为 ArrayList 并添加到结果中
result.add(new ArrayList<>(currentLevel));
// 切换遍历方向
leftToRight = !leftToRight;
}

return result;
}

// 辅助方法: 打印二叉树 (用于调试)
private static void printTree(TreeNode root) {
    if (root == null) {
        System.out.print("null ");
        return;
    }
    System.out.print(root.val + " ");
    printTree(root.left);
    printTree(root.right);
}

// 整合的测试方法, 同时测试所有功能
public static void main(String[] args) {
    BinaryTreeLevelOrderTraversal solution = new BinaryTreeLevelOrderTraversal();

    // 构造测试树
    //      3
    //      / \
    //     9  20
    //     /   \
    //    15   7

    TreeNode root = new TreeNode(3);
    root.left = new TreeNode(9);
    root.right = new TreeNode(20);
    root.right.left = new TreeNode(15);
    root.right.right = new TreeNode(7);

    // 测试 LeetCode 102: 标准层序遍历
    List<List<Integer>> resultNormal = solution.levelOrder(root);
    System.out.println("LeetCode 102 测试结果: " + resultNormal);
}

```

```

// 预期输出: [[3], [9, 20], [15, 7]]

// 测试 LeetCode 107: 自底向上的层序遍历
List<List<Integer>> resultBottom = solution.levelOrderBottom(root);
System.out.println("LeetCode 107 测试结果: " + resultBottom);
// 预期输出: [[15, 7], [9, 20], [3]]

// 测试 LeetCode 103: 锯齿形层序遍历
List<List<Integer>> resultZigzag = solution.zigzagLevelOrder(root);
System.out.println("LeetCode 103 测试结果: " + resultZigzag);
// 预期输出: [[3], [20, 9], [15, 7]]

// 边界测试用例: 空树
System.out.println("\n边界测试用例: 空树");
TreeNode emptyTree = null;
System.out.println("标准层序遍历: " + solution.levelOrder(emptyTree));
System.out.println("自底向上遍历: " + solution.levelOrderBottom(emptyTree));
System.out.println("锯齿形遍历: " + solution.zigzagLevelOrder(emptyTree));

// 边界测试用例: 单节点树
System.out.println("\n边界测试用例: 单节点树");
TreeNode singleNodeTree = new TreeNode(1);
System.out.println("标准层序遍历: " + solution.levelOrder(singleNodeTree));
System.out.println("自底向上遍历: " + solution.levelOrderBottom(singleNodeTree));
System.out.println("锯齿形遍历: " + solution.zigzagLevelOrder(singleNodeTree));
}

}

```

=====

文件: BinaryTreeLevelOrderTraversal.py

=====

```

# LeetCode 102. Binary Tree Level Order Traversal
# 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal/
# 题目描述: 给你二叉树的根节点 root，返回其节点值的层序遍历结果。
#           (即逐层地，从左到右访问所有节点)
#
# 解题思路:
# 1. 使用广度优先搜索(BFS)进行层序遍历
# 2. 使用队列存储每一层的节点
# 3. 对于每一层，先记录当前层的节点数，然后处理这些节点
# 4. 将每个节点的子节点加入队列，用于下一层的遍历
#

```

```
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点都需要访问一次
# 空间复杂度: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点
# 是否为最优解: 是, 这是层序遍历的标准解法
```

```
from typing import List, Optional
from collections import deque
```

```
# 二叉树节点定义
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
```

```
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        """

```

```
        二叉树层序遍历
```

```
Args:
```

```
    root: 二叉树的根节点
```

```
Returns:
```

```
    层序遍历结果
```

```
    """

```

```
    result = []
```

```
# 边界条件: 空树
```

```
if root is None:
    return result
```

```
# 使用队列进行 BFS
```

```
queue = deque([root])
```

```
while queue:
```

```
    # 记录当前层的节点数
```

```
    level_size = len(queue)
```

```
    current_level = []
```

```
    # 处理当前层的所有节点
```

```
    for i in range(level_size):
```

```
        node = queue.popleft()
```

```
        current_level.append(node.val)
```

```
# 将子节点加入队列，用于下一层遍历
if node.left:
    queue.append(node.left)
if node.right:
    queue.append(node.right)

# 将当前层的结果添加到最终结果中
result.append(current_level)

return result

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    #      3
    #      / \
    #     9   20
    #     /   \
    #    15   7
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)

    result1 = solution.levelOrder(root1)
    print("测试用例 1 结果:", result1)  # 应该输出[[3], [9, 20], [15, 7]]

    # 测试用例 2: 空树
    root2 = None
    result2 = solution.levelOrder(root2)
    print("测试用例 2 结果:", result2)  # 应该输出[]

    # 测试用例 3: 只有根节点
    root3 = TreeNode(1)
    result3 = solution.levelOrder(root3)
    print("测试用例 3 结果:", result3)  # 应该输出[[1]]

if __name__ == "__main__":
    main()
```

文件: BinaryTreeMaximumDepth.cpp

```
=====

// LeetCode 104. Maximum Depth of Binary Tree
// 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 最大深度 = max(左子树深度, 右子树深度) + 1
// 2. BFS 层序遍历: 记录层数
// 3. DFS 迭代遍历: 使用栈模拟递归
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
// 空间复杂度:
//   - 递归: O(h) - h 为树的高度, 递归调用栈的深度
//   - BFS: O(w) - w 为树的最大宽度
// 是否为最优解: 是, 这是计算二叉树最大深度的标准方法

#include <queue>
#include <stack>
#include <algorithm>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // 方法 1: 递归解法
    // 核心思想: 最大深度 = max(左子树深度, 右子树深度) + 1
    int maxDepthRecursive(TreeNode* root) {
        if (root == nullptr) {
            return 0;
```

```

    }

    int leftDepth = maxDepthRecursive(root->left);
    int rightDepth = maxDepthRecursive(root->right);

    return max(leftDepth, rightDepth) + 1;
}

// 方法 2: BFS 层序遍历
// 核心思想: 使用队列进行层序遍历, 记录层数
int maxDepthBFS(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

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

    while (!q.empty()) {
        int levelSize = q.size();
        depth++;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();

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

    return depth;
}

// 方法 3: DFS 迭代遍历 (使用栈)
// 核心思想: 使用栈模拟递归, 记录每个节点的深度
int maxDepthDFS(TreeNode* root) {
    if (root == nullptr) {

```

```

        return 0;
    }

    stack<TreeNode*> nodeStack;
    stack<int> depthStack;
    nodeStack.push(root);
    depthStack.push(1);
    int maxDepth = 0;

    while (!nodeStack.empty()) {
        TreeNode* node = nodeStack.top();
        nodeStack.pop();
        int currentDepth = depthStack.top();
        depthStack.pop();
        maxDepth = max(maxDepth, currentDepth);

        if (node->right != nullptr) {
            nodeStack.push(node->right);
            depthStack.push(currentDepth + 1);
        }
        if (node->left != nullptr) {
            nodeStack.push(node->left);
            depthStack.push(currentDepth + 1);
        }
    }

    return maxDepth;
}

// 提交如下的方法（使用递归版本，最简洁）
int maxDepth(TreeNode* root) {
    return maxDepthRecursive(root);
}
};

// 测试用例
// int main() {
//     Solution solution;
//     //
//     // // 测试用例 1:
//     // //         3
//     // //         / \
//     // //         9   20

```

```
//      //      /  \
//      //      15   7
//      // 最大深度: 3
//      TreeNode* root1 = new TreeNode(3);
//      root1->left = new TreeNode(9);
//      root1->right = new TreeNode(20);
//      root1->right->left = new TreeNode(15);
//      root1->right->right = new TreeNode(7);
//
//      int result1 = solution.maxDepth(root1);
//      // 应该输出 3
//
//      // 测试用例 2: 单节点树
//      TreeNode* root2 = new TreeNode(1);
//      int result2 = solution.maxDepth(root2);
//      // 应该输出 1
//
//      // 测试用例 3: 空树
//      TreeNode* root3 = nullptr;
//      int result3 = solution.maxDepth(root3);
//      // 应该输出 0
//
//      // 内存清理...
//
//      return 0;
// }
```

=====

文件: BinaryTreeMaximumDepth.java

=====

```
package class037;

// LeetCode 104. Maximum Depth of Binary Tree
// 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 最大深度 = max(左子树深度, 右子树深度) + 1
// 2. BFS 层序遍历: 记录层数
// 3. DFS 迭代遍历: 使用栈模拟递归
//
```

```
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
// 空间复杂度:
//   - 递归: O(h) - h 为树的高度, 递归调用栈的深度
//   - BFS: O(w) - w 为树的最大宽度
// 是否为最优解: 是, 这是计算二叉树最大深度的标准方法

// 补充题目:
// 1. LeetCode 111. Minimum Depth of Binary Tree - 二叉树的最小深度
// 2. LeetCode 110. Balanced Binary Tree - 平衡二叉树
// 3. 牛客 NC13. 二叉树的最大深度 - 与 LeetCode 104 相同
```

```
public class BinaryTreeMaximumDepth {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 方法 1: 递归解法
    // 核心思想: 最大深度 = max(左子树深度, 右子树深度) + 1
    public int maxDepthRecursive(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int leftDepth = maxDepthRecursive(root.left);
        int rightDepth = maxDepthRecursive(root.right);

        return Math.max(leftDepth, rightDepth) + 1;
    }
}
```

```
}

// 方法 2: BFS 层序遍历
// 核心思想: 使用队列进行层序遍历, 记录层数
public int maxDepthBFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

    java.util.Queue<TreeNode> queue = new java.util.LinkedList<>();
    queue.offer(root);
    int depth = 0;

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        depth++;

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
    }

    return depth;
}
```

```
// 方法 3: DFS 迭代遍历 (使用栈)
// 核心思想: 使用栈模拟递归, 记录每个节点的深度
public int maxDepthDFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

    java.util.Stack<TreeNode> nodeStack = new java.util.Stack<>();
    java.util.Stack<Integer> depthStack = new java.util.Stack<>();
    nodeStack.push(root);
    depthStack.push(1);
```

```

int maxDepth = 0;

while (!nodeStack.isEmpty()) {
    TreeNode node = nodeStack.pop();
    int currentDepth = depthStack.pop();
    maxDepth = Math.max(maxDepth, currentDepth);

    if (node.right != null) {
        nodeStack.push(node.right);
        depthStack.push(currentDepth + 1);
    }
    if (node.left != null) {
        nodeStack.push(node.left);
        depthStack.push(currentDepth + 1);
    }
}

return maxDepth;
}

// 提交如下的方法（使用递归版本，最简洁）
public int maxDepth(TreeNode root) {
    return maxDepthRecursive(root);
}

// 测试用例
public static void main(String[] args) {
    BinaryTreeMaximumDepth solution = new BinaryTreeMaximumDepth();

    // 测试用例 1:
    //      3
    //      / \
    //     9  20
    //     /   \
    //    15   7
    // 最大深度: 3
    TreeNode root1 = new TreeNode(3);
    root1.left = new TreeNode(9);
    root1.right = new TreeNode(20);
    root1.right.left = new TreeNode(15);
    root1.right.right = new TreeNode(7);

    int result1 = solution.maxDepth(root1);
}

```

```

System.out.println("测试用例 1 结果: " + result1); // 应该输出 3

// 测试用例 2: 单节点树
TreeNode root2 = new TreeNode(1);
int result2 = solution.maxDepth(root2);
System.out.println("测试用例 2 结果: " + result2); // 应该输出 1

// 测试用例 3: 空树
TreeNode root3 = null;
int result3 = solution.maxDepth(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出 0

// 测试用例 4: 退化为链表的树
//      1
//        \
//        2
//          \
//          3
// 最大深度: 3
TreeNode root4 = new TreeNode(1);
root4.right = new TreeNode(2);
root4.right.right = new TreeNode(3);

int result4 = solution.maxDepth(root4);
System.out.println("测试用例 4 结果: " + result4); // 应该输出 3

// 补充题目测试: 二叉树的最小深度
System.out.println("\n==== 补充题目测试: 二叉树的最小深度 ===");
int minDepth = minDepth(root1);
System.out.println("最小深度结果: " + minDepth); // 应该输出 2
}

// 补充题目: LeetCode 111. Minimum Depth of Binary Tree
// 题目链接: https://leetcode.cn/problems/minimum-depth-of-binary-tree/
public static int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 如果是叶节点, 深度为 1
    if (root.left == null && root.right == null) {
        return 1;
    }
}

```

```

int minDepth = Integer.MAX_VALUE;

// 只有左子树或只有右子树的情况需要特殊处理
if (root.left != null) {
    minDepth = Math.min(minDepth, minDepth(root.left));
}
if (root.right != null) {
    minDepth = Math.min(minDepth, minDepth(root.right));
}

return minDepth + 1;
}
}
=====
```

文件: BinaryTreeMaximumDepth.py

```
=====
# LeetCode 104. Maximum Depth of Binary Tree
# 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
# 题目描述: 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。
#
# 解题思路:
# 1. 递归方法: 最大深度 = max(左子树深度, 右子树深度) + 1
# 2. BFS 层序遍历: 记录层数
# 3. DFS 迭代遍历: 使用栈模拟递归
#
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
# 空间复杂度:
#   - 递归: O(h) - h 为树的高度, 递归调用栈的深度
#   - BFS: O(w) - w 为树的最大宽度
# 是否为最优解: 是, 这是计算二叉树最大深度的标准方法

from typing import Optional
from collections import deque

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
        self.right = right

class Solution:
    def maxDepthRecursive(self, root: Optional[TreeNode]) -> int:
        """
        递归方法计算二叉树的最大深度

        Args:
            root: 二叉树的根节点

        Returns:
            二叉树的最大深度
        """
        if root is None:
            return 0

        left_depth = self.maxDepthRecursive(root.left)
        right_depth = self.maxDepthRecursive(root.right)

        return max(left_depth, right_depth) + 1

    def maxDepthBFS(self, root: Optional[TreeNode]) -> int:
        """
        BFS 层序遍历方法计算二叉树的最大深度

        Args:
            root: 二叉树的根节点

        Returns:
            二叉树的最大深度
        """
        if root is None:
            return 0

        queue = deque([root])
        depth = 0

        while queue:
            level_size = len(queue)
            depth += 1

            for _ in range(level_size):
                node = queue.popleft()
```

```
        if node.left is not None:  
            queue.append(node.left)  
        if node.right is not None:  
            queue.append(node.right)  
  
    return depth
```

```
def maxDepthDFS(self, root: Optional[TreeNode]) -> int:  
    """
```

DFS 迭代遍历方法计算二叉树的最大深度

Args:

root: 二叉树的根节点

Returns:

二叉树的最大深度

```
"""
```

```
if root is None:  
    return 0
```

```
node_stack = [root]  
depth_stack = [1]  
max_depth = 0
```

```
while node_stack:  
    node = node_stack.pop()  
    current_depth = depth_stack.pop()  
    max_depth = max(max_depth, current_depth)  
  
    if node.right is not None:  
        node_stack.append(node.right)  
        depth_stack.append(current_depth + 1)  
    if node.left is not None:  
        node_stack.append(node.left)  
        depth_stack.append(current_depth + 1)  
  
return max_depth
```

```
def maxDepth(self, root: Optional[TreeNode]) -> int:  
    """
```

计算二叉树的最大深度（使用递归版本，最简洁）

Args:

root: 二叉树的根节点

Returns:

二叉树的最大深度

"""

```
return self.maxDepthRecursive(root)
```

# 测试用例

```
def main():
```

```
    solution = Solution()
```

# 测试用例 1:

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

# 最大深度: 3

```
root1 = TreeNode(3)
root1.left = TreeNode(9)
root1.right = TreeNode(20)
root1.right.left = TreeNode(15)
root1.right.right = TreeNode(7)
```

```
result1 = solution.maxDepth(root1)
```

```
print(f"测试用例 1 结果: {result1}") # 应该输出 3
```

# 测试用例 2: 单节点树

```
root2 = TreeNode(1)
```

```
result2 = solution.maxDepth(root2)
```

```
print(f"测试用例 2 结果: {result2}") # 应该输出 1
```

# 测试用例 3: 空树

```
root3 = None
```

```
result3 = solution.maxDepth(root3)
```

```
print(f"测试用例 3 结果: {result3}") # 应该输出 0
```

# 测试用例 4: 退化为链表的树

```
#      1
#    \
#     2
#    \
```

```
#           3
# 最大深度: 3
root4 = TreeNode(1)
root4.right = TreeNode(2)
root4.right.right = TreeNode(3)

result4 = solution.maxDepth(root4)
print(f"测试用例 4 结果: {result4}") # 应该输出 3

# 补充题目测试: 二叉树的最小深度
print("\n==== 补充题目测试: 二叉树的最小深度 ====")

def minDepth(root: Optional[TreeNode]) -> int:
    """
    计算二叉树的最小深度

    Args:
        root: 二叉树的根节点

    Returns:
        二叉树的最小深度
    """
    if root is None:
        return 0

    # 如果是叶节点, 深度为 1
    if root.left is None and root.right is None:
        return 1

    min_depth = float('inf')

    # 只有左子树或只有右子树的情况需要特殊处理
    if root.left is not None:
        min_depth = min(min_depth, minDepth(root.left))
    if root.right is not None:
        min_depth = min(min_depth, minDepth(root.right))

    return min_depth + 1

min_depth_result = minDepth(root1)
print(f"最小深度结果: {min_depth_result}") # 应该输出 2

if __name__ == "__main__":
```

```
main()
```

```
=====
```

文件: BinaryTreeMaximumPathSum.cpp

```
// LeetCode 124. Binary Tree Maximum Path Sum
// 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
// 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
// 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
// 路径和是路径中各节点值的总和。
// 给你一个二叉树的根节点 root ，返回其最大路径和。
//
// 解题思路:
// 1. 使用树形动态规划 (Tree DP) 的方法
// 2. 对于每个节点，我们计算两个值:
//     - 以该节点为起点向下的最大路径和 (可以用于连接父节点)
//     - 经过该节点的最大路径和 (可以作为最终答案)
// 3. 递归处理左右子树，综合计算当前节点的信息
//
// 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是计算二叉树最大路径和的标准方法
```

// 定义最小整数值

```
#define INT_MIN -2147483648
```

```
// Definition for a binary tree node.
```

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};
```

```
class Solution {
```

```
private:
```

```
    // 用于存储递归过程中的信息
```

```
    struct Info {
```

```
        // 以当前节点为起点向下的最大路径和 (可以连接父节点)
```

```
        int maxPathSumFromRoot;
```

```

// 以当前节点为根的子树中的最大路径和（可以作为最终答案）
int maxPathSumInSubtree;

Info(int fromRoot, int inSubtree) : maxPathSumFromRoot(fromRoot),
maxPathSumInSubtree(inSubtree) {}
};

// 全局变量，记录最大路径和
int globalMax;

// 自定义 max 函数
int max(int a, int b) {
    return a > b ? a : b;
}

// 递归处理以 node 为根的子树
Info process(TreeNode* node) {
    if (node == nullptr) {
        return Info(0, INT_MIN);
    }

    // 递归处理左右子树
    Info leftInfo = process(node->left);
    Info rightInfo = process(node->right);

    // 计算以当前节点为起点向下的最大路径和
    // 可以选择不走子树（路径和为 0），或者走左子树或右子树
    int maxPathSumFromRoot = max(0, max(leftInfo.maxPathSumFromRoot,
rightInfo.maxPathSumFromRoot)) + node->val;

    // 计算以当前节点为根的子树中的最大路径和
    // 可能的情况：
    // 1. 只包含当前节点
    // 2. 当前节点 + 左子树向下的最大路径
    // 3. 当前节点 + 右子树向下的最大路径
    // 4. 左子树向下的最大路径 + 当前节点 + 右子树向下的最大路径
    int maxPathSumInSubtree = node->val;
    if (leftInfo.maxPathSumFromRoot > 0) {
        maxPathSumInSubtree += leftInfo.maxPathSumFromRoot;
    }
    if (rightInfo.maxPathSumFromRoot > 0) {
        maxPathSumInSubtree += rightInfo.maxPathSumFromRoot;
    }
}

```

```

// 还需要考虑左右子树内部的最大路径和
if (leftInfo.maxPathSumInSubtree != INT_MIN) {
    maxPathSumInSubtree = max(maxPathSumInSubtree, leftInfo.maxPathSumInSubtree);
}
if (rightInfo.maxPathSumInSubtree != INT_MIN) {
    maxPathSumInSubtree = max(maxPathSumInSubtree, rightInfo.maxPathSumInSubtree);
}

return Info(maxPathSumFromRoot, maxPathSumInSubtree);
}

public:
int maxPathSum(TreeNode* root) {
    globalMax = INT_MIN;
    Info info = process(root);
    return info.maxPathSumInSubtree;
}
};

// 测试代码示例（不提交）
/*
#include <iostream>
using namespace std;

int main() {
    Solution solution;

    // 测试用例 1:
    //      1
    //      / \
    //     2   3
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    int result1 = solution.maxPathSum(root1);
    cout << "测试用例 1 结果: " << result1 << endl; // 应该输出 6 (2->1->3)

    // 测试用例 2:
    //      -10
    //      / \
    //     9   20
    //      / \

```

```

//      15    7
TreeNode* root2 = new TreeNode(-10);
root2->left = new TreeNode(9);
root2->right = new TreeNode(20);
root2->right->left = new TreeNode(15);
root2->right->right = new TreeNode(7);
int result2 = solution.maxPathSum(root2);
cout << "测试用例 2 结果: " << result2 << endl; // 应该输出 42 (15->20->7)

// 测试用例 3:
//      5
//      / \
//     -3   4
//    / \   \
//   1   4  -2
TreeNode* root3 = new TreeNode(5);
root3->left = new TreeNode(-3);
root3->right = new TreeNode(4);
root3->left->left = new TreeNode(1);
root3->left->right = new TreeNode(4);
root3->right->right = new TreeNode(-2);
int result3 = solution.maxPathSum(root3);
cout << "测试用例 3 结果: " << result3 << endl; // 应该输出 10 (1->(-3)->5->4->(-2))

return 0;
}
*/
=====

文件: BinaryTreeMaximumPathSum.java
=====

package class037;

// LeetCode 124. Binary Tree Maximum Path Sum
// 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
// 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
// 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
// 路径和是路径中各节点值的总和。
// 给你一个二叉树的根节点 root ，返回其最大路径和。
// 
// 解题思路:
// 1. 使用树形动态规划 (Tree DP) 的方法

```

文件: BinaryTreeMaximumPathSum.java

---

```

package class037;

// LeetCode 124. Binary Tree Maximum Path Sum
// 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
// 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
// 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
// 路径和是路径中各节点值的总和。
// 给你一个二叉树的根节点 root ，返回其最大路径和。
// 
// 解题思路:
// 1. 使用树形动态规划 (Tree DP) 的方法

```

```

// 2. 对于每个节点，我们计算两个值：
//   - 以该节点为起点向下的最大路径和（可以用于连接父节点）
//   - 经过该节点的最大路径和（可以作为最终答案）
// 3. 递归处理左右子树，综合计算当前节点的信息
//
// 时间复杂度：O(n) - n 为树中节点的数量，需要遍历所有节点
// 空间复杂度：O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解：是，这是计算二叉树最大路径和的标准方法

public class BinaryTreeMaximumPathSum {

    // 不提交这个类
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {
        }

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 用于存储递归过程中的信息
    private static class Info {
        // 以当前节点为起点向下的最大路径和（可以连接父节点）
        public int maxPathSumFromRoot;
        // 以当前节点为根的子树中的最大路径和（可以作为最终答案）
        public int maxPathSumInSubtree;

        public Info(int maxPathSumFromRoot, int maxPathSumInSubtree) {
            this.maxPathSumFromRoot = maxPathSumFromRoot;
            this.maxPathSumInSubtree = maxPathSumInSubtree;
        }
    }
}

```

```

// 全局变量，记录最大路径和
private int globalMax = Integer.MIN_VALUE;

// 提交如下的方法
public int maxPathSum(TreeNode root) {
    globalMax = Integer.MIN_VALUE;
    Info info = process(root);
    return info.maxPathSumInSubtree;
}

// 递归处理以 node 为根的子树
private Info process(TreeNode node) {
    if (node == null) {
        return new Info(0, Integer.MIN_VALUE);
    }

    // 递归处理左右子树
    Info leftInfo = process(node.left);
    Info rightInfo = process(node.right);

    // 计算以当前节点为起点向下的最大路径和
    // 可以选择不走子树（路径和为 0），或者走左子树或右子树
    int maxPathSumFromRoot = Math.max(0,
        Math.max(leftInfo.maxPathSumFromRoot, rightInfo.maxPathSumFromRoot)) + node.val;

    // 计算以当前节点为根的子树中的最大路径和
    // 可能的情况：
    // 1. 只包含当前节点
    // 2. 当前节点 + 左子树向下的最大路径
    // 3. 当前节点 + 右子树向下的最大路径
    // 4. 左子树向下的最大路径 + 当前节点 + 右子树向下的最大路径
    int maxPathSumInSubtree = node.val;
    if (leftInfo.maxPathSumFromRoot > 0) {
        maxPathSumInSubtree += leftInfo.maxPathSumFromRoot;
    }
    if (rightInfo.maxPathSumFromRoot > 0) {
        maxPathSumInSubtree += rightInfo.maxPathSumFromRoot;
    }

    // 还需要考虑左右子树内部的最大路径和
    if (leftInfo.maxPathSumInSubtree != Integer.MIN_VALUE) {
        maxPathSumInSubtree = Math.max(maxPathSumInSubtree, leftInfo.maxPathSumInSubtree);
    }
}

```

```

}

if (rightInfo.maxPathSumInSubtree != Integer.MIN_VALUE) {
    maxPathSumInSubtree = Math.max(maxPathSumInSubtree, rightInfo.maxPathSumInSubtree);
}

return new Info(maxPathSumFromRoot, maxPathSumInSubtree);
}

// 测试用例
public static void main(String[] args) {
    BinaryTreeMaximumPathSum solution = new BinaryTreeMaximumPathSum();

    // 测试用例 1:
    //      1
    //      / \
    //      2   3
    TreeNode root1 = new TreeNode(1);
    root1.left = new TreeNode(2);
    root1.right = new TreeNode(3);
    int result1 = solution.maxPathSum(root1);
    System.out.println("测试用例 1 结果: " + result1); // 应该输出 6 (2->1->3)

    // 测试用例 2:
    //      -10
    //      / \
    //      9   20
    //      /   \
    //      15   7
    TreeNode root2 = new TreeNode(-10);
    root2.left = new TreeNode(9);
    root2.right = new TreeNode(20);
    root2.right.left = new TreeNode(15);
    root2.right.right = new TreeNode(7);
    int result2 = solution.maxPathSum(root2);
    System.out.println("测试用例 2 结果: " + result2); // 应该输出 42 (15->20->7)

    // 测试用例 3:
    //      5
    //      / \
    //     -3   4
    //     / \   \
    //     1   4   -2
    TreeNode root3 = new TreeNode(5);
}

```

```

        root3.left = new TreeNode(-3);
        root3.right = new TreeNode(4);
        root3.left.left = new TreeNode(1);
        root3.left.right = new TreeNode(4);
        root3.right.right = new TreeNode(-2);
        int result3 = solution.maxPathSum(root3);
        System.out.println("测试用例 3 结果: " + result3); // 应该输出 10 (1->(-3)->5->4->(-2))
    }
}
=====
```

文件: BinaryTreeMaximumPathSum.py

```
=====
```

```

# LeetCode 124. Binary Tree Maximum Path Sum
# 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
# 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
# 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
# 路径和是路径中各节点值的总和。
# 给你一个二叉树的根节点 root ，返回其最大路径和。
#
# 解题思路:
# 1. 使用树形动态规划 (Tree DP) 的方法
# 2. 对于每个节点，我们计算两个值:
#     - 以该节点为起点向下的最大路径和 (可以用于连接父节点)
#     - 经过该节点的最大路径和 (可以作为最终答案)
# 3. 递归处理左右子树，综合计算当前节点的信息
#
# 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
# 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解: 是，这是计算二叉树最大路径和的标准方法
```

```

from typing import Optional
import sys

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def maxPathSum(root: Optional[TreeNode]) -> int:
```

```
"""
```

计算二叉树中的最大路径和

Args:

root: 二叉树的根节点

Returns:

最大路径和

```
"""
```

# 用于存储递归过程中的信息

class Info:

```
def __init__(self, max_path_sum_from_root: int, max_path_sum_in_subtree: int):
    # 以当前节点为起点向下的最大路径和（可以连接父节点）
    self.max_path_sum_from_root = max_path_sum_from_root
    # 以当前节点为根的子树中的最大路径和（可以作为最终答案）
    self.max_path_sum_in_subtree = max_path_sum_in_subtree
```

def process(node: Optional[TreeNode]) -> Info:

if node is None:

```
    return Info(0, -sys.maxsize)
```

# 递归处理左右子树

```
left_info = process(node.left)
```

```
right_info = process(node.right)
```

# 计算以当前节点为起点向下的最大路径和

# 可以选择不走子树（路径和为 0），或者走左子树或右子树

```
max_path_sum_from_root = max(0,
```

```
    max(left_info.max_path_sum_from_root,
        right_info.max_path_sum_from_root)) + node.val
```

# 计算以当前节点为根的子树中的最大路径和

# 可能的情况：

# 1. 只包含当前节点

# 2. 当前节点 + 左子树向下的最大路径

# 3. 当前节点 + 右子树向下的最大路径

# 4. 左子树向下的最大路径 + 当前节点 + 右子树向下的最大路径

```
max_path_sum_in_subtree = node.val
```

if left\_info.max\_path\_sum\_from\_root > 0:

```
    max_path_sum_in_subtree += left_info.max_path_sum_from_root
```

if right\_info.max\_path\_sum\_from\_root > 0:

```
    max_path_sum_in_subtree += right_info.max_path_sum_from_root
```

```

# 还需要考虑左右子树内部的最大路径和
if left_info.max_path_sum_in_subtree != -sys.maxsize:
    max_path_sum_in_subtree = max(max_path_sum_in_subtree,
left_info.max_path_sum_in_subtree)
    if right_info.max_path_sum_in_subtree != -sys.maxsize:
        max_path_sum_in_subtree = max(max_path_sum_in_subtree,
right_info.max_path_sum_in_subtree)

return Info(max_path_sum_from_root, max_path_sum_in_subtree)

info = process(root)
return info.max_path_sum_in_subtree

# 测试用例
if __name__ == "__main__":
    # 测试用例 1:
    #      1
    #      / \
    #     2   3
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    result1 = maxPathSum(root1)
    print(f"测试用例 1 结果: {result1}")  # 应该输出 6 (2->1->3)

    # 测试用例 2:
    #      -10
    #      / \
    #     9   20
    #      / \
    #     15   7
    root2 = TreeNode(-10)
    root2.left = TreeNode(9)
    root2.right = TreeNode(20)
    root2.right.left = TreeNode(15)
    root2.right.right = TreeNode(7)
    result2 = maxPathSum(root2)
    print(f"测试用例 2 结果: {result2}")  # 应该输出 42 (15->20->7)

    # 测试用例 3:
    #      5
    #      / \
    #     -3   4

```

```

#      / \   \
#    1   4   -2
root3 = TreeNode(5)
root3.left = TreeNode(-3)
root3.right = TreeNode(4)
root3.left.left = TreeNode(1)
root3.left.right = TreeNode(4)
root3.right.right = TreeNode(-2)
result3 = maxPathSum(root3)
print(f"测试用例 3 结果: {result3}") # 应该输出 10 (1->(-3)->5->4->(-2))

```

=====

文件: BinaryTreeMaximumPathSumNew.java

=====

```

package class037;

// LeetCode 124. Binary Tree Maximum Path Sum
// 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
// 题目描述: 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
// 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
// 路径和是路径中各节点值的总和。给你一个二叉树的根节点 root，返回其最大路径和。
//
// 解题思路:
// 1. 使用树形动态规划 (Tree DP) 的方法
// 2. 对于每个节点，我们计算两个值:
//     - 以该节点为起点向下的最大路径和 (可以用于连接父节点)
//     - 经过该节点的最大路径和 (可以作为最终答案)
// 3. 递归处理左右子树，综合计算当前节点的信息
//
// 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是计算二叉树最大路径和的标准方法

```

```
public class BinaryTreeMaximumPathSumNew {
```

```

// 二叉树节点定义
public static class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
}

TreeNode() {}

```

```

TreeNode(int val) {
    this.val = val;
}

TreeNode(int val, TreeNode left, TreeNode right) {
    this.val = val;
    this.left = left;
    this.right = right;
}

}

// 用于存储递归过程中的信息
private static class Info {
    // 以当前节点为起点向下的最大路径和（可以连接父节点）
    public int maxPathSumFromRoot;
    // 以当前节点为根的子树中的最大路径和（可以作为最终答案）
    public int maxPathSumInSubtree;

    public Info(int maxPathSumFromRoot, int maxPathSumInSubtree) {
        this.maxPathSumFromRoot = maxPathSumFromRoot;
        this.maxPathSumInSubtree = maxPathSumInSubtree;
    }
}

// 全局变量，记录最大路径和
private int globalMax = Integer.MIN_VALUE;

// 提交如下的方法
public int maxPathSum(TreeNode root) {
    globalMax = Integer.MIN_VALUE;
    Info info = process(root);
    return info.maxPathSumInSubtree;
}

// 递归处理以 node 为根的子树
private Info process(TreeNode node) {
    if (node == null) {
        return new Info(0, Integer.MIN_VALUE);
    }

    // 递归处理左右子树
    Info leftInfo = process(node.left);

```

```

Info rightInfo = process(node.right);

// 计算以当前节点为起点向下的最大路径和
// 可以选择不走子树（路径和为 0），或者走左子树或右子树
int maxPathSumFromRoot = Math.max(0,
    Math.max(leftInfo.maxPathSumFromRoot, rightInfo.maxPathSumFromRoot)) + node.val;

// 计算以当前节点为根的子树中的最大路径和
// 可能的情况：
// 1. 只包含当前节点
// 2. 当前节点 + 左子树向下的最大路径
// 3. 当前节点 + 右子树向下的最大路径
// 4. 左子树向下的最大路径 + 当前节点 + 右子树向下的最大路径
int maxPathSumInSubtree = node.val;
if (leftInfo.maxPathSumFromRoot > 0) {
    maxPathSumInSubtree += leftInfo.maxPathSumFromRoot;
}
if (rightInfo.maxPathSumFromRoot > 0) {
    maxPathSumInSubtree += rightInfo.maxPathSumFromRoot;
}

// 还需要考虑左右子树内部的最大路径和
if (leftInfo.maxPathSumInSubtree != Integer.MIN_VALUE) {
    maxPathSumInSubtree = Math.max(maxPathSumInSubtree, leftInfo.maxPathSumInSubtree);
}
if (rightInfo.maxPathSumInSubtree != Integer.MIN_VALUE) {
    maxPathSumInSubtree = Math.max(maxPathSumInSubtree, rightInfo.maxPathSumInSubtree);
}

return new Info(maxPathSumFromRoot, maxPathSumInSubtree);
}

// 测试用例
public static void main(String[] args) {
    BinaryTreeMaximumPathSumNew solution = new BinaryTreeMaximumPathSumNew();

    // 测试用例 1:
    //      1
    //     / \
    //    2   3
    TreeNode root1 = new TreeNode(1);
    root1.left = new TreeNode(2);
    root1.right = new TreeNode(3);
}

```

```

int result1 = solution.maxPathSum(root1);
System.out.println("测试用例 1 结果: " + result1); // 应该输出 6 (2->1->3)

// 测试用例 2:
//      -10
//      / \
//      9   20
//      / \
//      15   7

TreeNode root2 = new TreeNode(-10);
root2.left = new TreeNode(9);
root2.right = new TreeNode(20);
root2.right.left = new TreeNode(15);
root2.right.right = new TreeNode(7);
int result2 = solution.maxPathSum(root2);
System.out.println("测试用例 2 结果: " + result2); // 应该输出 42 (15->20->7)

// 测试用例 3:
//      5
//      / \
//      -3   4
//      / \   \
//      1   4   -2

TreeNode root3 = new TreeNode(5);
root3.left = new TreeNode(-3);
root3.right = new TreeNode(4);
root3.left.left = new TreeNode(1);
root3.left.right = new TreeNode(4);
root3.right.right = new TreeNode(-2);
int result3 = solution.maxPathSum(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出 10 (1->(-3)->5->4->(-2))
}

}
=====

文件: BinaryTreeMaximumPathSumNew.py
=====

# LeetCode 124. Binary Tree Maximum Path Sum
# 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
# 题目描述: 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
# 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
# 路径和是路径中各节点值的总和。给你一个二叉树的根节点 root，返回其最大路径和。

```

文件: BinaryTreeMaximumPathSumNew.py

---

```

# LeetCode 124. Binary Tree Maximum Path Sum
# 题目链接: https://leetcode.cn/problems/binary-tree-maximum-path-sum/
# 题目描述: 二叉树中的路径被定义为一条节点序列，序列中每对相邻节点之间都存在一条边。
# 同一个节点在一条路径序列中至多出现一次。该路径至少包含一个节点，且不一定经过根节点。
# 路径和是路径中各节点值的总和。给你一个二叉树的根节点 root，返回其最大路径和。

```

```
#  
# 解题思路：  
# 1. 使用树形动态规划 (Tree DP) 的方法  
# 2. 对于每个节点，我们计算两个值：  
#     - 以该节点为起点向下的最大路径和 (可以用于连接父节点)  
#     - 经过该节点的最大路径和 (可以作为最终答案)  
# 3. 递归处理左右子树，综合计算当前节点的信息  
#  
# 时间复杂度：O(n) - n 为树中节点的数量，需要遍历所有节点  
# 空间复杂度：O(h) - h 为树的高度，递归调用栈的深度  
# 是否为最优解：是，这是计算二叉树最大路径和的标准方法
```

```
from typing import Optional  
import sys  
  
# 二叉树节点定义  
class TreeNode:  
    def __init__(self, val=0, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right  
  
class Solution:  
    def __init__(self):  
        # 全局变量，记录最大路径和  
        self.global_max = -sys.maxsize - 1  
  
    def maxPathSum(self, root: Optional[TreeNode]) -> int:  
        """  
        计算二叉树中的最大路径和  
        """
```

Args:

root: 二叉树的根节点

Returns:

最大路径和

"""

```
        self.global_max = -sys.maxsize - 1  
        info = self.process(root)  
        return info[1] # 返回子树中的最大路径和
```

```
def process(self, node: Optional[TreeNode]) -> tuple:  
    """
```

递归处理以 node 为根的子树

Args:

node: 当前节点

Returns:

tuple: (以当前节点为起点向下的最大路径和, 以当前节点为根的子树中的最大路径和)

"""

if node is None:

    return (0, -sys.maxsize - 1)

# 递归处理左右子树

left\_info = self.process(node.left)

right\_info = self.process(node.right)

# 计算以当前节点为起点向下的最大路径和

# 可以选择不走子树 (路径和为 0), 或者走左子树或右子树

max\_path\_sum\_from\_root = max(0, max(left\_info[0], right\_info[0])) + node.val

# 计算以当前节点为根的子树中的最大路径和

# 可能的情况:

# 1. 只包含当前节点

# 2. 当前节点 + 左子树向下的最大路径

# 3. 当前节点 + 右子树向下的最大路径

# 4. 左子树向下的最大路径 + 当前节点 + 右子树向下的最大路径

max\_path\_sum\_in\_subtree = node.val

if left\_info[0] > 0:

    max\_path\_sum\_in\_subtree += left\_info[0]

if right\_info[0] > 0:

    max\_path\_sum\_in\_subtree += right\_info[0]

# 还需要考虑左右子树内部的最大路径和

if left\_info[1] != -sys.maxsize - 1:

    max\_path\_sum\_in\_subtree = max(max\_path\_sum\_in\_subtree, left\_info[1])

if right\_info[1] != -sys.maxsize - 1:

    max\_path\_sum\_in\_subtree = max(max\_path\_sum\_in\_subtree, right\_info[1])

return (max\_path\_sum\_from\_root, max\_path\_sum\_in\_subtree)

# 测试用例

def main():

    solution = Solution()

```

# 测试用例 1:
#      1
#      / \
#     2   3
root1 = TreeNode(1)
root1.left = TreeNode(2)
root1.right = TreeNode(3)
result1 = solution.maxPathSum(root1)
print("测试用例 1 结果:", result1) # 应该输出 6 (2->1->3)

# 测试用例 2:
#      -10
#      / \
#     9   20
#      / \
#     15   7
root2 = TreeNode(-10)
root2.left = TreeNode(9)
root2.right = TreeNode(20)
root2.right.left = TreeNode(15)
root2.right.right = TreeNode(7)
result2 = solution.maxPathSum(root2)
print("测试用例 2 结果:", result2) # 应该输出 42 (15->20->7)

# 测试用例 3:
#      5
#      / \
#     -3   4
#      / \   \
#     1   4   -2
root3 = TreeNode(5)
root3.left = TreeNode(-3)
root3.right = TreeNode(4)
root3.left.left = TreeNode(1)
root3.left.right = TreeNode(4)
root3.right.right = TreeNode(-2)
result3 = solution.maxPathSum(root3)
print("测试用例 3 结果:", result3) # 应该输出 10 (1->(-3)->5->4->(-2))

if __name__ == "__main__":
    main()
=====
```

文件: BinaryTreeRightSideView.cpp

```
=====

// LeetCode 199. Binary Tree Right Side View
// 题目链接: https://leetcode.cn/problems/binary-tree-right-side-view/
// 题目描述: 给定一个二叉树的根节点 root，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。
//
// 解题思路:
// 1. BFS 层序遍历: 记录每层的最后一个节点
// 2. DFS 递归遍历: 先访问右子树, 记录每层第一个访问到的节点
// 3. 使用队列进行 BFS 是更直观的解法
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
// 空间复杂度: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点
// 是否为最优解: 是, 这是解决右视图问题的标准方法
```

```
#include <vector>
#include <queue>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // 方法 1: BFS 层序遍历
    // 核心思想: 使用队列进行层序遍历, 记录每层的最后一个节点
    vector<int> rightSideViewBFS(TreeNode* root) {
        vector<int> result;
        if (root == nullptr) {
            return result;
        }

        queue<TreeNode*> q;
        q.push(root);
```

```

while (!q.empty()) {
    int levelSize = q.size();

    for (int i = 0; i < levelSize; i++) {
        TreeNode* node = q.front();
        q.pop();

        // 如果是当前层的最后一个节点，加入结果
        if (i == levelSize - 1) {
            result.push_back(node->val);
        }

        // 将子节点加入队列
        if (node->left != nullptr) {
            q.push(node->left);
        }
        if (node->right != nullptr) {
            q.push(node->right);
        }
    }

    return result;
}

// 方法2：DFS 递归遍历
// 核心思想：先访问右子树，再访问左子树，记录每层第一个访问到的节点
vector<int> rightSideViewDFS(TreeNode* root) {
    vector<int> result;
    dfs(root, result, 0);
    return result;
}

private:
    void dfs(TreeNode* node, vector<int>& result, int depth) {
        if (node == nullptr) {
            return;
        }

        // 如果当前深度还没有记录节点，说明这是该层第一个访问到的节点
        if (depth == result.size()) {
            result.push_back(node->val);
        }
    }
}

```

```
}

// 先递归右子树，再递归左子树
dfs(node->right, result, depth + 1);
dfs(node->left, result, depth + 1);

}

public:

// 提交如下的方法（使用 BFS 版本，更直观）
vector<int> rightSideView(TreeNode* root) {
    return rightSideViewBFS(root);
}

};

// 测试用例
// int main() {
//     Solution solution;
//
//     // 测试用例 1:
//     //      1
//     //     / \
//     //    2   3
//     //    \   \
//     //     5   4
//     // 右视图: [1, 3, 4]
//     TreeNode* root1 = new TreeNode(1);
//     root1->left = new TreeNode(2);
//     root1->right = new TreeNode(3);
//     root1->left->right = new TreeNode(5);
//     root1->right->right = new TreeNode(4);
//
//     // vector<int> result1 = solution.rightSideView(root1);
//     // 应该输出[1, 3, 4]
//
//     // 测试用例 2: 只有左子树的树
//     //      1
//     //     /
//     //    2
//     //   /
//     //  3
//     // 右视图: [1, 2, 3]
//     TreeNode* root2 = new TreeNode(1);
//     root2->left = new TreeNode(2);
```

```
//     root2->left->left = new TreeNode(3);  
//  
//     vector<int> result2 = solution.rightSideView(root2);  
//     // 应该输出[1, 2, 3]  
//  
//     // 内存清理...  
//  
//     return 0;  
// }
```

---

文件: BinaryTreeRightSideView.java

```
package class037;  
  
import java.util.*;  
  
// LeetCode 199. Binary Tree Right Side View  
// 题目链接: https://leetcode.cn/problems/binary-tree-right-side-view/  
// 题目描述: 给定一个二叉树的根节点 root，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。  
//  
// 解题思路:  
// 1. BFS 层序遍历: 记录每层的最后一个节点  
// 2. DFS 递归遍历: 先访问右子树, 记录每层第一个访问到的节点  
// 3. 使用队列进行 BFS 是更直观的解法  
//  
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次  
// 空间复杂度: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点  
// 是否为最优解: 是, 这是解决右视图问题的标准方法  
  
// 补充题目:  
// 1. LeetCode 102. Binary Tree Level Order Traversal - 二叉树的层序遍历  
// 2. LeetCode 103. Binary Tree Zigzag Level Order Traversal - 锯齿形层序遍历  
// 3. 牛客 NC14. 二叉树的之字形层序遍历 - 与 LeetCode 103 相同
```

```
public class BinaryTreeRightSideView {
```

```
    // 二叉树节点定义  
    public static class TreeNode {  
        int val;  
        TreeNode left;
```

```
TreeNode right;

TreeNode() {}

TreeNode(int val) {
    this.val = val;
}

TreeNode(int val, TreeNode left, TreeNode right) {
    this.val = val;
    this.left = left;
    this.right = right;
}

}

// 方法 1: BFS 层序遍历
// 核心思想: 使用队列进行层序遍历, 记录每层的最后一个节点
public List<Integer> rightSideViewBFS(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

            // 如果是当前层的最后一个节点, 加入结果
            if (i == levelSize - 1) {
                result.add(node.val);
            }

            // 将子节点加入队列
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
    }
    return result;
}
```

```

        }
    }

    return result;
}

// 方法 2: DFS 递归遍历
// 核心思想: 先访问右子树, 再访问左子树, 记录每层第一个访问到的节点
public List<Integer> rightSideViewDFS(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    dfs(root, result, 0);
    return result;
}

private void dfs(TreeNode node, List<Integer> result, int depth) {
    if (node == null) {
        return;
    }

    // 如果当前深度还没有记录节点, 说明这是该层第一个访问到的节点 (因为是先右后左)
    if (depth == result.size()) {
        result.add(node.val);
    }

    // 先递归右子树, 再递归左子树
    dfs(node.right, result, depth + 1);
    dfs(node.left, result, depth + 1);
}

// 提交如下的方法 (使用 BFS 版本, 更直观)
public List<Integer> rightSideView(TreeNode root) {
    return rightSideViewBFS(root);
}

// 测试用例
public static void main(String[] args) {
    BinaryTreeRightSideView solution = new BinaryTreeRightSideView();

    // 测试用例 1:
    //      1
    //     / \
    //    2   3
}

```

```
//      \
//      5   4
// 右视图: [1, 3, 4]
TreeNode root1 = new TreeNode(1);
root1.left = new TreeNode(2);
root1.right = new TreeNode(3);
root1.left.right = new TreeNode(5);
root1.right.right = new TreeNode(4);

List<Integer> result1 = solution.rightSideView(root1);
System.out.println("测试用例 1 结果: " + result1); // 应该输出[1, 3, 4]

// 测试用例 2:
//      1
//      /
//      2
//      /
//      3
// 右视图: [1, 2, 3]
TreeNode root2 = new TreeNode(1);
root2.left = new TreeNode(2);
root2.left.left = new TreeNode(3);

List<Integer> result2 = solution.rightSideView(root2);
System.out.println("测试用例 2 结果: " + result2); // 应该输出[1, 2, 3]

// 测试用例 3: 空树
TreeNode root3 = null;
List<Integer> result3 = solution.rightSideView(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出[]

// 测试用例 4: 完全二叉树
//      1
//      / \
//      2   3
//      / \ / \
//      4  5 6  7
// 右视图: [1, 3, 7]
TreeNode root4 = new TreeNode(1);
root4.left = new TreeNode(2);
root4.right = new TreeNode(3);
root4.left.left = new TreeNode(4);
root4.left.right = new TreeNode(5);
```

```

root4.right.left = new TreeNode(6);
root4.right.right = new TreeNode(7);

List<Integer> result4 = solution.rightSideView(root4);
System.out.println("测试用例 4 结果: " + result4); // 应该输出[1, 3, 7]

// 补充题目测试: 二叉树的左视图
System.out.println("\n==== 补充题目测试: 二叉树的左视图 ===");
List<Integer> leftView = leftSideView(root4);
System.out.println("左视图结果: " + leftView); // 应该输出[1, 2, 4]
}

// 补充功能: 二叉树的左视图
public static List<Integer> leftSideView(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

            // 如果是当前层的第一个节点, 加入结果
            if (i == 0) {
                result.add(node.val);
            }

            // 将子节点加入队列 (先左后右)
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
    }
}

```

```
        return result;
    }
}
```

=====

文件: BinaryTreeRightSideView.py

=====

```
# LeetCode 199. Binary Tree Right Side View
# 题目链接: https://leetcode.cn/problems/binary-tree-right-side-view/
# 题目描述: 给定一个二叉树的根节点 root，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。
#
# 解题思路:
# 1. BFS 层序遍历: 记录每层的最后一个节点
# 2. DFS 递归遍历: 先访问右子树, 记录每层第一个访问到的节点
# 3. 使用队列进行 BFS 是更直观的解法
#
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
# 空间复杂度: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点
# 是否为最优解: 是, 这是解决右视图问题的标准方法
```

```
from typing import List, Optional
from collections import deque
```

```
# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
    def rightSideViewBFS(self, root: Optional[TreeNode]) -> List[int]:
        """

```

BFS 层序遍历方法获取二叉树的右视图

Args:

root: 二叉树的根节点

Returns:

从右侧看到的节点值列表

"""

```
result = []
if root is None:
    return result

queue = deque([root])

while queue:
    level_size = len(queue)

    for i in range(level_size):
        node = queue.popleft()

        # 如果是当前层的最后一个节点，加入结果
        if i == level_size - 1:
            result.append(node.val)

        # 将子节点加入队列
        if node.left is not None:
            queue.append(node.left)
        if node.right is not None:
            queue.append(node.right)

    return result
```

```
def rightSideViewDFS(self, root: Optional[TreeNode]) -> List[int]:
```

```
"""

```

```
DFS 递归遍历方法获取二叉树的右视图
```

Args:

root: 二叉树的根节点

Returns:

从右侧看到的节点值列表

```
"""

```

```
result = []
self._dfs(root, result, 0)
return result
```

```
def _dfs(self, node: Optional[TreeNode], result: List[int], depth: int) -> None:
```

```
"""

```

```
DFS 递归辅助函数
```

Args:

```

node: 当前节点
result: 结果列表
depth: 当前深度
"""
if node is None:
    return

# 如果当前深度还没有记录节点，说明这是该层第一个访问到的节点
if depth == len(result):
    result.append(node.val)

# 先递归右子树，再递归左子树
self._dfs(node.right, result, depth + 1)
self._dfs(node.left, result, depth + 1)

def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
    """
    获取二叉树的右视图（使用 BFS 版本，更直观）

    Args:
        root: 二叉树的根节点

    Returns:
        从右侧看到的节点值列表
    """
    return self.rightSideViewBFS(root)

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    #      1
    #      / \
    #     2   3
    #     \   \
    #      5   4
    # 右视图: [1, 3, 4]
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.right = TreeNode(5)
    root1.right.right = TreeNode(4)

```

```
result1 = solution.rightSideView(root1)
print(f"测试用例 1 结果: {result1}") # 应该输出[1, 3, 4]

# 测试用例 2:
#      1
#      /
#      2
#      /
#      3
# 右视图: [1, 2, 3]
root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.left.left = TreeNode(3)

result2 = solution.rightSideView(root2)
print(f"测试用例 2 结果: {result2}") # 应该输出[1, 2, 3]

# 测试用例 3: 空树
root3 = None
result3 = solution.rightSideView(root3)
print(f"测试用例 3 结果: {result3}") # 应该输出[]

# 测试用例 4: 完全二叉树
#      1
#      / \
#      2   3
#      / \ / \
#      4   5 6   7
# 右视图: [1, 3, 7]
root4 = TreeNode(1)
root4.left = TreeNode(2)
root4.right = TreeNode(3)
root4.left.left = TreeNode(4)
root4.left.right = TreeNode(5)
root4.right.left = TreeNode(6)
root4.right.right = TreeNode(7)

result4 = solution.rightSideView(root4)
print(f"测试用例 4 结果: {result4}") # 应该输出[1, 3, 7]

# 补充题目测试: 二叉树的左视图
print("\n==== 补充题目测试: 二叉树的左视图 ===")
```

```
def leftSideView(root: Optional[TreeNode]) -> List[int]:  
    """  
    获取二叉树的左视图  
  
    Args:  
        root: 二叉树的根节点  
  
    Returns:  
        从左侧看到的节点值列表  
    """  
  
    result = []  
    if root is None:  
        return result  
  
    queue = deque([root])  
  
    while queue:  
        level_size = len(queue)  
  
        for i in range(level_size):  
            node = queue.popleft()  
  
            # 如果是当前层的第一个节点，加入结果  
            if i == 0:  
                result.append(node.val)  
  
            # 将子节点加入队列（先左后右）  
            if node.left is not None:  
                queue.append(node.left)  
            if node.right is not None:  
                queue.append(node.right)  
  
    return result  
  
left_view_result = leftSideView(root4)  
print(f"左视图结果: {left_view_result}") # 应该输出[1, 2, 4]  
  
if __name__ == "__main__":  
    main()  
  
=====
```

文件: BSTIterator.cpp

```
=====

// LeetCode 173. Binary Search Tree Iterator
// 题目链接: https://leetcode.cn/problems/binary-search-tree-iterator/
// 题目描述: 实现一个二叉搜索树迭代器类 BSTIterator , 表示一个按中序遍历二叉搜索树 (BST) 的迭代器:
// - BSTIterator(TreeNode root) 初始化 BSTIterator 类的一个对象。BST 的根节点 root 会作为构造函数的一部分给出。
// - boolean hasNext() 如果向指针右侧遍历存在数字，则返回 true ；否则返回 false 。
// - int next() 将指针向右移动，然后返回指针处的数字。
//
// 解题思路:
// 1. 使用栈模拟中序遍历的递归过程
// 2. 初始化时将根节点及其所有左子节点入栈
// 3. next()方法弹出栈顶节点，处理其右子树
// 4. hasNext()方法检查栈是否为空
//
// 时间复杂度:
// - 构造函数: O(h) - h 为树的高度
// - next(): 平均 O(1), 最坏 O(h)
// - hasNext(): O(1)
// 空间复杂度: O(h) - 栈中最多存储 h 个节点
// 是否为最优解: 是, 这是 BST 迭代器的标准实现
```

```
#include <stack>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class BSTIterator {
private:
    stack<TreeNode*> stk;

    /**
     * 将节点及其所有左子节点入栈

```

```

* 用于模拟中序遍历的左子树递归过程
*
* @param node 当前节点
*/
void pushAllLeft(TreeNode* node) {
    while (node != nullptr) {
        stk.push(node);
        node = node->left;
    }
}

public:
/***
 * BSTIterator 构造函数
 * 初始化时将根节点及其所有左子节点入栈
 *
 * @param root 二叉搜索树的根节点
 */
BSTIterator(TreeNode* root) {
    pushAllLeft(root);
}

/***
 * 返回中序遍历序列的下一个元素
 *
 * @return 下一个节点的值
 */
int next() {
    // 弹出栈顶节点（当前最小的节点）
    TreeNode* node = stk.top();
    stk.pop();

    // 如果该节点有右子树，将右子树及其所有左子节点入栈
    if (node->right != nullptr) {
        pushAllLeft(node->right);
    }

    return node->val;
}

/***
 * 检查是否还有下一个元素
 *

```

```
* @return 如果还有下一个元素返回 true, 否则返回 false
*/
bool hasNext() {
    return !stk.empty();
}

// 测试用例
// int main() {
//     // 构造测试 BST:
//     //      7
//     //    / \
//     //   3   15
//     //   / \
//     //  9   20
//     TreeNode* root = new TreeNode(7);
//     root->left = new TreeNode(3);
//     root->right = new TreeNode(15);
//     root->right->left = new TreeNode(9);
//     root->right->right = new TreeNode(20);
//
//     // 创建 BST 迭代器
//     BSTIterator iterator(root);
//
//     // 应该按顺序输出: 3, 7, 9, 15, 20
//     while (iterator.hasNext()) {
//         cout << iterator.next() << " ";
//     }
//     cout << endl;
//
//     // 内存清理
//     delete root->right->left;
//     delete root->right->right;
//     delete root->right;
//     delete root->left;
//     delete root;
//
//     // return 0;
// }
```

```
=====
package class037;

import java.util.Stack;

// LeetCode 173. Binary Search Tree Iterator
// 题目链接: https://leetcode.cn/problems/binary-search-tree-iterator/
// 题目描述: 实现一个二叉搜索树迭代器类 BSTIterator，表示一个按中序遍历二叉搜索树（BST）的迭代器:
// - BSTIterator(TreeNode root) 初始化 BSTIterator 类的一个对象。BST 的根节点 root 会作为构造函数的一部分给出。
// - boolean hasNext() 如果向指针右侧遍历存在数字，则返回 true；否则返回 false。
// - int next() 将指针向右移动，然后返回指针处的数字。
//
// 解题思路:
// 1. 使用栈模拟中序遍历的递归过程
// 2. 初始化时将根节点及其所有左子节点入栈
// 3. next()方法弹出栈顶节点，处理其右子树
// 4. hasNext()方法检查栈是否为空
//
// 时间复杂度:
// - 构造函数: O(h) - h 为树的高度
// - next(): 平均 O(1)，最坏 O(h)
// - hasNext(): O(1)
// 空间复杂度: O(h) - 栈中最多存储 h 个节点
// 是否为最优解: 是，这是 BST 迭代器的标准实现

// 补充题目:
// 1. LeetCode 94. Binary Tree Inorder Traversal - 二叉树的中序遍历
// 2. LeetCode 230. Kth Smallest Element in a BST - BST 中第 K 小的元素
// 3. LintCode 86. Binary Search Tree Iterator - 与 LeetCode 173 相同

public class BSTIterator {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
```

```

        this.val = val;
    }

    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

private Stack<TreeNode> stack;

/***
 * BSTIterator 构造函数
 * 初始化时将根节点及其所有左子节点入栈
 *
 * @param root 二叉搜索树的根节点
 */
public BSTIterator(TreeNode root) {
    stack = new Stack<>();
    // 将根节点及其所有左子节点入栈
    pushAllLeft(root);
}

/***
 * 将节点及其所有左子节点入栈
 * 用于模拟中序遍历的左子树递归过程
 *
 * @param node 当前节点
 */
private void pushAllLeft(TreeNode node) {
    while (node != null) {
        stack.push(node);
        node = node.left;
    }
}

/***
 * 返回中序遍历序列的下一个元素
 *
 * @return 下一个节点的值
 */
public int next() {

```

```
// 弹出栈顶节点（当前最小的节点）
TreeNode node = stack.pop();
// 如果该节点有右子树，将右子树及其所有左子节点入栈
if (node.right != null) {
    pushAllLeft(node.right);
}
return node.val;
}

/**
 * 检查是否还有下一个元素
 *
 * @return 如果还有下一个元素返回 true，否则返回 false
 */
public boolean hasNext() {
    return !stack.isEmpty();
}

// 测试用例
public static void main(String[] args) {
    // 构造测试 BST:
    //      7
    //      / \
    //      3   15
    //      /   \
    //      9   20
    TreeNode root = new TreeNode(7);
    root.left = new TreeNode(3);
    root.right = new TreeNode(15);
    root.right.left = new TreeNode(9);
    root.right.right = new TreeNode(20);

    // 创建 BST 迭代器
    BSTIterator iterator = new BSTIterator(root);

    System.out.println("BST 中序遍历结果:");
    // 应该按顺序输出: 3, 7, 9, 15, 20
    while (iterator.hasNext()) {
        System.out.print(iterator.next() + " ");
    }
    System.out.println(); // 换行

    // 测试用例 2: 空树
}
```

```

System.out.println("\n==> 测试用例 2: 空树 ==>");
BSTIterator emptyIterator = new BSTIterator(null);
System.out.println("空树 hasNext: " + emptyIterator.hasNext()); // 应该输出 false

// 测试用例 3: 单节点树
System.out.println("\n==> 测试用例 3: 单节点树 ==>");
TreeNode singleNode = new TreeNode(1);
BSTIterator singleIterator = new BSTIterator(singleNode);
System.out.println("单节点树遍历:");
while (singleIterator.hasNext()) {
    System.out.print(singleIterator.next() + " "); // 应该输出 1
}
System.out.println();

// 补充题目测试: LeetCode 230 - BST 中第 K 小的元素
System.out.println("\n==> 补充题目测试: BST 中第 K 小的元素 ==>");
int k = 3;
int kthSmallest = kthSmallest(root, k);
System.out.println("第" + k + "小的元素: " + kthSmallest); // 应该输出 9
}

// 补充题目: LeetCode 230. Kth Smallest Element in a BST
// 题目链接: https://leetcode.cn/problems/kth-smallest-element-in-a-bst/
// 使用 BST 迭代器思路找到第 K 小的元素
public static int kthSmallest(TreeNode root, int k) {
    Stack<TreeNode> stack = new Stack<>();
    TreeNode current = root;
    int count = 0;

    while (current != null || !stack.isEmpty()) {
        // 将当前节点及其所有左子节点入栈
        while (current != null) {
            stack.push(current);
            current = current.left;
        }

        // 弹出栈顶节点 (当前最小的节点)
        current = stack.pop();
        count++;

        // 如果找到第 K 小的元素, 返回
        if (count == k) {
            return current.val;
        }
    }
}

```

```

        }

    // 处理右子树
    current = current.right;
}

return -1; // 理论上不会执行到这里
}

// 方法 2: 递归实现中序遍历找到第 K 小的元素
private static int result;
private static int count;

public static int kthSmallestRecursive(TreeNode root, int k) {
    count = 0;
    result = -1;
    inorderTraverse(root, k);
    return result;
}

private static void inorderTraverse(TreeNode node, int k) {
    if (node == null || count >= k) {
        return;
    }

    // 中序遍历: 先左子树
    inorderTraverse(node.left, k);

    // 处理当前节点
    count++;
    if (count == k) {
        result = node.val;
        return;
    }

    // 中序遍历: 后右子树
    inorderTraverse(node.right, k);
}

```

```
=====
# LeetCode 173. Binary Search Tree Iterator
# 题目链接: https://leetcode.cn/problems/binary-search-tree-iterator/
# 题目描述: 实现一个二叉搜索树迭代器类 BSTIterator , 表示一个按中序遍历二叉搜索树 (BST) 的迭代器:
# - BSTIterator(TreeNode root) 初始化 BSTIterator 类的一个对象。BST 的根节点 root 会作为构造函数的一部分给出。
# - boolean hasNext() 如果向指针右侧遍历存在数字, 则返回 true ; 否则返回 false 。
# - int next() 将指针向右移动, 然后返回指针处的数字。
#
# 解题思路:
# 1. 使用栈模拟中序遍历的递归过程
# 2. 初始化时将根节点及其所有左子节点入栈
# 3. next() 方法弹出栈顶节点, 处理其右子树
# 4. hasNext() 方法检查栈是否为空
#
# 时间复杂度:
#   - 构造函数: O(h) - h 为树的高度
#   - next(): 平均 O(1), 最坏 O(h)
#   - hasNext(): O(1)
# 空间复杂度: O(h) - 栈中最多存储 h 个节点
# 是否为最优解: 是, 这是 BST 迭代器的标准实现
```

```
from typing import Optional

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class BSTIterator:
    def __init__(self, root: Optional[TreeNode]):
        """
        BSTIterator 构造函数
        初始化时将根节点及其所有左子节点入栈

        Args:
            root: 二叉搜索树的根节点
        """
        self.stack = []
        self._push_all_left(root)
```

```
def _push_all_left(self, node: Optional[TreeNode]) -> None:
```

```
    """
```

```
        将节点及其所有左子节点入栈
```

```
        用于模拟中序遍历的左子树递归过程
```

```
Args:
```

```
    node: 当前节点
```

```
    """
```

```
while node is not None:
```

```
    self.stack.append(node)
```

```
    node = node.left
```

```
def next(self) -> int:
```

```
    """
```

```
        返回中序遍历序列的下一个元素
```

```
Returns:
```

```
    下一个节点的值
```

```
    """
```

```
# 弹出栈顶节点（当前最小的节点）
```

```
node = self.stack.pop()
```

```
# 如果该节点有右子树，将右子树及其所有左子节点入栈
```

```
if node.right is not None:
```

```
    self._push_all_left(node.right)
```

```
return node.val
```

```
def hasNext(self) -> bool:
```

```
    """
```

```
        检查是否还有下一个元素
```

```
Returns:
```

```
    如果还有下一个元素返回 True，否则返回 False
```

```
    """
```

```
return len(self.stack) > 0
```

```
# 测试用例
```

```
def main():
```

```
    # 构造测试 BST:
```

```
    #      7
```

```
    #      / \
```

```
    #     3   15
```

```

#           /   \
#         9     20
root = TreeNode(7)
root.left = TreeNode(3)
root.right = TreeNode(15)
root.right.left = TreeNode(9)
root.right.right = TreeNode(20)

# 创建 BST 迭代器
iterator = BSTIterator(root)

print("BST 中序遍历结果:")
# 应该按顺序输出: 3, 7, 9, 15, 20
while iterator.hasNext():
    print(iterator.next(), end=" ")
print() # 换行

# 测试用例 2: 空树
print("\n==== 测试用例 2: 空树 ====")
empty_iterator = BSTIterator(None)
print("空树 hasNext:", empty_iterator.hasNext()) # 应该输出 False

# 测试用例 3: 单节点树
print("\n==== 测试用例 3: 单节点树 ====")
single_node = TreeNode(1)
single_iterator = BSTIterator(single_node)
print("单节点树遍历:")
while single_iterator.hasNext():
    print(single_iterator.next(), end=" ") # 应该输出 1
print()

# 补充题目测试: LeetCode 230 - BST 中第 K 小的元素
print("\n==== 补充题目测试: BST 中第 K 小的元素 ====")

```

```

def kth_smallest(root: Optional[TreeNode], k: int) -> int:
    """
    使用 BST 迭代器思路找到第 K 小的元素
    
```

Args:

root: 二叉搜索树的根节点  
k: 第 k 小的元素位置

Returns:

```

第 k 小的元素值

"""

stack = []
current = root
count = 0

while current is not None or stack:
    # 将当前节点及其所有左子节点入栈
    while current is not None:
        stack.append(current)
        current = current.left

    # 弹出栈顶节点（当前最小的节点）
    current = stack.pop()
    count += 1

    # 如果找到第 K 小的元素，返回
    if count == k:
        return current.val

    # 处理右子树
    current = current.right

return -1 # 理论上不会执行到这里

k = 3
kth_result = kth_smallest(root, k)
print(f"第 {k} 小的元素: {kth_result}") # 应该输出 9

if __name__ == "__main__":
    main()

```

=====

文件: CheckCompletenessOfBinaryTree.cpp

=====

```

// LeetCode 958. Check Completeness of a Binary Tree
// 题目链接: https://leetcode.cn/problems/check-completeness-of-a-binary-tree/
// 给定一个二叉树的 root ，确定它是否是一个完全二叉树
// 在一棵完全二叉树中，除了最后一层，其他层都被完全填满，
// 并且最后一层中的所有节点都尽可能靠左
//
// 解题思路：

```

```

// 1. 使用层序遍历(BFS)的方式遍历二叉树
// 2. 在遍历过程中，一旦遇到空节点，之后就不应该再有非空节点
// 3. 如果在空节点之后又遇到了非空节点，则不是完全二叉树
//
// 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
// 空间复杂度: O(w) - w 为树的最大宽度，队列中最多存储一层的节点
// 是否为最优解：是，这是检查完全二叉树的标准方法

// Definition for a binary tree node.
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    bool isCompleteTree(TreeNode* root) {
        // 使用数组模拟队列进行层序遍历（假设最多 1000 个节点）
        TreeNode* queue[1000];
        int front = 0, rear = 0;
        queue[rear++] = root;
        int foundNull = 0; // 标记是否遇到了空节点(用 0 表示 false, 1 表示 true)

        while (front < rear) {
            TreeNode* node = queue[front++];

            if (node == nullptr) {
                // 遇到空节点，标记为 true
                foundNull = 1;
            } else {
                // 遇到非空节点
                if (foundNull) {
                    // 如果之前已经遇到过空节点，说明不是完全二叉树
                    return false;
                }
                // 将左右子节点加入队列（即使是 null 也要加入）
                queue[rear++] = node->left;
                queue[rear++] = node->right;
            }
        }
    }
}

```

```
}

// 遍历完成，没有发现问题，是完全二叉树
return true;
}

};

// 测试代码示例（不提交）
/*
#include <iostream>
using namespace std;

int main() {
    Solution solution;

    // 测试用例 1：完全二叉树
    //      1
    //      / \
    //      2   3
    //      / \
    //      4   5
    TreeNode* root1 = new TreeNode(1);
    root1->left = new TreeNode(2);
    root1->right = new TreeNode(3);
    root1->left->left = new TreeNode(4);
    root1->left->right = new TreeNode(5);
    cout << "完全二叉树测试：" << (solution.isCompleteTree(root1) ? "true" : "false") << endl; //
应该输出 true

    // 测试用例 2：非完全二叉树
    //      1
    //      / \
    //      2   3
    //      /     \
    //      4       5
    TreeNode* root2 = new TreeNode(1);
    root2->left = new TreeNode(2);
    root2->right = new TreeNode(3);
    root2->left->left = new TreeNode(4);
    root2->right->right = new TreeNode(5);
    cout << "非完全二叉树测试：" << (solution.isCompleteTree(root2) ? "true" : "false") << endl;
// 应该输出 false
```

```

// 测试用例 3: 完全二叉树
//      1
//      / \
//     2   3
//    / \
//   4   5
//   /
// 6

TreeNode* root3 = new TreeNode(1);
root3->left = new TreeNode(2);
root3->right = new TreeNode(3);
root3->left->left = new TreeNode(4);
root3->left->right = new TreeNode(5);
root3->left->left->left = new TreeNode(6);
cout << "完全二叉树测试 2: " << (solution.isCompleteTree(root3) ? "true" : "false") << endl;
// 应该输出 true

return 0;
}
*/
=====

文件: CheckCompletenessOfBinaryTree.java
=====

package class037;

import java.util.LinkedList;
import java.util.Queue;

// LeetCode 958. Check Completeness of a Binary Tree
// 题目链接: https://leetcode.cn/problems/check-completeness-of-a-binary-tree/
// 给定一个二叉树的 root，确定它是否是一个完全二叉树
// 在一棵完全二叉树中，除了最后一层，其他层都被完全填满，
// 并且最后一层中的所有节点都尽可能靠左
//
// 解题思路:
// 1. 使用层序遍历(BFS)的方式遍历二叉树
// 2. 在遍历过程中，一旦遇到空节点，之后就不应该再有非空节点
// 3. 如果在空节点之后又遇到了非空节点，则不是完全二叉树
//
// 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
// 空间复杂度: O(w) - w 为树的最大宽度，队列中最多存储一层的节点

```

```
// 是否为最优解：是，这是检查完全二叉树的标准方法

public class CheckCompletenessOfBinaryTree {

    // 不提交这个类
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {
        }

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 提交如下的方法
    public boolean isCompleteTree(TreeNode root) {
        // 使用队列进行层序遍历
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        boolean foundNull = false; // 标记是否遇到了空节点

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();

            if (node == null) {
                // 遇到空节点，标记为 true
                foundNull = true;
            } else {
                // 遇到非空节点
                if (foundNull) {
                    // 如果之前已经遇到过空节点，说明不是完全二叉树
                    return false;
                }
            }
        }
    }
}
```

```
// 将左右子节点加入队列（即使是 null 也要加入）
queue.offer(node.left);
queue.offer(node.right);

}

}

// 遍历完成，没有发现问题，是完全二叉树
return true;
}

// 测试用例
public static void main(String[] args) {
    CheckCompletenessOfBinaryTree solution = new CheckCompletenessOfBinaryTree();

    // 测试用例 1：完全二叉树
    //      1
    //      / \
    //      2   3
    //      / \
    //      4   5
    TreeNode root1 = new TreeNode(1);
    root1.left = new TreeNode(2);
    root1.right = new TreeNode(3);
    root1.left.left = new TreeNode(4);
    root1.left.right = new TreeNode(5);
    System.out.println("完全二叉树测试：" + solution.isCompleteTree(root1)); // 应该输出 true

    // 测试用例 2：非完全二叉树
    //      1
    //      / \
    //      2   3
    //      /     \
    //      4       5
    TreeNode root2 = new TreeNode(1);
    root2.left = new TreeNode(2);
    root2.right = new TreeNode(3);
    root2.left.left = new TreeNode(4);
    root2.right.right = new TreeNode(5);
    System.out.println("非完全二叉树测试：" + solution.isCompleteTree(root2)); // 应该输出
false

    // 测试用例 3：完全二叉树
    //      1
```

```

//      / \
//     2   3
//     / \
//    4   5
//   /
// 6

TreeNode root3 = new TreeNode(1);
root3.left = new TreeNode(2);
root3.right = new TreeNode(3);
root3.left.left = new TreeNode(4);
root3.left.right = new TreeNode(5);
root3.left.left.left = new TreeNode(6);
System.out.println("完全二叉树测试 2: " + solution.isCompleteTree(root3)); // 应该输出 true
}

=====

文件: CheckCompletenessOfBinaryTree.py
=====

# LeetCode 958. Check Completeness of a Binary Tree
# 题目链接: https://leetcode.cn/problems/check-completeness-of-a-binary-tree/
# 给定一个二叉树的 root ，确定它是否是一个完全二叉树
# 在一棵完全二叉树中，除了最后一层，其他层都被完全填满，
# 并且最后一层中的所有节点都尽可能靠左
#
# 解题思路：
# 1. 使用层序遍历(BFS)的方式遍历二叉树
# 2. 在遍历过程中，一旦遇到空节点，之后就不应该再有非空节点
# 3. 如果在空节点之后又遇到了非空节点，则不是完全二叉树
#
# 时间复杂度: O(n) - n 为树中节点的数量，需要遍历所有节点
# 空间复杂度: O(w) - w 为树的最大宽度，队列中最多存储一层的节点
# 是否为最优解：是，这是检查完全二叉树的标准方法
```

```

from typing import Optional
from collections import deque

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
    self.right = right

def isCompleteTree(root: Optional[TreeNode]) -> bool:
    """
    判断给定的二叉树是否是完全二叉树

    Args:
        root: 二叉树的根节点

    Returns:
        如果是完全二叉树返回 True, 否则返回 False
    """
    if not root:
        return True

    # 使用队列进行层序遍历
    queue: deque[Optional[TreeNode]] = deque([root])
    found_null = False # 标记是否遇到了空节点

    while queue:
        node = queue.popleft()

        if node is None:
            # 遇到空节点, 标记为 True
            found_null = True
        else:
            # 遇到非空节点
            if found_null:
                # 如果之前已经遇到过空节点, 说明不是完全二叉树
                return False
            # 将左右子节点加入队列 (即使是 None 也要加入)
            queue.append(node.left)
            queue.append(node.right)

    # 遍历完成, 没有发现问题, 是完全二叉树
    return True

# 测试用例
if __name__ == "__main__":
    # 测试用例 1: 完全二叉树
    #      1
    #     / \
    #    2   3
```

```
#      / \
#    4   5
root1 = TreeNode(1)
root1.left = TreeNode(2)
root1.right = TreeNode(3)
root1.left.left = TreeNode(4)
root1.left.right = TreeNode(5)
print(f"完全二叉树测试: {isCompleteTree(root1)}") # 应该输出 True
```

# 测试用例 2: 非完全二叉树

```
#      1
#      / \
#    2   3
#      /   \
#    4       5
root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.right = TreeNode(3)
root2.left.left = TreeNode(4)
root2.right.right = TreeNode(5)
print(f"非完全二叉树测试: {isCompleteTree(root2)}") # 应该输出 False
```

# 测试用例 3: 完全二叉树

```
#      1
#      / \
#    2   3
#      / \
#    4   5
#      /
#    6
root3 = TreeNode(1)
root3.left = TreeNode(2)
root3.right = TreeNode(3)
root3.left.left = TreeNode(4)
root3.left.right = TreeNode(5)
root3.left.left.left = TreeNode(6)
print(f"完全二叉树测试 2: {isCompleteTree(root3)}") # 应该输出 True
```

---

文件: Code01\_LowestCommonAncestor.java

---

```
package class037;
```

```

// 普通二叉树上寻找两个节点的最近公共祖先
// 测试链接 : https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
public class Code01_LowestCommonAncestor {

    // 不提交这个类
    public static class TreeNode {
        public int val;
        public TreeNode left;
        public TreeNode right;
    }

    // 提交如下的方法
    public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) {
            // 遇到空, 或者 p, 或者 q, 直接返回
            return root;
        }

        TreeNode l = lowestCommonAncestor(root.left, p, q);
        TreeNode r = lowestCommonAncestor(root.right, p, q);
        if (l != null && r != null) {
            // 左树也搜到, 右树也搜到, 返回 root
            return root;
        }
        if (l == null && r == null) {
            // 都没搜到返回空
            return null;
        }
        // l 和 r 一个为空, 一个不为空
        // 返回不空的那个
        return l != null ? l : r;
    }
}

```

=====

文件: Code02\_LowestCommonAncestorBinarySearch.java

=====

```

package class037;

// 搜索二叉树上寻找两个节点的最近公共祖先
// 测试链接 : https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-search-tree/

```

```

public class Code02_LowestCommonAncestorBinarySearch {

    // 不提交这个类
    public static class TreeNode {
        public int val;
        public TreeNode left;
        public TreeNode right;
    }

    // 提交如下的方法
    public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        // root 从上到下
        // 如果先遇到了 p, 说明 p 是答案
        // 如果先遇到了 q, 说明 q 是答案
        // 如果 root 在 p~q 的值之间, 不用管 p 和 q 谁大谁小, 只要 root 在中间, 那么此时的 root 就是答案
        // 如果 root 在 p~q 的值的左侧, 那么 root 往右移动
        // 如果 root 在 p~q 的值的右侧, 那么 root 往左移动
        while (root.val != p.val && root.val != q.val) {
            if (Math.min(p.val, q.val) < root.val && root.val < Math.max(p.val, q.val)) {
                break;
            }
            root = root.val < Math.min(p.val, q.val) ? root.right : root.left;
        }
        return root;
    }
}

```

=====

文件: Code03\_PathSumII.java

=====

```

package class037;

import java.util.ArrayList;
import java.util.List;

// 收集累加和等于 aim 的所有路径
// 测试链接 : https://leetcode.cn/problems/path-sum-ii/
public class Code03_PathSumII {

    // 不提交这个类
    public static class TreeNode {

```

```

public int val;
public TreeNode left;
public TreeNode right;
}

// 提交如下的方法
public static List<List<Integer>> pathSum(TreeNode root, int aim) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        List<Integer> path = new ArrayList<>();
        f(root, aim, 0, path, ans);
    }
    return ans;
}

public static void f(TreeNode cur, int aim, int sum, List<Integer> path, List<List<Integer>> ans) {
    if (cur.left == null && cur.right == null) {
        // 叶节点
        if (cur.val + sum == aim) {
            path.add(cur.val);
            copy(path, ans);
            path.remove(path.size() - 1);
        }
    } else {
        // 不是叶节点
        path.add(cur.val);
        if (cur.left != null) {
            f(cur.left, aim, sum + cur.val, path, ans);
        }
        if (cur.right != null) {
            f(cur.right, aim, sum + cur.val, path, ans);
        }
        path.remove(path.size() - 1);
    }
}

public static void copy(List<Integer> path, List<List<Integer>> ans) {
    List<Integer> copy = new ArrayList<>();
    for (Integer num : path) {
        copy.add(num);
    }
    ans.add(copy);
}

```

```
}
```

```
}
```

```
=====
```

文件: Code04\_BalancedBinaryTree.java

```
=====
```

```
package class037;
```

```
// 验证平衡二叉树
```

```
// 测试链接 : https://leetcode.cn/problems/balanced-binary-tree/
```

```
public class Code04_BalancedBinaryTree {
```

```
// 不提交这个类
```

```
public static class TreeNode {
```

```
    public int val;
```

```
    public TreeNode left;
```

```
    public TreeNode right;
```

```
}
```

```
// 提交如下的方法
```

```
public static boolean balance;
```

```
public static boolean isBalanced(TreeNode root) {
```

```
    // balance 是全局变量，所有调用过程共享
```

```
    // 所以每次判断开始时，设置为 true
```

```
    balance = true;
```

```
    height(root);
```

```
    return balance;
```

```
}
```

```
// 一旦发现不平衡，返回什么高度已经不重要了
```

```
public static int height(TreeNode cur) {
```

```
    if (!balance || cur == null) {
```

```
        return 0;
```

```
}
```

```
    int lh = height(cur.left);
```

```
    int rh = height(cur.right);
```

```
    if (Math.abs(lh - rh) > 1) {
```

```
        balance = false;
```

```
}
```

```
    return Math.max(lh, rh) + 1;
```

```
}
```

```
}
```

```
=====
```

文件: Code05\_ValidateBinarySearchTree. java

```
=====
```

```
package class037;
```

```
// 验证搜索二叉树
```

```
// 测试链接 : https://leetcode.cn/problems/validate-binary-search-tree/
```

```
public class Code05_ValidateBinarySearchTree {
```

```
// 不提交这个类
```

```
public static class TreeNode {
```

```
    public int val;
```

```
    public TreeNode left;
```

```
    public TreeNode right;
```

```
}
```

```
// 提交以下的方法
```

```
public static int MAXN = 10001;
```

```
public static TreeNode[] stack = new TreeNode[MAXN];
```

```
public static int r;
```

```
// 提交时改名为 isValidBST
```

```
public static boolean isValidBST1(TreeNode head) {
```

```
    if (head == null) {
```

```
        return true;
```

```
}
```

```
TreeNode pre = null;
```

```
r = 0;
```

```
while (r > 0 || head != null) {
```

```
    if (head != null) {
```

```
        stack[r++] = head;
```

```
        head = head.left;
```

```
    } else {
```

```
        head = stack[--r];
```

```
        if (pre != null && pre.val >= head.val) {
```

```
            return false;
```

```

        }
        pre = head;
        head = head.right;
    }
}

return true;
}

public static long min, max;

// 提交时改名为 isValidBST
public static boolean isValidBST2(TreeNode head) {
    if (head == null) {
        min = Long.MAX_VALUE;
        max = Long.MIN_VALUE;
        return true;
    }

    boolean lok = isValidBST2(head.left);
    long lmin = min;
    long lmax = max;
    boolean rok = isValidBST2(head.right);
    long rmin = min;
    long rmax = max;
    min = Math.min(Math.min(lmin, rmin), head.val);
    max = Math.max(Math.max(lmax, rmax), head.val);
    return lok && rok && lmax < head.val && head.val < rmin;
}

}

```

}

=====

文件: Code06\_TrimBinarySearchTree.java

=====

```

package class037;

// 修剪搜索二叉树
// 测试链接 : https://leetcode.cn/problems/trim-a-binary-search-tree/
public class Code06_TrimBinarySearchTree {

    // 不提交这个类
    public static class TreeNode {
        public int val;

```

```
public TreeNode left;
public TreeNode right;
}

// 提交以下的方法
// [low, high]
public static TreeNode trimBST(TreeNode cur, int low, int high) {
    if (cur == null) {
        return null;
    }
    if (cur.val < low) {
        return trimBST(cur.right, low, high);
    }
    if (cur.val > high) {
        return trimBST(cur.left, low, high);
    }
    // cur 在范围内
    cur.left = trimBST(cur.left, low, high);
    cur.right = trimBST(cur.right, low, high);
    return cur;
}

}
```

}

=====

文件: Code07\_HouseRobberIII.java

=====

```
package class037;

// 二叉树打家劫舍问题
// 测试链接 : https://leetcode.cn/problems/house-robber-iii/
public class Code07_HouseRobberIII {

    // 不提交这个类
    public static class TreeNode {
        public int val;
        public TreeNode left;
        public TreeNode right;
    }

    // 提交如下的方法
    public static int rob(TreeNode root) {
```

```

    f(root);
    return Math.max(yes, no);
}

// 全局变量，完成了 X 子树的遍历，返回之后
// yes 变成，X 子树在偷头节点的情况下，最大的收益
public static int yes;

// 全局变量，完成了 X 子树的遍历，返回之后
// no 变成，X 子树在不偷头节点的情况下，最大的收益
public static int no;

public static void f(TreeNode root) {
    if (root == null) {
        yes = 0;
        no = 0;
    } else {
        int y = root.val;
        int n = 0;
        f(root.left);
        y += no;
        n += Math.max(yes, no);
        f(root.right);
        y += no;
        n += Math.max(yes, no);
        yes = y;
        no = n;
    }
}
}

```

}

---

文件: Code08\_BinaryTreeVerticalOrderTraversal.cpp

---

```

// LeetCode 314. Binary Tree Vertical Order Traversal
// 题目链接: https://leetcode.cn/problems/binary-tree-vertical-order-traversal/
// 题目描述: 给你一个二叉树的根结点，返回其节点按垂直方向（从上到下，逐列）遍历的结果。
// 如果两个节点在同一行和列，那么顺序则为从左到右。
//
// 解题思路:
// 1. 使用 BFS 层序遍历，同时记录每个节点的列号

```

```

// 2. 根节点列号为 0, 左子节点列号减 1, 右子节点列号加 1
// 3. 使用 unordered_map 记录每列的节点值列表
// 4. 使用 minCol 和 maxCol 记录列号的范围
// 5. 按列号从小到大收集结果
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
// 空间复杂度: O(n) - 队列和 unordered_map 最多存储 n 个节点
// 是否为最优解: 是, 这是垂直遍历的标准解法

#include <vector>
#include <queue>
#include <unordered_map>
#include <algorithm>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * 二叉树垂直遍历
     *
     * @param root 二叉树的根节点
     * @return 按垂直方向遍历的节点值列表
     */
    vector<vector<int>> verticalOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
            return result;
        }

        // 使用 unordered_map 记录每列的节点值列表
        unordered_map<int, vector<int>> columnMap;

        // 使用队列进行 BFS, 存储节点和对应的列号

```

```
queue<pair<TreeNode*, int>> q; // 存储节点和列号的 pair

q.push({root, 0});

// 记录列号的范围
int minCol = 0, maxCol = 0;

while (!q.empty()) {
    auto p = q.front();
    q.pop();
    TreeNode* node = p.first;
    int col = p.second;

    // 将节点值添加到对应列的列表中
    columnMap[col].push_back(node->val);

    // 更新列号范围
    minCol = min(minCol, col);
    maxCol = max(maxCol, col);

    // 处理左右子节点
    if (node->left != nullptr) {
        q.push({node->left, col - 1});
    }
    if (node->right != nullptr) {
        q.push({node->right, col + 1});
    }
}

// 按列号从小到大收集结果
for (int i = minCol; i <= maxCol; i++) {
    result.push_back(columnMap[i]);
}

return result;
};

// 测试用例
// int main() {
//     Solution solution;
//     //
//     // 测试用例 1:
// }
```

```
//      //      3
//      //      / \
//      //      9   20
//      //      /   \
//      //      15   7
//      // 垂直遍历结果: [[9], [3, 15], [20], [7]]
// TreeNode* root1 = new TreeNode(3);
// root1->left = new TreeNode(9);
// root1->right = new TreeNode(20);
// root1->right->left = new TreeNode(15);
// root1->right->right = new TreeNode(7);
//
// vector<vector<int>> result1 = solution.verticalOrder(root1);
// // 应该输出[[9], [3, 15], [20], [7]]
//
// // 测试用例 2:
//      //      3
//      //      / \
//      //      9   8
//      //      / \   \
//      //      4   0   1
//      //      / \   \
//      //      5   2   7
//      // 垂直遍历结果: [[4], [9, 5], [3, 0, 1], [8, 2], [7]]
// TreeNode* root2 = new TreeNode(3);
// root2->left = new TreeNode(9);
// root2->right = new TreeNode(8);
// root2->left->left = new TreeNode(4);
// root2->left->right = new TreeNode(0);
// root2->right->right = new TreeNode(1);
// root2->left->right->left = new TreeNode(5);
// root2->left->right->right = new TreeNode(2);
// root2->right->right->right = new TreeNode(7);
//
// vector<vector<int>> result2 = solution.verticalOrder(root2);
// // 应该输出[[4], [9, 5], [3, 0, 1], [8, 2], [7]]
//
// // 测试用例 3: 空树
// TreeNode* root3 = nullptr;
// vector<vector<int>> result3 = solution.verticalOrder(root3);
// // 应该输出[]
//
// // 内存清理...
```

```
//  
//      return 0;  
// }
```

=====

文件: Code08\_BinaryTreeVerticalOrderTraversal.java

=====

```
package class037;  
  
import java.util.*;  
  
// LeetCode 314. Binary Tree Vertical Order Traversal  
// 题目链接: https://leetcode.cn/problems/binary-tree-vertical-order-traversal/  
// 题目描述: 给你一个二叉树的根结点, 返回其节点按垂直方向(从上到下, 逐列)遍历的结果。  
// 如果两个节点在同一行和列, 那么顺序则为从左到右。  
//  
// 解题思路:  
// 1. 使用BFS层序遍历, 同时记录每个节点的列号  
// 2. 根节点列号为0, 左子节点列号减1, 右子节点列号加1  
// 3. 使用HashMap记录每列的节点值列表  
// 4. 使用minCol和maxCol记录列号的范围  
// 5. 按列号从小到大收集结果  
//  
// 时间复杂度: O(n) - n为树中节点的数量, 每个节点访问一次  
// 空间复杂度: O(n) - 队列和HashMap最多存储n个节点  
// 是否为最优解: 是, 这是垂直遍历的标准解法
```

```
public class Code08_BinaryTreeVerticalOrderTraversal {
```

```
    // 二叉树节点定义
```

```
    public static class TreeNode {  
        int val;  
        TreeNode left;  
        TreeNode right;
```

```
        TreeNode() {}
```

```
        TreeNode(int val) {  
            this.val = val;  
        }
```

```
        TreeNode(int val, TreeNode left, TreeNode right) {
```

```

        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// 提交如下的方法
public List<List<Integer>> verticalOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 使用 HashMap 记录每列的节点值列表
    Map<Integer, List<Integer>> columnMap = new HashMap<>();

    // 使用队列进行 BFS，存储节点和对应的列号
    Queue<TreeNode> nodeQueue = new LinkedList<>();
    Queue<Integer> columnQueue = new LinkedList<>();

    nodeQueue.offer(root);
    columnQueue.offer(0);

    // 记录列号的范围
    int minCol = 0, maxCol = 0;

    while (!nodeQueue.isEmpty()) {
        TreeNode node = nodeQueue.poll();
        int col = columnQueue.poll();

        // 将节点值添加到对应列的列表中
        columnMap.computeIfAbsent(col, k -> new ArrayList<>()).add(node.val);

        // 更新列号范围
        minCol = Math.min(minCol, col);
        maxCol = Math.max(maxCol, col);

        // 处理左右子节点
        if (node.left != null) {
            nodeQueue.offer(node.left);
            columnQueue.offer(col - 1);
        }
        if (node.right != null) {
    }
}

```

```

        nodeQueue.offer(node.right);
        columnQueue.offer(col + 1);
    }
}

// 按列号从小到大收集结果
for (int i = minCol; i <= maxCol; i++) {
    result.add(columnMap.get(i));
}

return result;
}

// 测试用例
public static void main(String[] args) {
    Code08_BinaryTreeVerticalOrderTraversal solution = new
Code08_BinaryTreeVerticalOrderTraversal();

    // 测试用例 1:
    //      3
    //      / \
    //      9  20
    //      /   \
    //      15  7
    // 垂直遍历结果: [[9], [3, 15], [20], [7]]
    TreeNode root1 = new TreeNode(3);
    root1.left = new TreeNode(9);
    root1.right = new TreeNode(20);
    root1.right.left = new TreeNode(15);
    root1.right.right = new TreeNode(7);

    List<List<Integer>> result1 = solution.verticalOrder(root1);
    System.out.println("测试用例 1 结果: " + result1); // 应该输出[[9], [3, 15], [20], [7]]

    // 测试用例 2:
    //      3
    //      / \
    //      9  8
    //      / \   \
    //      4  0   1
    //      / \   \
    //      5  2   7
    // 垂直遍历结果: [[4], [9, 5], [3, 0, 1], [8, 2], [7]]
}

```

```

TreeNode root2 = new TreeNode(3);
root2.left = new TreeNode(9);
root2.right = new TreeNode(8);
root2.left.left = new TreeNode(4);
root2.left.right = new TreeNode(0);
root2.right.right = new TreeNode(1);
root2.left.right.left = new TreeNode(5);
root2.left.right.right = new TreeNode(2);
root2.right.right.right = new TreeNode(7);

List<List<Integer>> result2 = solution.verticalOrder(root2);
System.out.println("测试用例 2 结果: " + result2); // 应该输出[[4], [9, 5], [3, 0, 1], [8, 2], [7]]


// 测试用例 3: 空树
TreeNode root3 = null;
List<List<Integer>> result3 = solution.verticalOrder(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出[]

}
}

```

=====

文件: Code08\_BinaryTreeVerticalOrderTraversal.py

=====

```

# LeetCode 314. Binary Tree Vertical Order Traversal
# 题目链接: https://leetcode.cn/problems/binary-tree-vertical-order-traversal/
# 题目描述: 给你一个二叉树的根结点，返回其节点按垂直方向（从上到下，逐列）遍历的结果。
# 如果两个节点在同一行和列，那么顺序则为从左到右。
#
# 解题思路:
# 1. 使用 BFS 层序遍历，同时记录每个节点的列号
# 2. 根节点列号为 0，左子节点列号减 1，右子节点列号加 1
# 3. 使用字典记录每列的节点值列表
# 4. 使用 minCol 和 maxCol 记录列号的范围
# 5. 按列号从小到大收集结果
#
# 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
# 空间复杂度: O(n) - 队列和字典最多存储 n 个节点
# 是否为最优解: 是，这是垂直遍历的标准解法

```

```

from typing import List, Optional
from collections import deque, defaultdict

```

```
# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def verticalOrder(root: Optional[TreeNode]) -> List[List[int]]:
    """
    二叉树垂直遍历

    Args:
        root: 二叉树的根节点

    Returns:
        按垂直方向遍历的节点值列表
    """
    result = []
    if root is None:
        return result

    # 使用字典记录每列的节点值列表
    column_map = defaultdict(list)

    # 使用队列进行 BFS，存储节点和对应的列号
    node_queue = deque([root])
    column_queue = deque([0])

    # 记录列号的范围
    min_col, max_col = 0, 0

    while node_queue:
        node = node_queue.popleft()
        col = column_queue.popleft()

        # 将节点值添加到对应列的列表中
        column_map[col].append(node.val)

        # 更新列号范围
        min_col = min(min_col, col)
        max_col = max(max_col, col)
```

```

# 处理左右子节点
if node.left is not None:
    node_queue.append(node.left)
    column_queue.append(col - 1)
if node.right is not None:
    node_queue.append(node.right)
    column_queue.append(col + 1)

# 按列号从小到大收集结果
for i in range(min_col, max_col + 1):
    result.append(column_map[i])

return result

# 测试用例
if __name__ == "__main__":
    # 测试用例 1:
    #      3
    #     / \
    #    9  20
    #   / \
    #  15  7
    # 垂直遍历结果: [[9], [3, 15], [20], [7]]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)

    result1 = verticalOrder(root1)
    print(f"测试用例 1 结果: {result1}")  # 应该输出[[9], [3, 15], [20], [7]]

    # 测试用例 2:
    #      3
    #     / \
    #    9  8
    #   / \  \
    #  4  0  1
    #   / \  \
    #  5  2  7
    # 垂直遍历结果: [[4], [9, 5], [3, 0, 1], [8, 2], [7]]
    root2 = TreeNode(3)
    root2.left = TreeNode(9)

```

```

root2.right = TreeNode(8)
root2.left.left = TreeNode(4)
root2.left.right = TreeNode(0)
root2.right.right = TreeNode(1)
root2.left.right.left = TreeNode(5)
root2.left.right.right = TreeNode(2)
root2.right.right.right = TreeNode(7)

result2 = verticalOrder(root2)
print(f"测试用例 2 结果: {result2}") # 应该输出[[4], [9, 5], [3, 0, 1], [8, 2], [7]]

```

```

# 测试用例 3: 空树
root3 = None
result3 = verticalOrder(root3)
print(f"测试用例 3 结果: {result3}") # 应该输出[]

```

=====

文件: Code09\_LowestCommonAncestorIII.cpp

=====

```

// LeetCode 1650. Lowest Common Ancestor of a Binary Tree III
// 题目链接: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree-iii/
// 题目描述: 给定二叉树中的两个节点 p 和 q，返回它们的最低公共祖先(LCA)。
// 每个节点都有指向其父节点的引用。
//
// 解题思路:
// 1. 利用每个节点都有父指针的特点
// 2. 首先从节点 p 向上遍历到根节点，将路径上的所有节点存储在 unordered_set 中
// 3. 然后从节点 q 向上遍历，第一个出现在 unordered_set 中的节点就是 LCA
//
// 时间复杂度: O(h) - h 为树的高度，最坏情况下需要遍历从叶子节点到根节点的路径两次
// 空间复杂度: O(h) - unordered_set 最多存储从 p 到根节点的 h 个节点
// 是否为最优解: 是，这是利用父指针求 LCA 的标准解法

```

```

#include <unordered_set>
using namespace std;

// 二叉树节点定义（带父指针）
class Node {
public:
    int val;
    Node* left;
    Node* right;
}

```

```

Node* parent;

Node(int x) : val(x), left(nullptr), right(nullptr), parent(nullptr) {}

};

class Solution {
public:
    /**
     * 寻找二叉树中两个节点的最低公共祖先
     *
     * @param p 第一个节点
     * @param q 第二个节点
     * @return 最低公共祖先节点
     */
    Node* lowestCommonAncestor(Node* p, Node* q) {
        // 创建 unordered_set 存储节点 p 的所有祖先节点（包括 p 本身）
        unordered_set<Node*> visitedAncestors;

        // 从节点 p 向上遍历到根节点，将路径上的所有节点加入 unordered_set
        Node* current = p;
        while (current != nullptr) {
            visitedAncestors.insert(current);
            current = current->parent;
        }

        // 从节点 q 向上遍历到根节点，第一个出现在 unordered_set 中的节点就是 LCA
        current = q;
        while (current != nullptr) {
            if (visitedAncestors.count(current)) {
                return current; // 找到 LCA
            }
            current = current->parent;
        }

        return nullptr; // 理论上不会执行到这里
    }
};

// 测试用例
// int main() {
//     Solution solution;
//     //
//     // 构建测试用例:

```

```
//      //      3
//      //      / \
//      //      5   1
//      //      / \   \
//      //      6   2   8
//      //      / \
//      //      7   4
//
//      Node* root = new Node(3);
//      Node* node5 = new Node(5);
//      Node* node1 = new Node(1);
//      Node* node6 = new Node(6);
//      Node* node2 = new Node(2);
//      Node* node8 = new Node(8);
//      Node* node7 = new Node(7);
//      Node* node4 = new Node(4);
//
//      // 建立父子关系
//      root->left = node5;
//      root->right = node1;
//      node5->parent = root;
//      node1->parent = root;
//
//      node5->left = node6;
//      node5->right = node2;
//      node6->parent = node5;
//      node2->parent = node5;
//
//      node1->right = node8;
//      node8->parent = node1;
//
//      node2->left = node7;
//      node2->right = node4;
//      node7->parent = node2;
//      node4->parent = node2;
//
//      // 测试用例 1: p=7, q=4, LCA 应该是 2
//      Node* result1 = solution.lowestCommonAncestor(node7, node4);
//      // 应该输出 2
//
//      // 测试用例 2: p=6, q=8, LCA 应该是 3
//      Node* result2 = solution.lowestCommonAncestor(node6, node8);
//      // 应该输出 3
```

```
//  
//    // 测试用例 3: p=5, q=1, LCA 应该是 3  
//    Node* result3 = solution.lowestCommonAncestor(node5, node1);  
//    // 应该输出 3  
//  
//    return 0;  
// }
```

=====

文件: Code09\_LowestCommonAncestorIII.java

=====

```
package class037;  
  
import java.util.*;  
  
// LeetCode 1650. Lowest Common Ancestor of a Binary Tree III  
// 题目链接: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree-iii/  
// 题目描述: 给定二叉树中的两个节点 p 和 q, 返回它们的最低公共祖先(LCA)。  
// 每个节点都有指向其父节点的引用。  
  
//  
// 解题思路:  
// 1. 利用每个节点都有父指针的特点  
// 2. 首先从节点 p 向上遍历到根节点, 将路径上的所有节点存储在 HashSet 中  
// 3. 然后从节点 q 向上遍历, 第一个出现在 HashSet 中的节点就是 LCA  
  
//  
// 时间复杂度: O(h) - h 为树的高度, 最坏情况下需要遍历从叶子节点到根节点的路径两次  
// 空间复杂度: O(h) - HashSet 最多存储从 p 到根节点的 h 个节点  
// 是否为最优解: 是, 这是利用父指针求 LCA 的标准解法
```

```
public class Code09_LowestCommonAncestorIII {
```

```
    // 二叉树节点定义 (带父指针)
```

```
    static class Node {  
        public int val;  
        public Node left;  
        public Node right;  
        public Node parent;  
  
        public Node(int val) {  
            this.val = val;  
        }  
    }
```

```

/**
 * 寻找二叉树中两个节点的最低公共祖先
 *
 * @param p 第一个节点
 * @param q 第二个节点
 * @return 最低公共祖先节点
 */
public Node lowestCommonAncestor(Node p, Node q) {
    // 创建 HashSet 存储节点 p 的所有祖先节点（包括 p 本身）
    Set<Node> visitedNodes = new HashSet<>();

    // 从节点 p 向上遍历到根节点，将路径上的所有节点加入 HashSet
    Node current = p;
    while (current != null) {
        visitedNodes.add(current);
        current = current.parent;
    }

    // 从节点 q 向上遍历到根节点，第一个出现在 HashSet 中的节点就是 LCA
    current = q;
    while (current != null) {
        if (visitedNodes.contains(current)) {
            return current; // 找到 LCA
        }
        current = current.parent;
    }

    return null; // 理论上不会执行到这里
}

// 测试用例
public static void main(String[] args) {
    Code09_LowestCommonAncestorIII solution = new Code09_LowestCommonAncestorIII();

    // 构建测试用例:
    //      3
    //     / \
    //    5   1
    //   / \   \
    //  6   2   8
    //     / \
    //    7   4
}

```

```
Node root = new Node(3);
Node node5 = new Node(5);
Node node1 = new Node(1);
Node node6 = new Node(6);
Node node2 = new Node(2);
Node node8 = new Node(8);
Node node7 = new Node(7);
Node node4 = new Node(4);

// 建立父子关系
root.left = node5;
root.right = node1;
node5.parent = root;
node1.parent = root;

node5.left = node6;
node5.right = node2;
node6.parent = node5;
node2.parent = node5;

node1.right = node8;
node8.parent = node1;

node2.left = node7;
node2.right = node4;
node7.parent = node2;
node4.parent = node2;

// 测试用例 1: p=7, q=4, LCA 应该是 2
Node result1 = solution.lowestCommonAncestor(node7, node4);
System.out.println("测试用例 1 结果: " + result1.val); // 应该输出 2

// 测试用例 2: p=6, q=8, LCA 应该是 3
Node result2 = solution.lowestCommonAncestor(node6, node8);
System.out.println("测试用例 2 结果: " + result2.val); // 应该输出 3

// 测试用例 3: p=5, q=1, LCA 应该是 3
Node result3 = solution.lowestCommonAncestor(node5, node1);
System.out.println("测试用例 3 结果: " + result3.val); // 应该输出 3
}
```

文件: Code09\_LowestCommonAncestorIII.py

```
# LeetCode 1650. Lowest Common Ancestor of a Binary Tree III
# 题目链接: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree-iii/
# 题目描述: 给定二叉树中的两个节点 p 和 q，返回它们的最低公共祖先(LCA)。
# 每个节点都有指向其父节点的引用。
#
# 解题思路:
# 1. 利用每个节点都有父指针的特点
# 2. 首先从节点 p 向上遍历到根节点，将路径上的所有节点存储在集合中
# 3. 然后从节点 q 向上遍历，第一个出现在集合中的节点就是 LCA
#
# 时间复杂度: O(h) - h 为树的高度，最坏情况下需要遍历从叶子节点到根节点的路径两次
# 空间复杂度: O(h) - 集合最多存储从 p 到根节点的 h 个节点
# 是否为最优解: 是，这是利用父指针求 LCA 的标准解法
```

class Node:

```
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None
```

```
def lowestCommonAncestor(p: 'Node', q: 'Node') -> 'Node':
```

```
    """

```

寻找二叉树中两个节点的最低公共祖先

Args:

```
    p: 第一个节点
    q: 第二个节点
```

Returns:

最低公共祖先节点

```
    """

```

```
# 创建集合存储节点 p 的所有祖先节点（包括 p 本身）
visited_ancestors = set()
```

```
# 从节点 p 向上遍历到根节点，将路径上的所有节点加入集合
```

```
current = p
while current:
    visited_ancestors.add(current)
```

```
current = current.parent

# 从节点 q 向上遍历到根节点，第一个出现在集合中的节点就是 LCA
current = q
while current:
    if current in visited_ancestors:
        return current # 找到 LCA
    current = current.parent

return None # 理论上不会执行到这里

# 测试用例
if __name__ == "__main__":
    # 构建测试用例:
    #      3
    #     / \
    #    5   1
    #   / \   \
    #  6   2   8
    #   / \
    #  7   4

    root = Node(3)
    node5 = Node(5)
    node1 = Node(1)
    node6 = Node(6)
    node2 = Node(2)
    node8 = Node(8)
    node7 = Node(7)
    node4 = Node(4)

    # 建立父子关系
    root.left = node5
    root.right = node1
    node5.parent = root
    node1.parent = root

    node5.left = node6
    node5.right = node2
    node6.parent = node5
    node2.parent = node5

    node1.right = node8
```

```

node8.parent = node1

node2.left = node7
node2.right = node4
node7.parent = node2
node4.parent = node2

# 测试用例 1: p=7, q=4, LCA 应该是 2
result1 = lowestCommonAncestor(node7, node4)
print(f"测试用例 1 结果: {result1.val}") # 应该输出 2

# 测试用例 2: p=6, q=8, LCA 应该是 3
result2 = lowestCommonAncestor(node6, node8)
print(f"测试用例 2 结果: {result2.val}") # 应该输出 3

# 测试用例 3: p=5, q=1, LCA 应该是 3
result3 = lowestCommonAncestor(node5, node1)
print(f"测试用例 3 结果: {result3.val}") # 应该输出 3

```

=====

文件: Code10\_FindLeavesOfBinaryTree.cpp

=====

```

// LeetCode 366. Find Leaves of Binary Tree
// 题目链接: https://leetcode.com/problems/find-leaves-of-binary-tree/
// 题目描述: 给你一棵二叉树, 收集树的节点, 就像这样: 收集并移除所有叶子, 重复直到树为空。
//
// 解题思路:
// 1. 使用后序遍历计算每个节点到叶子节点的高度
// 2. 叶子节点高度为 0, 父节点高度为 max(左子树高度, 右子树高度) + 1
// 3. 将相同高度的节点放在同一个列表中
// 4. 高度为 0 的节点是第一轮要删除的叶子节点, 高度为 1 的节点是第二轮要删除的叶子节点, 以此类推
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
// 是否为最优解: 是, 这是收集二叉树叶子节点的标准解法

```

```

#include <vector>
#include <algorithm>
using namespace std;

```

```

// 二叉树节点定义
struct TreeNode {

```

```

int val;
TreeNode *left;
TreeNode *right;
TreeNode() : val(0), left(nullptr), right(nullptr) {}
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * 收集二叉树的叶子节点
     *
     * @param root 二叉树的根节点
     * @return 按照删除顺序分组的节点值列表
     */
    vector<vector<int>> findLeaves(TreeNode* root) {
        vector<vector<int>> result;
        calculateHeightAndCollect(root, result);
        return result;
    }

private:
    /**
     * 计算节点高度并按高度收集节点
     *
     * @param node 当前节点
     * @param result 结果列表
     * @return 当前节点的高度（叶子节点高度为0）
     */
    int calculateHeightAndCollect(TreeNode* node, vector<vector<int>>& result) {
        // 基础情况：空节点高度为-1
        if (node == nullptr) {
            return -1;
        }

        // 递归计算左右子树的高度
        int leftHeight = calculateHeightAndCollect(node->left, result);
        int rightHeight = calculateHeightAndCollect(node->right, result);

        // 当前节点的高度为左右子树最大高度+1
        int currentHeight = max(leftHeight, rightHeight) + 1;

```

```
// 如果结果列表还没有当前高度对应的层级，则添加新列表
if (result.size() == currentHeight) {
    result.push_back(vector<int>());
}

// 将当前节点添加到对应高度的列表中
result[currentHeight].push_back(node->val);

// 返回当前节点的高度给父节点使用
return currentHeight;
}

};

// 测试用例
// int main() {
//     Solution solution;
//
//     // 测试用例 1:
//     //      1
//     //     / \
//     //    2   3
//     //    / \
//     //   4   5
//     TreeNode* root1 = new TreeNode(1);
//     root1->left = new TreeNode(2);
//     root1->right = new TreeNode(3);
//     root1->left->left = new TreeNode(4);
//     root1->left->right = new TreeNode(5);
//
//     // vector<vector<int>> result1 = solution.findLeaves(root1);
//     // 应该输出[[4, 5, 3], [2], [1]]
//
//     // 测试用例 2: 空树
//     TreeNode* root2 = nullptr;
//     vector<vector<int>> result2 = solution.findLeaves(root2);
//     // 应该输出[]
//
//     // 测试用例 3: 只有根节点
//     TreeNode* root3 = new TreeNode(1);
//     vector<vector<int>> result3 = solution.findLeaves(root3);
//     // 应该输出[[1]]
//
//     return 0;
}
```

```
// }
```

```
=====
```

文件: Code10\_FindLeavesOfBinaryTree.java

```
=====
```

```
package class037;
```

```
import java.util.*;
```

```
// LeetCode 366. Find Leaves of Binary Tree
```

```
// 题目链接: https://leetcode.com/problems/find-leaves-of-binary-tree/
```

```
// 题目描述: 给你一棵二叉树, 收集树的节点, 就像这样: 收集并移除所有叶子, 重复直到树为空。
```

```
//
```

```
// 解题思路:
```

```
// 1. 使用后序遍历计算每个节点到叶子节点的高度
```

```
// 2. 叶子节点高度为 0, 父节点高度为 max(左子树高度, 右子树高度) + 1
```

```
// 3. 将相同高度的节点放在同一个列表中
```

```
// 4. 高度为 0 的节点是第一轮要删除的叶子节点, 高度为 1 的节点是第二轮要删除的叶子节点, 以此类推
```

```
//
```

```
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
```

```
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
```

```
// 是否为最优解: 是, 这是收集二叉树叶子节点的标准解法
```

```
public class Code10_FindLeavesOfBinaryTree {
```

```
    // 二叉树节点定义
```

```
    public static class TreeNode {
```

```
        int val;
```

```
        TreeNode left;
```

```
        TreeNode right;
```

```
        TreeNode() {}
```

```
        TreeNode(int val) {
```

```
            this.val = val;
```

```
        }
```

```
        TreeNode(int val, TreeNode left, TreeNode right) {
```

```
            this.val = val;
```

```
            this.left = left;
```

```
            this.right = right;
```

```
        }
```

```
}

/**
 * 收集二叉树的叶子节点
 *
 * @param root 二叉树的根节点
 * @return 按照删除顺序分组的节点值列表
 */
public List<List<Integer>> findLeaves(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    calculateHeightAndCollect(root, result);
    return result;
}

/**
 * 计算节点高度并按高度收集节点
 *
 * @param node 当前节点
 * @param result 结果列表
 * @return 当前节点的高度（叶子节点高度为0）
 */
private int calculateHeightAndCollect(TreeNode node, List<List<Integer>> result) {
    // 基础情况：空节点高度为-1
    if (node == null) {
        return -1;
    }

    // 递归计算左右子树的高度
    int leftHeight = calculateHeightAndCollect(node.left, result);
    int rightHeight = calculateHeightAndCollect(node.right, result);

    // 当前节点的高度为左右子树最大高度+1
    int currentHeight = Math.max(leftHeight, rightHeight) + 1;

    // 如果结果列表还没有当前高度对应的层级，则添加新列表
    if (result.size() == currentHeight) {
        result.add(new ArrayList<>());
    }

    // 将当前节点添加到对应高度的列表中
    result.get(currentHeight).add(node.val);

    // 返回当前节点的高度给父节点使用
}
```

```

        return currentHeight;
    }

// 测试用例
public static void main(String[] args) {
    Code10_FindLeavesOfBinaryTree solution = new Code10_FindLeavesOfBinaryTree();

    // 测试用例 1:
    //      1
    //      / \
    //     2   3
    //     / \
    //    4   5

    TreeNode root1 = new TreeNode(1);
    root1.left = new TreeNode(2);
    root1.right = new TreeNode(3);
    root1.left.left = new TreeNode(4);
    root1.left.right = new TreeNode(5);

    List<List<Integer>> result1 = solution.findLeaves(root1);
    System.out.println("测试用例 1 结果: " + result1); // 应该输出[[4, 5], [2], [1]]

    // 测试用例 2: 空树
    TreeNode root2 = null;
    List<List<Integer>> result2 = solution.findLeaves(root2);
    System.out.println("测试用例 2 结果: " + result2); // 应该输出[]

    // 测试用例 3: 只有根节点
    TreeNode root3 = new TreeNode(1);
    List<List<Integer>> result3 = solution.findLeaves(root3);
    System.out.println("测试用例 3 结果: " + result3); // 应该输出[[1]]
}

}
=====
```

文件: Code10\_FindLeavesOfBinaryTree.py

```

# LeetCode 366. Find Leaves of Binary Tree
# 题目链接: https://leetcode.com/problems/find-leaves-of-binary-tree/
# 题目描述: 给你一棵二叉树，收集树的节点，就像这样：收集并移除所有叶子，重复直到树为空。
#
# 解题思路:
```

```
# 1. 使用后序遍历计算每个节点到叶子节点的高度
# 2. 叶子节点高度为 0, 父节点高度为 max(左子树高度, 右子树高度) + 1
# 3. 将相同高度的节点放在同一个列表中
# 4. 高度为 0 的节点是第一轮要删除的叶子节点, 高度为 1 的节点是第二轮要删除的叶子节点, 以此类推
#
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
# 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
# 是否为最优解: 是, 这是收集二叉树叶子节点的标准解法
```

```
from typing import List, Optional
```

```
# 二叉树节点定义
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
def findLeaves(root: Optional[TreeNode]) -> List[List[int]]:
```

```
"""

```

```
收集二叉树的叶子节点
```

```
Args:
```

```
    root: 二叉树的根节点
```

```
Returns:
```

```
    按照删除顺序分组的节点值列表
```

```
"""

```

```
result = []
```

```
def calculate_height_and_collect(node: Optional[TreeNode]) -> int:
```

```
"""

```

```
计算节点高度并按高度收集节点
```

```
Args:
```

```
    node: 当前节点
```

```
Returns:
```

```
    当前节点的高度 (叶子节点高度为 0)
```

```
"""

```

```
# 基础情况: 空节点高度为-1
```

```
if node is None:
```

```
    return -1
```

```
# 递归计算左右子树的高度
left_height = calculate_height_and_collect(node.left)
right_height = calculate_height_and_collect(node.right)

# 当前节点的高度为左右子树最大高度+1
current_height = max(left_height, right_height) + 1

# 如果结果列表还没有当前高度对应的层级，则添加新列表
if len(result) == current_height:
    result.append([])

# 将当前节点添加到对应高度的列表中
result[current_height].append(node.val)

# 返回当前节点的高度给父节点使用
return current_height

calculate_height_and_collect(root)
return result

# 测试用例
if __name__ == "__main__":
    # 测试用例 1:
    #      1
    #     / \
    #    2   3
    #   / \
    #  4   5
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.left = TreeNode(4)
    root1.left.right = TreeNode(5)

    result1 = findLeaves(root1)
    print(f"测试用例 1 结果: {result1}")  # 应该输出 [[4, 5, 3], [2], [1]]

    # 测试用例 2: 空树
    root2 = None
    result2 = findLeaves(root2)
    print(f"测试用例 2 结果: {result2}")  # 应该输出 []
```

```
# 测试用例 3: 只有根节点
root3 = TreeNode(1)
result3 = findLeaves(root3)
print(f"测试用例 3 结果: {result3}") # 应该输出[[1]]
```

=====

文件: Code11\_BSTToSortedDoublyLinkedList.cpp

=====

```
// LeetCode 426. Convert Binary Search Tree to Sorted Doubly Linked List
// 题目链接: https://leetcode.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/
// 题目描述: 将一个二叉搜索树就地转换为已排序的循环双向链表。
// 对于双向循环链表，左右子指针分别作为前驱和后继指针。
//
// 解题思路:
// 1. 利用 BST 的中序遍历特性，可以按升序访问所有节点
// 2. 在中序遍历过程中，维护前一个访问的节点(prev)
// 3. 将当前节点与前一个节点连接起来
// 4. 遍历完成后，将首尾节点连接形成循环链表
//
// 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是将 BST 转换为排序双向链表的标准解法
```

```
#include <iostream>
using namespace std;

// 二叉树节点定义
class Node {
public:
    int val;
    Node* left;
    Node* right;

    Node() : val(0), left(nullptr), right(nullptr) {}

    Node(int x) : val(x), left(nullptr), right(nullptr) {}

    Node(int x, Node* left, Node* right) : val(x), left(left), right(right) {}
};

class Solution {
```

```
private:
    // 用于跟踪前一个节点和头节点
    Node* prev;
    Node* head;

    /**
     * 中序遍历并构建双向链表
     *
     * @param node 当前节点
     */
    void inorderTraversal(Node* node) {
        // 基础情况：空节点
        if (node == nullptr) {
            return;
        }

        // 递归处理左子树
        inorderTraversal(node->left);

        // 处理当前节点
        if (prev == nullptr) {
            // 第一个节点，设置为头节点
            head = node;
        } else {
            // 将当前节点与前一个节点连接
            prev->right = node;
            node->left = prev;
        }

        // 更新前一个节点
        prev = node;

        // 递归处理右子树
        inorderTraversal(node->right);
    }

public:
    /**
     * 将二叉搜索树转换为排序的循环双向链表
     *
     * @param root 二叉搜索树的根节点
     * @return 排序的循环双向链表的头节点
     */

```

```
Node* treeToDoublyList(Node* root) {
    // 边界条件: 空树
    if (root == nullptr) {
        return nullptr;
    }

    // 重置全局变量
    prev = nullptr;
    head = nullptr;

    // 中序遍历构建双向链表
    inorderTraversal(root);

    // 连接首尾节点形成循环链表
    if (head != nullptr && prev != nullptr) {
        prev->right = head;
        head->left = prev;
    }

    return head;
}

// 测试用例
// int main() {
//     Solution solution;
//     //

//     // 测试用例 1:
//     // 构建 BST:
//     //      4
//     //     / \
//     //    2   5
//     //   / \
//     //  1   3
//     Node* root1 = new Node(4);
//     root1->left = new Node(2);
//     root1->right = new Node(5);
//     root1->left->left = new Node(1);
//     root1->left->right = new Node(3);
//     //

//     Node* result1 = solution.treeToDoublyList(root1);
//     //

//     // 遍历循环双向链表验证结果
```

```

//     cout << "测试用例 1 结果: ";
//     if (result1 != nullptr) {
//         Node* current = result1;
//         do {
//             cout << current->val << " ";
//             current = current->right;
//         } while (current != result1);
//     }
//     cout << endl; // 应该输出: 1 2 3 4 5
//
//     // 测试用例 2: 空树
//     Node* root2 = nullptr;
//     Node* result2 = solution.treeToDoublyList(root2);
//     cout << "测试用例 2 结果: " << result2 << endl; // 应该输出: 0 (nullptr)
//
//     // 测试用例 3: 只有根节点
//     Node* root3 = new Node(1);
//     Node* result3 = solution.treeToDoublyList(root3);
//     cout << "测试用例 3 结果: ";
//     if (result3 != nullptr) {
//         Node* current = result3;
//         do {
//             cout << current->val << " ";
//             current = current->right;
//         } while (current != result3);
//     }
//     cout << endl; // 应该输出: 1
//
//     return 0;
// }

```

---

文件: Code11\_BSTToSortedDoublyLinkedList.java

---

```

package class037;

// LeetCode 426. Convert Binary Search Tree to Sorted Doubly Linked List
// 题目链接: https://leetcode.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/
// 题目描述: 将一个二叉搜索树就地转换为已排序的循环双向链表。
// 对于双向循环链表，左右子指针分别作为前驱和后继指针。
//

```

```
// 解题思路：  
// 1. 利用 BST 的中序遍历特性，可以按升序访问所有节点  
// 2. 在中序遍历过程中，维护前一个访问的节点(prev)  
// 3. 将当前节点与前一个节点连接起来  
// 4. 遍历完成后，将首尾节点连接形成循环链表  
  
//  
// 时间复杂度：O(n) - n 为树中节点的数量，每个节点访问一次  
// 空间复杂度：O(h) - h 为树的高度，递归调用栈的深度  
// 是否为最优解：是，这是将 BST 转换为排序双向链表的标准解法
```

```
public class Code11_BSTToSortedDoublyLinkedList {
```

```
// 二叉树节点定义
```

```
static class Node {
```

```
    public int val;
```

```
    public Node left;
```

```
    public Node right;
```

```
    public Node() {}
```

```
    public Node(int val) {
```

```
        this.val = val;
```

```
    }
```

```
    public Node(int val, Node left, Node right) {
```

```
        this.val = val;
```

```
        this.left = left;
```

```
        this.right = right;
```

```
    }
```

```
}
```

```
// 全局变量用于跟踪前一个节点和头节点
```

```
private Node prev = null;
```

```
private Node head = null;
```

```
/**
```

```
* 将二叉搜索树转换为排序的循环双向链表
```

```
*
```

```
* @param root 二叉搜索树的根节点
```

```
* @return 排序的循环双向链表的头节点
```

```
*/
```

```
public Node treeToDoublyList(Node root) {
```

```
    // 边界条件：空树
```

```
if (root == null) {
    return null;
}

// 重置全局变量
prev = null;
head = null;

// 中序遍历构建双向链表
inorderTraversal(root);

// 连接首尾节点形成循环链表
if (head != null && prev != null) {
    prev.right = head;
    head.left = prev;
}

return head;
}

/***
 * 中序遍历并构建双向链表
 *
 * @param node 当前节点
 */
private void inorderTraversal(Node node) {
    // 基础情况：空节点
    if (node == null) {
        return;
    }

    // 递归处理左子树
    inorderTraversal(node.left);

    // 处理当前节点
    if (prev == null) {
        // 第一个节点，设置为头节点
        head = node;
    } else {
        // 将当前节点与前一个节点连接
        prev.right = node;
        node.left = prev;
    }
}
```

```

// 更新前一个节点
prev = node;

// 递归处理右子树
inorderTraversal(node.right);
}

// 测试用例
public static void main(String[] args) {
    Code11_BSTToSortedDoublyLinkedList solution = new Code11_BSTToSortedDoublyLinkedList();

    // 测试用例 1:
    // 构建 BST:
    //      4
    //      / \
    //     2   5
    //     / \
    //    1   3
    Node root1 = new Node(4);
    root1.left = new Node(2);
    root1.right = new Node(5);
    root1.left.left = new Node(1);
    root1.left.right = new Node(3);

    Node result1 = solution.treeToDoublyList(root1);

    // 遍历循环双向链表验证结果
    System.out.print("测试用例 1 结果: ");
    if (result1 != null) {
        Node current = result1;
        do {
            System.out.print(current.val + " ");
            current = current.right;
        } while (current != result1);
    }
    System.out.println(); // 应该输出: 1 2 3 4 5

    // 测试用例 2: 空树
    Node root2 = null;
    Node result2 = solution.treeToDoublyList(root2);
    System.out.println("测试用例 2 结果: " + result2); // 应该输出: null
}

```

```

// 测试用例 3: 只有根节点
Node root3 = new Node(1);
Node result3 = solution.treeToDoublyList(root3);
System.out.print("测试用例 3 结果: ");
if (result3 != null) {
    Node current = result3;
    do {
        System.out.print(current.val + " ");
        current = current.right;
    } while (current != result3);
}
System.out.println(); // 应该输出: 1
}
}
=====
```

文件: Code11\_BSTToSortedDoublyLinkedList.py

```

# LeetCode 426. Convert Binary Search Tree to Sorted Doubly Linked List
# 题目链接: https://leetcode.com/problems/convert-binary-search-tree-to-sorted-doubly-linked-list/
# 题目描述: 将一个二叉搜索树就地转换为已排序的循环双向链表。
# 对于双向循环链表，左右子指针分别作为前驱和后继指针。
#
# 解题思路:
# 1. 利用 BST 的中序遍历特性，可以按升序访问所有节点
# 2. 在中序遍历过程中，维护前一个访问的节点(prev)
# 3. 将当前节点与前一个节点连接起来
# 4. 遍历完成后，将首尾节点连接形成循环链表
#
# 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
# 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解: 是，这是将 BST 转换为排序双向链表的标准解法
```

```

from typing import Optional

# 二叉树节点定义
class Node:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
def treeToDoublyList(root: Optional[Node]) -> Optional[Node]:
```

```
"""
```

```
    将二叉搜索树转换为排序的循环双向链表
```

```
Args:
```

```
    root: 二叉搜索树的根节点
```

```
Returns:
```

```
    排序的循环双向链表的头节点
```

```
"""
```

```
# 边界条件: 空树
```

```
if not root:
```

```
    return None
```

```
# 用于跟踪前一个节点和头节点
```

```
prev: list[Optional[Node]] = [None] # 使用列表来模拟引用传递
```

```
head: list[Optional[Node]] = [None]
```

```
def inorder_traversal(node: Optional[Node]) -> None:
```

```
"""
```

```
    中序遍历并构建双向链表
```

```
Args:
```

```
    node: 当前节点
```

```
"""
```

```
# 基础情况: 空节点
```

```
if not node:
```

```
    return
```

```
# 递归处理左子树
```

```
inorder_traversal(node.left)
```

```
# 处理当前节点
```

```
if prev[0] is None:
```

```
    # 第一个节点, 设置为头节点
```

```
    head[0] = node
```

```
else:
```

```
    # 将当前节点与前一个节点连接
```

```
    assert prev[0] is not None
```

```
    prev[0].right = node
```

```
    node.left = prev[0]
```

```

# 更新前一个节点
prev[0] = node

# 递归处理右子树
inorder_traversal(node.right)

# 中序遍历构建双向链表
inorder_traversal(root)

# 连接首尾节点形成循环链表
if head[0] is not None and prev[0] is not None:
    prev[0].right = head[0]
    head[0].left = prev[0]

return head[0]

# 测试用例
if __name__ == "__main__":
    # 测试用例 1:
    # 构建 BST:
    #      4
    #     / \
    #    2   5
    #   / \
    #  1   3
    root1 = Node(4)
    root1.left = Node(2)
    root1.right = Node(5)
    root1.left.left = Node(1)
    root1.left.right = Node(3)

    result1 = treeToDoublyList(root1)

    # 遍历循环双向链表验证结果
    print("测试用例 1 结果: ", end="")
    if result1:
        current = result1
        while True:
            print(current.val, end=" ")
            current = current.right
            if current is not None and current == result1:
                break
    print() # 应该输出: 1 2 3 4 5

```

```

# 测试用例 2: 空树
root2 = None
result2 = treeToDoublyList(root2)
print("测试用例 2 结果:", result2) # 应该输出: None

# 测试用例 3: 只有根节点
root3 = Node(1)
result3 = treeToDoublyList(root3)
print("测试用例 3 结果: ", end="")
if result3:
    current = result3
    while True:
        print(current.val, end=" ")
        current = current.right
        if current is not None and current == result3:
            break
print() # 应该输出: 1

```

=====

文件: ConstructBinaryTree.cpp

=====

```

// LeetCode 105. Construct Binary Tree from Preorder and Inorder Traversal
// 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
// 题目描述: 给定两个整数数组 preorder 和 inorder , 其中 preorder 是二叉树的前序遍历,
// inorder 是同一棵树的中序遍历, 请构造二叉树并返回其根节点。
//
// 解题思路:
// 1. 递归构建: 前序遍历的第一个元素是根节点
// 2. 在中序遍历中找到根节点的位置, 确定左右子树的范围
// 3. 递归构建左右子树
// 4. 使用 HashMap 优化中序遍历中查找根节点位置的时间
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点处理一次
// 空间复杂度: O(n) - 存储中序遍历的索引映射, 递归栈深度 O(h)
// 是否为最优解: 是, 这是构建二叉树的标准方法

```

```

#include <unordered_map>
#include <vector>
using namespace std;

```

```

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
private:
    unordered_map<int, int> inorderIndexMap; // 存储中序遍历中值到索引的映射

    /**
     * 递归构建二叉树的辅助方法
     *
     * @param preorder 前序遍历数组
     * @param preStart 前序遍历起始索引
     * @param preEnd 前序遍历结束索引
     * @param inorder 中序遍历数组
     * @param inStart 中序遍历起始索引
     * @param inEnd 中序遍历结束索引
     * @return 构建的子树根节点
     */
    TreeNode* buildTreeHelper(vector<int>& preorder, int preStart, int preEnd,
                             vector<int>& inorder, int inStart, int inEnd) {
        // 递归终止条件：没有节点需要处理
        if (preStart > preEnd || inStart > inEnd) {
            return nullptr;
        }

        // 前序遍历的第一个元素是根节点
        int rootVal = preorder[preStart];
        TreeNode* root = new TreeNode(rootVal);

        // 在中序遍历中找到根节点的位置
        int rootIndexInInorder = inorderIndexMap[rootVal];

        // 计算左子树的大小
        int leftSubtreeSize = rootIndexInInorder - inStart;

        // 递归构建左子树
        root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize, inorder, inStart, rootIndexInInorder - 1);
        // 递归构建右子树
        root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd, inorder, rootIndexInInorder + 1, inEnd);
    }
}

```

```

// 前序遍历中左子树范围: [preStart + 1, preStart + leftSubtreeSize]
// 中序遍历中左子树范围: [inStart, rootIndexInInorder - 1]
root->left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                             inorder, inStart, rootIndexInInorder - 1);

// 递归构建右子树
// 前序遍历中右子树范围: [preStart + leftSubtreeSize + 1, preEnd]
// 中序遍历中右子树范围: [rootIndexInInorder + 1, inEnd]
root->right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                               inorder, rootIndexInInorder + 1, inEnd);

return root;
}

public:
/***
 * 根据前序遍历和中序遍历构建二叉树
 *
 * @param preorder 前序遍历数组
 * @param inorder 中序遍历数组
 * @return 构建的二叉树的根节点
 */
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    // 边界条件检查
    if (preorder.empty() || inorder.empty() || preorder.size() != inorder.size()) {
        return nullptr;
    }

    // 构建中序遍历的索引映射
    for (int i = 0; i < inorder.size(); i++) {
        inorderIndexMap[inorder[i]] = i;
    }

    return buildTreeHelper(preorder, 0, preorder.size() - 1,
                          inorder, 0, inorder.size() - 1);
}

// 测试用例
// int main() {
//     Solution solution;
//     //
//     // 测试用例 1:

```

```
//      // 前序遍历: [3, 9, 20, 15, 7]
//      // 中序遍历: [9, 3, 15, 20, 7]
//      // 构建的二叉树:
//      //          3
//      //        / \
//      //       9   20
//      //       / \
//      //      15   7
//      vector<int> preorder1 = {3, 9, 20, 15, 7};
//      vector<int> inorder1 = {9, 3, 15, 20, 7};
//
//      TreeNode* root1 = solution.buildTree(preorder1, inorder1);
//      // 验证构建结果...
//
//      // 测试用例 2: 单节点树
//      vector<int> preorder2 = {1};
//      vector<int> inorder2 = {1};
//      TreeNode* root2 = solution.buildTree(preorder2, inorder2);
//
//      // 内存清理...
//
//      return 0;
// }
```

=====

文件: ConstructBinaryTree.java

=====

```
package class037;

import java.util.HashMap;
import java.util.Map;

// LeetCode 105. Construct Binary Tree from Preorder and Inorder Traversal
// 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
// 题目描述: 给定两个整数数组 preorder 和 inorder , 其中 preorder 是二叉树的前序遍历,
// inorder 是同一棵树的中序遍历, 请构造二叉树并返回其根节点。
//
// 解题思路:
// 1. 递归构建: 前序遍历的第一个元素是根节点
// 2. 在中序遍历中找到根节点的位置, 确定左右子树的范围
// 3. 递归构建左右子树
```

```

// 4. 使用 HashMap 优化中序遍历中查找根节点位置的时间
//
// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点处理一次
// 空间复杂度: O(n) - 存储中序遍历的索引映射, 递归栈深度 O(h)
// 是否为最优解: 是, 这是构建二叉树的标准方法

// 补充题目:
// 1. LeetCode 106. Construct Binary Tree from Inorder and Postorder Traversal
// 2. LeetCode 889. Construct Binary Tree from Preorder and Postorder Traversal
// 3. 牛客 NC12. 重建二叉树 - 与 LeetCode 105 相同

public class ConstructBinaryTree {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 存储中序遍历中值到索引的映射, 用于快速查找
    private Map<Integer, Integer> inorderIndexMap;

    /**
     * 根据前序遍历和中序遍历构建二叉树
     *
     * @param preorder 前序遍历数组
     * @param inorder 中序遍历数组
     * @return 构建的二叉树的根节点
     */
    public TreeNode buildTree(int[] preorder, int[] inorder) {

```

```

// 边界条件检查
if (preorder == null || inorder == null ||
    preorder.length != inorder.length || preorder.length == 0) {
    return null;
}

// 构建中序遍历的索引映射
inorderIndexMap = new HashMap<>();
for (int i = 0; i < inorder.length; i++) {
    inorderIndexMap.put(inorder[i], i);
}

return buildTreeHelper(preorder, 0, preorder.length - 1,
                      inorder, 0, inorder.length - 1);
}

/**
 * 递归构建二叉树的辅助方法
 *
 * @param preorder 前序遍历数组
 * @param preStart 前序遍历起始索引
 * @param preEnd 前序遍历结束索引
 * @param inorder 中序遍历数组
 * @param inStart 中序遍历起始索引
 * @param inEnd 中序遍历结束索引
 * @return 构建的子树根节点
 */
private TreeNode buildTreeHelper(int[] preorder, int preStart, int preEnd,
                                 int[] inorder, int inStart, int inEnd) {
    // 递归终止条件：没有节点需要处理
    if (preStart > preEnd || inStart > inEnd) {
        return null;
    }

    // 前序遍历的第一个元素是根节点
    int rootVal = preorder[preStart];
    TreeNode root = new TreeNode(rootVal);

    // 在中序遍历中找到根节点的位置
    int rootIndexInInorder = inorderIndexMap.get(rootVal);

    // 计算左子树的大小
    int leftSubtreeSize = rootIndexInInorder - inStart;

```

```

// 递归构建左子树
// 前序遍历中左子树范围: [preStart + 1, preStart + leftSubtreeSize]
// 中序遍历中左子树范围: [inStart, rootIndexInInorder - 1]
root.left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                           inorder, inStart, rootIndexInInorder - 1);

// 递归构建右子树
// 前序遍历中右子树范围: [preStart + leftSubtreeSize + 1, preEnd]
// 中序遍历中右子树范围: [rootIndexInInorder + 1, inEnd]
root.right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                           inorder, rootIndexInInorder + 1, inEnd);

return root;
}

// 方法 2: 使用迭代方法构建 (更工程化的写法)
public TreeNode buildTreeIterative(int[] preorder, int[] inorder) {
    if (preorder == null || preorder.length == 0) {
        return null;
    }

    // 构建中序遍历的索引映射
    Map<Integer, Integer> indexMap = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        indexMap.put(inorder[i], i);
    }

    return buildTreeIterativeHelper(preorder, 0, preorder.length - 1,
                                    0, inorder.length - 1, indexMap);
}

private TreeNode buildTreeIterativeHelper(int[] preorder, int preStart, int preEnd,
   int inStart, int inEnd,
   Map<Integer, Integer> indexMap) {
    if (preStart > preEnd || inStart > inEnd) {
        return null;
    }

    int rootVal = preorder[preStart];
    TreeNode root = new TreeNode(rootVal);
    int rootIndex = indexMap.get(rootVal);
    int leftSize = rootIndex - inStart;

```

```

root.left = buildTreeIterativeHelper(preorder, preStart + 1, preStart + leftSize,
                                      inStart, rootIndex - 1, indexMap);
root.right = buildTreeIterativeHelper(preorder, preStart + leftSize + 1, preEnd,
                                       rootIndex + 1, inEnd, indexMap);

return root;
}

// 测试用例
public static void main(String[] args) {
    ConstructBinaryTree solution = new ConstructBinaryTree();

    // 测试用例 1:
    // 前序遍历: [3,9,20,15,7]
    // 中序遍历: [9,3,15,20,7]
    // 构建的二叉树:
    //      3
    //      / \
    //     9   20
    //     /   \
    //    15   7
    int[] preorder1 = {3, 9, 20, 15, 7};
    int[] inorder1 = {9, 3, 15, 20, 7};

    TreeNode root1 = solution.buildTree(preorder1, inorder1);
    System.out.println("测试用例 1 构建完成");
    printInorder(root1); // 应该输出: 9 3 15 20 7
    System.out.println();

    // 测试用例 2: 单节点树
    int[] preorder2 = {1};
    int[] inorder2 = {1};
    TreeNode root2 = solution.buildTree(preorder2, inorder2);
    System.out.println("测试用例 2 构建完成");
    printInorder(root2); // 应该输出: 1
    System.out.println();

    // 测试用例 3: 完全二叉树
    // 前序遍历: [1,2,4,5,3,6,7]
    // 中序遍历: [4,2,5,1,6,3,7]
    // 构建的二叉树:
    //      1

```

```

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

int[] preorder3 = {1, 2, 4, 5, 3, 6, 7};
int[] inorder3 = {4, 2, 5, 1, 6, 3, 7};

TreeNode root3 = solution.buildTree(preorder3, inorder3);
System.out.println("测试用例 3 构建完成");
printInorder(root3); // 应该输出: 4 2 5 1 6 3 7
System.out.println();

// 补充题目测试: LeetCode 106 - 从中序与后序遍历序列构造二叉树
System.out.println("== 补充题目测试: 从中序与后序遍历序列构造二叉树 ==");
int[] inorder4 = {9, 3, 15, 20, 7};
int[] postorder4 = {9, 15, 7, 20, 3};
TreeNode root4 = buildTreeFromInorderPostorder(inorder4, postorder4);
printInorder(root4); // 应该输出: 9 3 15 20 7
System.out.println();
}

// 辅助方法: 中序遍历打印二叉树 (用于验证)
private static void printInorder(TreeNode root) {
    if (root == null) {
        return;
    }
    printInorder(root.left);
    System.out.print(root.val + " ");
    printInorder(root.right);
}

// 补充题目: LeetCode 106. Construct Binary Tree from Inorder and Postorder Traversal
// 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-inorder-and-postorder-traversal/
public static TreeNode buildTreeFromInorderPostorder(int[] inorder, int[] postorder) {
    if (inorder == null || postorder == null ||
        inorder.length != postorder.length || inorder.length == 0) {
        return null;
    }

    // 构建中序遍历的索引映射
    Map<Integer, Integer> indexMap = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {

```

```

    indexMap.put(inorder[i], i);
}

return buildTreeHelperPostorder(inorder, 0, inorder.length - 1,
                                postorder, 0, postorder.length - 1, indexMap);
}

private static TreeNode buildTreeHelperPostorder(int[] inorder, int inStart, int inEnd,
  int[] postorder, int postStart, int postEnd,
  Map<Integer, Integer> indexMap) {
    if (inStart > inEnd || postStart > postEnd) {
        return null;
    }

    // 后序遍历的最后一个元素是根节点
    int rootVal = postorder[postEnd];
    TreeNode root = new TreeNode(rootVal);

    int rootIndex = indexMap.get(rootVal);
    int leftSize = rootIndex - inStart;

    // 递归构建左子树
    root.left = buildTreeHelperPostorder(inorder, inStart, rootIndex - 1,
   postorder, postStart, postStart + leftSize - 1,
   indexMap);

    // 递归构建右子树
    root.right = buildTreeHelperPostorder(inorder, rootIndex + 1, inEnd,
   postorder, postStart + leftSize, postEnd - 1,
   indexMap);

    return root;
}
}
=====

文件: ConstructBinaryTree.py
=====

# LeetCode 105. Construct Binary Tree from Preorder and Inorder Traversal
# 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
# 题目描述: 给定两个整数数组 preorder 和 inorder，其中 preorder 是二叉树的前序遍历，
```

```

# inorder 是同一棵树的中序遍历, 请构造二叉树并返回其根节点。
#
# 解题思路:
# 1. 递归构建: 前序遍历的第一个元素是根节点
# 2. 在中序遍历中找到根节点的位置, 确定左右子树的范围
# 3. 递归构建左右子树
# 4. 使用字典优化中序遍历中查找根节点位置的时间
#
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点处理一次
# 空间复杂度: O(n) - 存储中序遍历的索引映射, 递归栈深度 O(h)
# 是否为最优解: 是, 这是构建二叉树的标准方法

from typing import List, Optional

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        """
        根据前序遍历和中序遍历构建二叉树
        """

        Args:
            preorder: 前序遍历数组
            inorder: 中序遍历数组

        Returns:
            构建的二叉树的根节点
        """

        # 边界条件检查
        if not preorder or not inorder or len(preorder) != len(inorder):
            return None

        # 构建中序遍历的索引映射
        inorder_index_map = {val: idx for idx, val in enumerate(inorder)}

        return self._build_tree_helper(preorder, 0, len(preorder) - 1,
                                       inorder, 0, len(inorder) - 1, inorder_index_map)

```

```
def _build_tree_helper(self, preorder: List[int], pre_start: int, pre_end: int,
                      inorder: List[int], in_start: int, in_end: int,
                      index_map: dict[int, int]) -> Optional[TreeNode]:
    """
    递归构建二叉树的辅助方法

    Args:
        preorder: 前序遍历数组
        pre_start: 前序遍历起始索引
        pre_end: 前序遍历结束索引
        inorder: 中序遍历数组
        in_start: 中序遍历起始索引
        in_end: 中序遍历结束索引
        index_map: 中序遍历值到索引的映射

    Returns:
        构建的子树根节点
    """

    # 递归终止条件: 没有节点需要处理
    if pre_start > pre_end or in_start > in_end:
        return None

    # 前序遍历的第一个元素是根节点
    root_val = preorder[pre_start]
    root = TreeNode(root_val)

    # 在中序遍历中找到根节点的位置
    root_index_in_inorder = index_map[root_val]

    # 计算左子树的大小
    left_subtree_size = root_index_in_inorder - in_start

    # 递归构建左子树
    # 前序遍历中左子树范围: [pre_start + 1, pre_start + left_subtree_size]
    # 中序遍历中左子树范围: [in_start, root_index_in_inorder - 1]
    root.left = self._build_tree_helper(preorder, pre_start + 1, pre_start +
left_subtree_size,
   inorder, in_start, root_index_in_inorder - 1,
   index_map)

    # 递归构建右子树
    # 前序遍历中右子树范围: [pre_start + left_subtree_size + 1, pre_end]
    # 中序遍历中右子树范围: [root_index_in_inorder + 1, in_end]
```

```
root.right = self._build_tree_helper(preorder, pre_start + left_subtree_size + 1,
pre_end,
   inorder, root_index_in_inorder + 1, in_end, index_map)

return root

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    # 前序遍历: [3, 9, 20, 15, 7]
    # 中序遍历: [9, 3, 15, 20, 7]
    # 构建的二叉树:
    #      3
    #      / \
    #     9   20
    #     /   \
    #    15   7
    preorder1 = [3, 9, 20, 15, 7]
    inorder1 = [9, 3, 15, 20, 7]

    root1 = solution.buildTree(preorder1, inorder1)
    print("测试用例 1 构建完成")
    print_inorder(root1)  # 应该输出: 9 3 15 20 7
    print()

    # 测试用例 2: 单节点树
    preorder2 = [1]
    inorder2 = [1]
    root2 = solution.buildTree(preorder2, inorder2)
    print("测试用例 2 构建完成")
    print_inorder(root2)  # 应该输出: 1
    print()

    # 测试用例 3: 完全二叉树
    # 前序遍历: [1, 2, 4, 5, 3, 6, 7]
    # 中序遍历: [4, 2, 5, 1, 6, 3, 7]
    # 构建的二叉树:
    #      1
    #      / \
    #     2   3
    #     / \ / \
```

```

# 4 5 6 7
preorder3 = [1, 2, 4, 5, 3, 6, 7]
inorder3 = [4, 2, 5, 1, 6, 3, 7]

root3 = solution.buildTree(preorder3, inorder3)
print("测试用例 3 构建完成")
print_inorder(root3) # 应该输出: 4 2 5 1 6 3 7
print()

def print_inorder(root: Optional[TreeNode]) -> None:
    """中序遍历打印二叉树（用于验证）"""
    if root is None:
        return
    print_inorder(root.left)
    print(root.val, end=" ")
    print_inorder(root.right)

if __name__ == "__main__":
    main()

```

=====

文件: DeleteNodeInBST.cpp

=====

```

// LeetCode 450. Delete Node in a BST
// 题目链接: https://leetcode.cn/problems/delete-node-in-a-bst/
// 给定一个二叉搜索树的根节点 root 和一个值 key,
// 删除二叉搜索树中值等于 key 的节点, 保持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
//
// 解题思路:
// 1. 首先查找到要删除的节点
// 2. 找到节点后, 根据节点的子节点情况分三种情况处理:
//     - 情况 1: 节点没有子节点(叶子节点), 直接删除
//     - 情况 2: 节点只有一个子节点, 用子节点替换该节点
//     - 情况 3: 节点有两个子节点, 找到右子树中的最小节点(后继节点)替换该节点
//
// 时间复杂度: O(h) - h 为树的高度, 查找和删除操作都需要沿着树的高度进行
// 空间复杂度: O(h) - 递归调用栈的深度
// 是否为最优解: 是, 这是删除 BST 节点的标准方法

// Definition for a binary tree node.
struct TreeNode {

```

```

int val;
TreeNode *left;
TreeNode *right;
TreeNode() : val(0), left(nullptr), right(nullptr) {}
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

};

class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if (root == nullptr) {
            return nullptr;
        }

        // 递归查找要删除的节点
        if (key < root->val) {
            // key 小于当前节点值, 在左子树中查找
            root->left = deleteNode(root->left, key);
        } else if (key > root->val) {
            // key 大于当前节点值, 在右子树中查找
            root->right = deleteNode(root->right, key);
        } else {
            // 找到要删除的节点
            if (root->left == nullptr) {
                // 情况 1 和情况 2: 节点没有左子树, 直接返回右子树
                return root->right;
            } else if (root->right == nullptr) {
                // 情况 1 和情况 2: 节点没有右子树, 直接返回左子树
                return root->left;
            } else {
                // 情况 3: 节点有两个子节点
                // 找到右子树中的最小节点 (后继节点)
                TreeNode* successor = findMin(root->right);
                // 用后继节点的值替换当前节点的值
                root->val = successor->val;
                // 删除右子树中的后继节点
                root->right = deleteNode(root->right, successor->val);
            }
        }
    }

    return root;
}

```

```
private:
    // 查找以 node 为根的子树中的最小节点
    TreeNode* findMin(TreeNode* node) {
        while (node->left != nullptr) {
            node = node->left;
        }
        return node;
    }
};

// 测试代码示例（不提交）
/*
#include <iostream>
using namespace std;

int main() {
    Solution solution;

    // 构造测试用例:
    //      5
    //      / \
    //      3   6
    //      / \   \
    //      2   4   7
    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(3);
    root->right = new TreeNode(6);
    root->left->left = new TreeNode(2);
    root->left->right = new TreeNode(4);
    root->right->right = new TreeNode(7);

    // 删除节点 3（有两个子节点）
    TreeNode* result = solution.deleteNode(root, 3);
    cout << "删除节点 3 后的树根节点值: " << result->val << endl; // 应该输出 5

    // 删除节点 0（不存在的节点）
    TreeNode* result2 = solution.deleteNode(root, 0);
    cout << "删除不存在的节点 0 后的树根节点值: " << result2->val << endl; // 应该输出 5

    // 删除节点 2（叶子节点）
    TreeNode* result3 = solution.deleteNode(root, 2);
    cout << "删除节点 2 后的树根节点值: " << result3->val << endl; // 应该输出 5
```

```
    return 0;  
}  
*/  
  
=====
```

文件: DeleteNodeInBST.java

```
=====  
package class037;  
  
// LeetCode 450. Delete Node in a BST  
// 题目链接: https://leetcode.cn/problems/delete-node-in-a-bst/  
// 给定一个二叉搜索树的根节点 root 和一个值 key,  
// 删除二叉搜索树中值等于 key 的节点，保持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  
//  
// 解题思路:  
// 1. 首先查找要删除的节点  
// 2. 找到节点后，根据节点的子节点情况分三种情况处理:  
//     - 情况 1: 节点没有子节点（叶子节点），直接删除  
//     - 情况 2: 节点只有一个子节点，用子节点替换该节点  
//     - 情况 3: 节点有两个子节点，找到右子树中的最小节点（后继节点）替换该节点  
//  
// 时间复杂度: O(h) - h 为树的高度，查找和删除操作都需要沿着树的高度进行  
// 空间复杂度: O(h) - 递归调用栈的深度  
// 是否为最优解: 是，这是删除 BST 节点的标准方法
```

```
public class DeleteNodeInBST {
```

```
    // 不提交这个类  
    public static class TreeNode {  
        int val;  
        TreeNode left;  
        TreeNode right;  
  
        TreeNode() {  
        }  
  
        TreeNode(int val) {  
            this.val = val;  
        }
```

```
TreeNode(int val, TreeNode left, TreeNode right) {
    this.val = val;
    this.left = left;
    this.right = right;
}

}

// 提交如下的方法
public TreeNode deleteNode(TreeNode root, int key) {
    if (root == null) {
        return null;
    }

    // 递归查找要删除的节点
    if (key < root.val) {
        // key 小于当前节点值, 在左子树中查找
        root.left = deleteNode(root.left, key);
    } else if (key > root.val) {
        // key 大于当前节点值, 在右子树中查找
        root.right = deleteNode(root.right, key);
    } else {
        // 找到要删除的节点
        if (root.left == null) {
            // 情况 1 和情况 2: 节点没有左子树, 直接返回右子树
            return root.right;
        } else if (root.right == null) {
            // 情况 1 和情况 2: 节点没有右子树, 直接返回左子树
            return root.left;
        } else {
            // 情况 3: 节点有两个子节点
            // 找到右子树中的最小节点 (后继节点)
            TreeNode successor = findMin(root.right);
            // 用后继节点的值替换当前节点的值
            root.val = successor.val;
            // 删除右子树中的后继节点
            root.right = deleteNode(root.right, successor.val);
        }
    }
}

return root;
}

// 查找以 node 为根的子树中的最小节点
```

```

private TreeNode findMin(TreeNode node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
}

// 测试用例
public static void main(String[] args) {
    DeleteNodeInBST solution = new DeleteNodeInBST();

    // 构造测试用例:
    //      5
    //      / \
    //      3   6
    //      / \   \
    //      2   4   7
    TreeNode root = new TreeNode(5);
    root.left = new TreeNode(3);
    root.right = new TreeNode(6);
    root.left.left = new TreeNode(2);
    root.left.right = new TreeNode(4);
    root.right.right = new TreeNode(7);

    // 删除节点 3 (有两个子节点)
    TreeNode result = solution.deleteNode(root, 3);
    System.out.println("删除节点 3 后的树根节点值: " + result.val); // 应该输出 5

    // 删除节点 0 (不存在的节点)
    TreeNode result2 = solution.deleteNode(root, 0);
    System.out.println("删除不存在的节点 0 后的树根节点值: " + result2.val); // 应该输出 5

    // 删除节点 2 (叶子节点)
    TreeNode result3 = solution.deleteNode(root, 2);
    System.out.println("删除节点 2 后的树根节点值: " + result3.val); // 应该输出 5
}
}
=====
```

文件: DeleteNodeInBST.py

```
# LeetCode 450. Delete Node in a BST
```

```
# 题目链接: https://leetcode.cn/problems/delete-node-in-a-bst/
# 给定一个二叉搜索树的根节点 root 和一个值 key,
# 删除二叉搜索树中值等于 key 的节点, 保持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
#
# 解题思路:
# 1. 首先查找要删除的节点
# 2. 找到节点后, 根据节点的子节点情况分三种情况处理:
#   - 情况 1: 节点没有子节点(叶子节点), 直接删除
#   - 情况 2: 节点只有一个子节点, 用子节点替换该节点
#   - 情况 3: 节点有两个子节点, 找到右子树中的最小节点(后继节点)替换该节点
#
# 时间复杂度: O(h) - h 为树的高度, 查找和删除操作都需要沿着树的高度进行
# 空间复杂度: O(h) - 递归调用栈的深度
# 是否为最优解: 是, 这是删除 BST 节点的标准方法
```

```
from typing import Optional

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def deleteNode(root: Optional[TreeNode], key: int) -> Optional[TreeNode]:
    """
    删除二叉搜索树中值等于 key 的节点
    """

    Args:
```

```
    root: 二叉搜索树的根节点
    key: 要删除的节点值
```

```
    Returns:
```

```
    删除节点后的二叉搜索树根节点
    """

    if root is None:
        return None
```

```
# 递归查找要删除的节点
if key < root.val:
    # key 小于当前节点值, 在左子树中查找
    root.left = deleteNode(root.left, key)
```

```
        elif key > root.val:
            # key 大于当前节点值, 在右子树中查找
            root.right = deleteNode(root.right, key)
        else:
            # 找到要删除的节点
            if root.left is None:
                # 情况 1 和情况 2: 节点没有左子树, 直接返回右子树
                return root.right
            elif root.right is None:
                # 情况 1 和情况 2: 节点没有右子树, 直接返回左子树
                return root.left
            else:
                # 情况 3: 节点有两个子节点
                # 找到右子树中的最小节点(后继节点)
                successor = findMin(root.right)
                # 用后继节点的值替换当前节点的值
                root.val = successor.val
                # 删除右子树中的后继节点
                root.right = deleteNode(root.right, successor.val)

    return root
```

```
def findMin(node: TreeNode) -> TreeNode:
```

```
    """
```

```
    查找以 node 为根的子树中的最小节点
```

Args:

node: 子树的根节点

Returns:

最小节点

```
    """
```

```
    while node.left is not None:
        node = node.left
    return node
```

```
# 测试用例
```

```
if __name__ == "__main__":
    # 构造测试用例:
    #      5
    #      / \
    #      3   6
    #      / \   \
```

```

#   2   4   7
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.right = TreeNode(7)

# 删除节点 3 (有两个子节点)
result = deleteNode(root, 3)
if result:
    print(f"删除节点 3 后的树根节点值: {result.val}") # 应该输出 5

# 删除节点 0 (不存在的节点)
result2 = deleteNode(root, 0)
if result2:
    print(f"删除不存在的节点 0 后的树根节点值: {result2.val}") # 应该输出 5

# 删除节点 2 (叶子节点)
result3 = deleteNode(root, 2)
if result3:
    print(f"删除节点 2 后的树根节点值: {result3.val}") # 应该输出 5

```

=====

文件: DiameterOfBinaryTree.cpp

=====

```

// LeetCode 543. Diameter of Binary Tree
// 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/
// 题目描述: 给定一棵二叉树，你需要计算它的直径长度。
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。
//
// 解题思路:
// 1. 使用树形动态规划 (Tree DP) 的方法
// 2. 对于每个节点，直径等于左子树高度 + 右子树高度
// 3. 在递归计算高度的过程中，同时更新最大直径
// 4. 注意：直径不一定经过根节点，需要在所有节点中寻找最大值
//
// 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是计算二叉树直径的标准方法

```

```
#include <algorithm>
```

```

using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
private:
    int maxDiameter; // 全局变量，记录最大直径

    /**
     * 计算二叉树的高度，同时更新最大直径
     *
     * @param node 当前节点
     * @return 以当前节点为根的子树的高度
     */
    int height(TreeNode* node) {
        if (node == nullptr) {
            return 0;
        }

        // 递归计算左右子树的高度
        int leftHeight = height(node->left);
        int rightHeight = height(node->right);

        // 更新最大直径：当前节点的直径 = 左子树高度 + 右子树高度
        maxDiameter = max(maxDiameter, leftHeight + rightHeight);

        // 返回当前节点的高度：左右子树高度的最大值 + 1
        return max(leftHeight, rightHeight) + 1;
    }

public:
    // 提交如下的方法
    int diameterOfBinaryTree(TreeNode* root) {
        maxDiameter = 0;
        height(root);
    }
}

```

```

    return maxDiameter;
}

// 方法2：使用引用传递结果（更工程化的写法）
int diameterOfBinaryTree2(TreeNode* root) {
    int result = 0; // 存储最大直径
    height2(root, result);
    return result;
}

private:
    int height2(TreeNode* node, int& result) {
        if (node == nullptr) {
            return 0;
        }

        int leftHeight = height2(node->left, result);
        int rightHeight = height2(node->right, result);

        // 更新最大直径
        result = max(result, leftHeight + rightHeight);

        return max(leftHeight, rightHeight) + 1;
    }
};

// 测试用例
// int main() {
//     Solution solution;
//     //
//     // // 测试用例 1:
//     // //      1
//     // //      / \
//     // //      2   3
//     // //      / \
//     // //      4   5
//     // // 直径路径：4->2->1->3 或 5->2->1->3，长度为 3
//     // TreeNode* root1 = new TreeNode(1);
//     // root1->left = new TreeNode(2);
//     // root1->right = new TreeNode(3);
//     // root1->left->left = new TreeNode(4);
//     // root1->left->right = new TreeNode(5);
//     //

```

```
//     int result1 = solution.diameterOfBinaryTree(root1);
//     // 应该输出 3
//
//     // 测试用例 2: 直径不经过根节点
//     //      1
//     //      / \
//     //      2   3
//     //      / \
//     //      4   5
//     //      / \
//     //      6   7
//     // 直径路径: 6->4->2->5->7, 长度为 4
//     TreeNode* root2 = new TreeNode(1);
//     root2->left = new TreeNode(2);
//     root2->right = new TreeNode(3);
//     root2->left->left = new TreeNode(4);
//     root2->left->right = new TreeNode(5);
//     root2->left->left->left = new TreeNode(6);
//     root2->left->right->right = new TreeNode(7);
//
//     int result2 = solution.diameterOfBinaryTree(root2);
//     // 应该输出 4
//
//     // 测试用例 3: 单节点树
//     TreeNode* root3 = new TreeNode(1);
//     int result3 = solution.diameterOfBinaryTree(root3);
//     // 应该输出 0
//
//     // 测试用例 4: 空树
//     TreeNode* root4 = nullptr;
//     int result4 = solution.diameterOfBinaryTree(root4);
//     // 应该输出 0
//
//     // 测试用例 5: 退化为链表的树
//     //      1
//     //      \
//     //      2
//     //      \
//     //      3
//     // 直径路径: 1->2->3, 长度为 2
//     TreeNode* root5 = new TreeNode(1);
//     root5->right = new TreeNode(2);
//     root5->right->right = new TreeNode(3);
```

```
//  
//    int result5 = solution.diameterOfBinaryTree(root5);  
//    // 应该输出 2  
  
//  
//    // 内存清理  
//    delete root1->left->left;  
//    delete root1->left->right;  
//    delete root1->left;  
//    delete root1->right;  
//    delete root1;  
  
//  
//    delete root2->left->left->left;  
//    delete root2->left->right->right;  
//    delete root2->left->left;  
//    delete root2->left->right;  
//    delete root2->left;  
//    delete root2->right;  
//    delete root2;  
  
//  
//    delete root3;  
//    delete root5->right->right;  
//    delete root5->right;  
//    delete root5;  
  
//  
//    return 0;  
// }
```

=====

文件: DiameterOfBinaryTree.java

=====

```
package class037;  
  
// LeetCode 543. Diameter of Binary Tree  
// 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/  
// 题目描述: 给定一棵二叉树，你需要计算它的直径长度。  
// 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。  
  
// 解题思路:  
// 1. 使用树形动态规划 (Tree DP) 的方法  
// 2. 对于每个节点，直径等于左子树高度 + 右子树高度  
// 3. 在递归计算高度的过程中，同时更新最大直径  
// 4. 注意：直径不一定经过根节点，需要在所有节点中寻找最大值
```

```

//  

// 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次  

// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度  

// 是否为最优解: 是, 这是计算二叉树直径的标准方法

// 补充题目:  

// 1. LeetCode 124. Binary Tree Maximum Path Sum - 二叉树中的最大路径和  

// 2. LeetCode 687. Longest Univalue Path - 最长同值路径  

// 3. 牛客 NC6. 二叉树中的最大路径和 - 与 LeetCode 124 相同

public class DiameterOfBinaryTree {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 全局变量, 记录最大直径
    private int maxDiameter;

    // 提交如下的方法
    public int diameterOfBinaryTree(TreeNode root) {
        maxDiameter = 0;
        height(root);
        return maxDiameter;
    }

    /**
     * 计算二叉树的高度, 同时更新最大直径

```

```

*
 * @param node 当前节点
 * @return 以当前节点为根的子树的高度
 */
private int height(TreeNode node) {
    if (node == null) {
        return 0;
    }

    // 递归计算左右子树的高度
    int leftHeight = height(node.left);
    int rightHeight = height(node.right);

    // 更新最大直径: 当前节点的直径 = 左子树高度 + 右子树高度
    maxDiameter = Math.max(maxDiameter, leftHeight + rightHeight);

    // 返回当前节点的高度: 左右子树高度的最大值 + 1
    return Math.max(leftHeight, rightHeight) + 1;
}

// 方法2: 使用数组返回多个值(更工程化的写法)
public int diameterOfBinaryTree2(TreeNode root) {
    int[] result = new int[1]; // result[0]存储最大直径
    height2(root, result);
    return result[0];
}

private int height2(TreeNode node, int[] result) {
    if (node == null) {
        return 0;
    }

    int leftHeight = height2(node.left, result);
    int rightHeight = height2(node.right, result);

    // 更新最大直径
    result[0] = Math.max(result[0], leftHeight + rightHeight);

    return Math.max(leftHeight, rightHeight) + 1;
}

// 测试用例
public static void main(String[] args) {

```

```
DiameterOfBinaryTree solution = new DiameterOfBinaryTree();

// 测试用例 1:
//      1
//      / \
//     2   3
//    / \
//   4   5
// 直径路径: 4->2->1->3 或 5->2->1->3, 长度为 3
TreeNode root1 = new TreeNode(1);
root1.left = new TreeNode(2);
root1.right = new TreeNode(3);
root1.left.left = new TreeNode(4);
root1.left.right = new TreeNode(5);

int result1 = solution.diameterOfBinaryTree(root1);
System.out.println("测试用例 1 结果: " + result1); // 应该输出 3

// 测试用例 2: 直径不经过根节点
//      1
//      / \
//     2   3
//    / \
//   4   5
//  /   \
// 6     7
// 直径路径: 6->4->2->5->7, 长度为 4
TreeNode root2 = new TreeNode(1);
root2.left = new TreeNode(2);
root2.right = new TreeNode(3);
root2.left.left = new TreeNode(4);
root2.left.right = new TreeNode(5);
root2.left.left.left = new TreeNode(6);
root2.left.right.right = new TreeNode(7);

int result2 = solution.diameterOfBinaryTree(root2);
System.out.println("测试用例 2 结果: " + result2); // 应该输出 4

// 测试用例 3: 单节点树
TreeNode root3 = new TreeNode(1);
int result3 = solution.diameterOfBinaryTree(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出 0
```

```

// 测试用例 4: 空树
TreeNode root4 = null;
int result4 = solution.diameterOfBinaryTree(root4);
System.out.println("测试用例 4 结果: " + result4); // 应该输出 0

// 测试用例 5: 退化为链表的树
//      1
//        \
//        2
//          \
//          3
// 直径路径: 1->2->3, 长度为 2
TreeNode root5 = new TreeNode(1);
root5.right = new TreeNode(2);
root5.right.right = new TreeNode(3);

int result5 = solution.diameterOfBinaryTree(root5);
System.out.println("测试用例 5 结果: " + result5); // 应该输出 2

// 补充题目测试: LeetCode 687 - 最长同值路径
System.out.println("\n== 补充题目测试: 最长同值路径 ==");
//      5
//      / \
//      4   5
//      / \   \
//      1   1   5
TreeNode root6 = new TreeNode(5);
root6.left = new TreeNode(4);
root6.right = new TreeNode(5);
root6.left.left = new TreeNode(1);
root6.left.right = new TreeNode(1);
root6.right.right = new TreeNode(5);

int longestUnivalue = longestUnivaluePath(root6);
System.out.println("最长同值路径: " + longestUnivalue); // 应该输出 2
}

// 补充题目: LeetCode 687. Longest Univalue Path
// 题目链接: https://leetcode.cn/problems/longest-univalue-path/
// 计算二叉树中最长的同值路径长度
private static int maxUnivaluePath = 0;

public static int longestUnivaluePath(TreeNode root) {

```

```

maxUnivalPath = 0;
univaluePathLength(root);
return maxUnivalPath;
}

private static int univaluePathLength(TreeNode node) {
    if (node == null) {
        return 0;
    }

    int leftLength = univaluePathLength(node.left);
    int rightLength = univaluePathLength(node.right);

    // 计算左右子树的同值路径长度
    int leftUnivalue = 0, rightUnivalue = 0;

    if (node.left != null && node.left.val == node.val) {
        leftUnivalue = leftLength + 1;
    }

    if (node.right != null && node.right.val == node.val) {
        rightUnivalue = rightLength + 1;
    }

    // 更新最大同值路径长度
    maxUnivalPath = Math.max(maxUnivalPath, leftUnivalue + rightUnivalue);

    // 返回以当前节点为起点的最长同值路径长度
    return Math.max(leftUnivalue, rightUnivalue);
}
}
=====
```

文件: DiameterOfBinaryTree.py

```

=====
# LeetCode 543. Diameter of Binary Tree
# 题目链接: https://leetcode.cn/problems/diameter-of-binary-tree/
# 题目描述: 给定一棵二叉树，你需要计算它的直径长度。
# 一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。
#
# 解题思路:
# 1. 使用树形动态规划 (Tree DP) 的方法
```

```
# 2. 对于每个节点，直径等于左子树高度 + 右子树高度
# 3. 在递归计算高度的过程中，同时更新最大直径
# 4. 注意：直径不一定经过根节点，需要在所有节点中寻找最大值
#
# 时间复杂度：O(n) - n 为树中节点的数量，每个节点访问一次
# 空间复杂度：O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解：是，这是计算二叉树直径的标准方法
```

```
from typing import Optional
```

```
# 二叉树节点定义
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
```

```
    def __init__(self):
        """
        初始化 Solution 类
        """
        self.max_diameter = 0
```

```
def diameterOfBinaryTree(self, root: Optional[TreeNode]) -> int:
    """
    计算二叉树的直径长度
    
```

Args:

root: 二叉树的根节点

Returns:

二叉树的直径长度

```
"""
self.max_diameter = 0
self._height(root)
return self.max_diameter
```

```
def _height(self, node: Optional[TreeNode]) -> int:
    """
    计算二叉树的高度，同时更新最大直径
    
```

Args:

node: 当前节点

Returns:

以当前节点为根的子树的高度

"""

if node is None:

    return 0

# 递归计算左右子树的高度

left\_height = self.\_height(node.left)

right\_height = self.\_height(node.right)

# 更新最大直径: 当前节点的直径 = 左子树高度 + 右子树高度

self.max\_diameter = max(self.max\_diameter, left\_height + right\_height)

# 返回当前节点的高度: 左右子树高度的最大值 + 1

return max(left\_height, right\_height) + 1

# 方法 2: 使用非实例变量 (函数式编程风格)

def diameterOfBinaryTree2(self, root: Optional[TreeNode]) -> int:

"""

使用非实例变量计算二叉树直径

Args:

root: 二叉树的根节点

Returns:

二叉树的直径长度

"""

max\_diameter = [0] # 使用列表实现引用传递效果

def height(node: Optional[TreeNode]) -> int:

    if node is None:

        return 0

    left\_height = height(node.left)

    right\_height = height(node.right)

# 更新最大直径

    max\_diameter[0] = max(max\_diameter[0], left\_height + right\_height)

return max(left\_height, right\_height) + 1

```
height(root)
    return max_diameter[0]

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    #      1
    #      / \
    #     2   3
    #    / \
    #   4   5
    # 直径路径: 4->2->1->3 或 5->2->1->3, 长度为 3
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(3)
    root1.left.left = TreeNode(4)
    root1.left.right = TreeNode(5)

    result1 = solution.diameterOfBinaryTree(root1)
    print(f"测试用例 1 结果: {result1}") # 应该输出 3

    # 测试用例 2: 直径不经过根节点
    #      1
    #      / \
    #     2   3
    #    / \
    #   4   5
    #  /   \
    # 6     7
    # 直径路径: 6->4->2->5->7, 长度为 4
    root2 = TreeNode(1)
    root2.left = TreeNode(2)
    root2.right = TreeNode(3)
    root2.left.left = TreeNode(4)
    root2.left.right = TreeNode(5)
    root2.left.left.left = TreeNode(6)
    root2.left.right.right = TreeNode(7)

    result2 = solution.diameterOfBinaryTree(root2)
    print(f"测试用例 2 结果: {result2}") # 应该输出 4
```

```

# 测试用例 3: 单节点树
root3 = TreeNode(1)
result3 = solution.diameterOfBinaryTree(root3)
print(f"测试用例 3 结果: {result3}") # 应该输出 0

# 测试用例 4: 空树
root4 = None
result4 = solution.diameterOfBinaryTree(root4)
print(f"测试用例 4 结果: {result4}") # 应该输出 0

# 测试用例 5: 退化为链表的树
#      1
#        \
#      2
#        \
#      3
# 直径路径: 1->2->3, 长度为 2
root5 = TreeNode(1)
root5.right = TreeNode(2)
root5.right.right = TreeNode(3)

result5 = solution.diameterOfBinaryTree(root5)
print(f"测试用例 5 结果: {result5}") # 应该输出 2

# 补充题目测试: LeetCode 687 - 最长同值路径
print("\n==== 补充题目测试: 最长同值路径 ====")
#      5
#    / \
#   4   5
#  / \   \
# 1   1   5
root6 = TreeNode(5)
root6.left = TreeNode(4)
root6.right = TreeNode(5)
root6.left.left = TreeNode(1)
root6.left.right = TreeNode(1)
root6.right.right = TreeNode(5)

def longest_univalue_path(root: Optional[TreeNode]) -> int:
    """
    计算二叉树中最长的同值路径长度
    """

Args:

```

root: 二叉树的根节点

Returns:

最长同值路径长度

"""

max\_length = [0] # 使用列表实现引用传递

def path\_length(node: Optional[TreeNode]) -> int:

if node is None:

return 0

left\_length = path\_length(node.left)

right\_length = path\_length(node.right)

# 计算左右子树的同值路径长度

left\_univalue = 0

right\_univalue = 0

if node.left and node.left.val == node.val:

left\_univalue = left\_length + 1

if node.right and node.right.val == node.val:

right\_univalue = right\_length + 1

# 更新最大同值路径长度

max\_length[0] = max(max\_length[0], left\_univalue + right\_univalue)

# 返回以当前节点为起点的最长同值路径长度

return max(left\_univalue, right\_univalue)

path\_length(root)

return max\_length[0]

longest\_result = longest\_univalue\_path(root6)

print(f"最长同值路径: {longest\_result}") # 应该输出 2

if \_\_name\_\_ == "\_\_main\_\_":

main()

=====

文件: FlattenBinaryTree.cpp

=====

```
// LeetCode 114. Flatten Binary Tree to Linked List
// 题目链接: https://leetcode.cn/problems/flatten-binary-tree-to-linked-list/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 后序遍历, 先处理左右子树, 再将左子树插入到根节点和右子树之间
// 2. 迭代方法: 使用前序遍历, 将节点按顺序连接
// 3. Morris 遍历: O(1) 空间复杂度的解法
//
// 时间复杂度: O(n) - n 为树中节点的数量
// 空间复杂度:
// - 递归: O(h) - h 为树的高度
// - 迭代: O(n) - 需要存储前序遍历结果
// - Morris: O(1) - 只使用常数空间
// 是否为最优解: Morris 遍历是最优解, 空间复杂度 O(1)
```

```
#include <stack>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    // 方法 1: 递归解法 (后序遍历)
    // 核心思想: 先递归处理左右子树, 然后将左子树插入到根节点和右子树之间
    void flattenRecursive(TreeNode* root) {
        if (root == nullptr) {
            return;
        }

        // 递归处理左右子树
        flattenRecursive(root->left);
        flattenRecursive(root->right);

        // 将左子树插入到根节点和右子树之间
        TreeNode* left = root->left;
        root->left = nullptr;
        root->right = left;
    }
}
```

```
flattenRecursive(root->right);

// 保存右子树
TreeNode* right = root->right;

// 将左子树移到右子树位置
root->right = root->left;
root->left = nullptr;

// 找到当前右子树（原左子树）的最后一个节点
TreeNode* current = root;
while (current->right != nullptr) {
    current = current->right;
}

// 将原右子树接到末尾
current->right = right;
}

// 方法 2: 迭代解法（前序遍历）
// 核心思想: 使用栈进行前序遍历, 按顺序连接节点
void flattenIterative(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    stack<TreeNode*> stk;
    stk.push(root);
    TreeNode* prev = nullptr;

    while (!stk.empty()) {
        TreeNode* current = stk.top();
        stk.pop();

        if (prev != nullptr) {
            prev->right = current;
            prev->left = nullptr;
        }

        // 先右后左入栈, 保证出栈顺序是前序遍历
        if (current->right != nullptr) {
            stk.push(current->right);
        }
    }
}
```

```

    if (current->left != nullptr) {
        stk.push(current->left);
    }

    prev = current;
}
}

// 方法 3: Morris 遍历 (最优解, O(1)空间)
// 核心思想: 利用线索二叉树的思想, 在遍历过程中建立连接
void flattenMorris(TreeNode* root) {
    TreeNode* current = root;

    while (current != nullptr) {
        if (current->left != nullptr) {
            // 找到当前节点左子树的最右节点 (前驱节点)
            TreeNode* predecessor = current->left;
            while (predecessor->right != nullptr) {
                predecessor = predecessor->right;
            }

            // 将当前节点的右子树接到前驱节点的右边
            predecessor->right = current->right;
            // 将左子树移到右子树位置
            current->right = current->left;
            current->left = nullptr;
        }

        // 移动到下一个节点
        current = current->right;
    }
}

// 提交如下的方法 (使用 Morris 遍历, 最优解)
void flatten(TreeNode* root) {
    flattenMorris(root);
}
};

// 测试用例
// int main() {
//     Solution solution;
// }

```

```
// // 测试用例 1:  
// //      1  
// //      / \  
// //      2   5  
// //      / \   \  
// //      3   4   6  
// // 展开后: 1->2->3->4->5->6  
// TreeNode* root1 = new TreeNode(1);  
// root1->left = new TreeNode(2);  
// root1->right = new TreeNode(5);  
// root1->left->left = new TreeNode(3);  
// root1->left->right = new TreeNode(4);  
// root1->right->right = new TreeNode(6);  
  
//  
// solution.flatten(root1);  
  
//  
// // 验证展开结果  
// TreeNode* current = root1;  
// while (current != nullptr) {  
//     cout << current->val << " ";  
//     current = current->right;  
// }  
// cout << endl; // 应该输出: 1 2 3 4 5 6  
  
//  
// // 测试用例 2: 空树  
// TreeNode* root2 = nullptr;  
// solution.flatten(root2);  
  
//  
// // 测试用例 3: 只有左子树的树  
// //      1  
// //      /  
// //      2  
// //      /  
// //      3  
// TreeNode* root3 = new TreeNode(1);  
// root3->left = new TreeNode(2);  
// root3->left->left = new TreeNode(3);  
  
//  
// solution.flatten(root3);  
// current = root3;  
// while (current != nullptr) {  
//     cout << current->val << " ";  
//     current = current->right;
```

```
// }
// cout << endl; // 应该输出: 1 2 3
//
// // 内存清理
// delete root1->left->left;
// delete root1->left->right;
// delete root1->right->right;
// delete root1->left;
// delete root1->right;
// delete root1;
//
// delete root3->left->left;
// delete root3->left;
// delete root3;
//
// return 0;
// }
```

=====

文件: FlattenBinaryTree.java

=====

```
package class037;

// LeetCode 114. Flatten Binary Tree to Linked List
// 题目链接: https://leetcode.cn/problems/flatten-binary-tree-to-linked-list/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 后序遍历, 先处理左右子树, 再将左子树插入到根节点和右子树之间
// 2. 迭代方法: 使用前序遍历, 将节点按顺序连接
// 3. Morris 遍历: O(1) 空间复杂度的解法
//
// 时间复杂度: O(n) - n 为树中节点的数量
// 空间复杂度:
// - 递归: O(h) - h 为树的高度
// - 迭代: O(n) - 需要存储前序遍历结果
// - Morris: O(1) - 只使用常数空间
// 是否为最优解: Morris 遍历是最优解, 空间复杂度 O(1)
```

```
// 补充题目：  
// 1. LeetCode 144. Binary Tree Preorder Traversal - 二叉树的前序遍历  
// 2. LeetCode 94. Binary Tree Inorder Traversal - 二叉树的中序遍历  
// 3. 牛客 NC5. 二叉树根节点到叶子节点的所有路径和  
  
public class FlattenBinaryTree {  
  
    // 二叉树节点定义  
    public static class TreeNode {  
        int val;  
        TreeNode left;  
        TreeNode right;  
  
        TreeNode() {}  
  
        TreeNode(int val) {  
            this.val = val;  
        }  
  
        TreeNode(int val, TreeNode left, TreeNode right) {  
            this.val = val;  
            this.left = left;  
            this.right = right;  
        }  
    }  
  
    // 方法1：递归解法（后序遍历）  
    // 核心思想：先递归处理左右子树，然后将左子树插入到根节点和右子树之间  
    public void flattenRecursive(TreeNode root) {  
        if (root == null) {  
            return;  
        }  
  
        // 递归处理左右子树  
        flattenRecursive(root.left);  
        flattenRecursive(root.right);  
  
        // 保存右子树  
        TreeNode right = root.right;  
  
        // 将左子树移到右子树位置  
        root.right = root.left;  
        root.left = null;  
    }  
}
```

```

// 找到当前右子树（原左子树）的最后一个节点
TreeNode current = root;
while (current.right != null) {
    current = current.right;
}

// 将原右子树接到末尾
current.right = right;
}

// 方法 2：迭代解法（前序遍历）
// 核心思想：使用栈进行前序遍历，按顺序连接节点
public void flattenIterative(TreeNode root) {
    if (root == null) {
        return;
    }

    java.util.Stack<TreeNode> stack = new java.util.Stack<>();
    stack.push(root);
    TreeNode prev = null;

    while (!stack.isEmpty()) {
        TreeNode current = stack.pop();

        if (prev != null) {
            prev.right = current;
            prev.left = null;
        }

        // 先右后左入栈，保证出栈顺序是前序遍历
        if (current.right != null) {
            stack.push(current.right);
        }
        if (current.left != null) {
            stack.push(current.left);
        }

        prev = current;
    }
}

// 方法 3：Morris 遍历（最优解，O(1) 空间）

```

```

// 核心思想：利用线索二叉树的思想，在遍历过程中建立连接
public void flattenMorris(TreeNode root) {
    TreeNode current = root;

    while (current != null) {
        if (current.left != null) {
            // 找到当前节点左子树的最右节点（前驱节点）
            TreeNode predecessor = current.left;
            while (predecessor.right != null) {
                predecessor = predecessor.right;
            }

            // 将当前节点的右子树接到前驱节点的右边
            predecessor.right = current.right;
            // 将左子树移到右子树位置
            current.right = current.left;
            current.left = null;
        }

        // 移动到下一个节点
        current = current.right;
    }
}

// 提交如下的方法（使用 Morris 遍历，最优解）
public void flatten(TreeNode root) {
    flattenMorris(root);
}

// 辅助方法：打印展开后的链表（用于测试）
public void printFlattenedTree(TreeNode root) {
    TreeNode current = root;
    while (current != null) {
        System.out.print(current.val + " ");
        current = current.right;
    }
    System.out.println();
}

// 测试用例
public static void main(String[] args) {
    FlattenBinaryTree solution = new FlattenBinaryTree();
}

```

```
// 测试用例 1:  
//      1  
//      / \  
//      2   5  
//      / \   \  
//      3   4   6  
// 展开后: 1->2->3->4->5->6  
TreeNode root1 = new TreeNode(1);  
root1.left = new TreeNode(2);  
root1.right = new TreeNode(5);  
root1.left.left = new TreeNode(3);  
root1.left.right = new TreeNode(4);  
root1.right.right = new TreeNode(6);  
  
System.out.println("原始树前序遍历:");  
printPreorder(root1); // 应该输出: 1 2 3 4 5 6  
  
solution.flatten(root1);  
System.out.println("展开后链表:");  
solution.printFlattenedTree(root1); // 应该输出: 1 2 3 4 5 6  
  
// 测试用例 2: 空树  
TreeNode root2 = null;  
solution.flatten(root2);  
System.out.println("空树展开结果:");  
solution.printFlattenedTree(root2); // 应该输出空行  
  
// 测试用例 3: 只有左子树的树  
//      1  
//      /  
//      2  
//      /  
//      3  
TreeNode root3 = new TreeNode(1);  
root3.left = new TreeNode(2);  
root3.left.left = new TreeNode(3);  
  
solution.flatten(root3);  
System.out.println("只有左子树展开结果:");  
solution.printFlattenedTree(root3); // 应该输出: 1 2 3  
  
// 补充题目测试: 二叉树的前序遍历  
System.out.println("\n==== 补充题目测试: 二叉树的前序遍历 ====");
```

```

TreeNode root4 = new TreeNode(1);
root4.left = new TreeNode(2);
root4.right = new TreeNode(3);
root4.left.left = new TreeNode(4);
root4.left.right = new TreeNode(5);

java.util.List<Integer> preorder = preorderTraversal(root4);
System.out.println("前序遍历结果: " + preorder); // 应该输出: [1, 2, 4, 5, 3]
}

// 辅助方法: 前序遍历 (递归)
private static void printPreorder(TreeNode root) {
    if (root == null) {
        return;
    }
    System.out.print(root.val + " ");
    printPreorder(root.left);
    printPreorder(root.right);
}

// 补充题目: LeetCode 144. Binary Tree Preorder Traversal
// 题目链接: https://leetcode.cn/problems/binary-tree-preorder-traversal/
public static java.util.List<Integer> preorderTraversal(TreeNode root) {
    java.util.List<Integer> result = new java.util.ArrayList<>();
    if (root == null) {
        return result;
    }

    java.util.Stack<TreeNode> stack = new java.util.Stack<>();
    stack.push(root);

    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();
        result.add(node.val);

        // 先右后左入栈, 保证出栈顺序是根->左->右
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }
}

```

```
        return result;
    }
}
```

文件: FlattenBinaryTree.py

```
# LeetCode 114. Flatten Binary Tree to Linked List
# 题目链接: https://leetcode.cn/problems/flatten-binary-tree-to-linked-list/
# 题目描述: 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。
#
# 解题思路:
# 1. 递归方法: 后序遍历, 先处理左右子树, 再将左子树插入到根节点和右子树之间
# 2. 迭代方法: 使用前序遍历, 将节点按顺序连接
# 3. Morris 遍历: O(1) 空间复杂度的解法
#
# 时间复杂度: O(n) - n 为树中节点的数量
# 空间复杂度:
# - 递归: O(h) - h 为树的高度
# - 迭代: O(n) - 需要存储前序遍历结果
# - Morris: O(1) - 只使用常数空间
# 是否为最优解: Morris 遍历是最优解, 空间复杂度 O(1)
```

```
from typing import Optional
```

```
# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def flattenRecursive(self, root: Optional[TreeNode]) -> None:
        """
        递归方法展开二叉树为链表

```

Args:

```
root: 二叉树的根节点
"""
if root is None:
    return

# 递归处理左右子树
self.flattenRecursive(root.left)
self.flattenRecursive(root.right)

# 保存右子树
right = root.right

# 将左子树移到右子树位置
root.right = root.left
root.left = None

# 找到当前右子树（原左子树）的最后一个节点
current = root
while current.right is not None:
    current = current.right

# 将原右子树接到末尾
current.right = right

def flattenIterative(self, root: Optional[TreeNode]) -> None:
"""
迭代方法展开二叉树为链表

```

Args:

root: 二叉树的根节点

"""
if root is None:
 return

stack = [root]
prev = None

while stack:
 current = stack.pop()

 if prev is not None:
 prev.right = current
 prev.left = None

```
# 先右后左入栈，保证出栈顺序是前序遍历
if current.right is not None:
    stack.append(current.right)
if current.left is not None:
    stack.append(current.left)

prev = current
```

```
def flattenMorris(self, root: Optional[TreeNode]) -> None:
    """
```

Morris 遍历方法展开二叉树为链表（最优解）

Args:

root: 二叉树的根节点

"""

```
current = root
```

```
while current is not None:
```

```
    if current.left is not None:
```

# 找到当前节点左子树的最右节点（前驱节点）

```
    predecessor = current.left
```

```
    while predecessor.right is not None:
```

```
        predecessor = predecessor.right
```

# 将当前节点的右子树接到前驱节点的右边

```
    predecessor.right = current.right
```

# 将左子树移到右子树位置

```
    current.right = current.left
```

```
    current.left = None
```

# 移动到下一个节点

```
    current = current.right
```

```
def flatten(self, root: Optional[TreeNode]) -> None:
    """
```

展开二叉树为链表（使用 Morris 遍历，最优解）

Args:

root: 二叉树的根节点

"""

```
self.flattenMorris(root)
```

```
# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    #      1
    #      / \
    #      2   5
    #      / \   \
    #      3   4   6
    # 展开后: 1->2->3->4->5->6
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(5)
    root1.left.left = TreeNode(3)
    root1.left.right = TreeNode(4)
    root1.right.right = TreeNode(6)

    print("原始树结构:")
    print_tree(root1)

    solution.flatten(root1)
    print("展开后链表:")
    print_flattened(root1)  # 应该输出: 1 2 3 4 5 6

    # 测试用例 2: 空树
    root2 = None
    solution.flatten(root2)
    print("空树展开结果:")
    print_flattened(root2)  # 应该输出空行

    # 测试用例 3: 只有左子树的树
    #      1
    #      /
    #      2
    #      /
    #      3
    root3 = TreeNode(1)
    root3.left = TreeNode(2)
    root3.left.left = TreeNode(3)

    solution.flatten(root3)
    print("只有左子树展开结果:")
```

```

print_flattened(root3) # 应该输出: 1 2 3

def print_tree(root: Optional[TreeNode]) -> None:
    """打印二叉树（前序遍历）"""
    if root is None:
        return
    print(root.val, end=" ")
    print_tree(root.left)
    print_tree(root.right)

def print_flattened(root: Optional[TreeNode]) -> None:
    """打印展开后的链表"""
    current = root
    while current is not None:
        print(current.val, end=" ")
        current = current.right
    print()

if __name__ == "__main__":
    main()

```

=====

文件: InvertBinaryTree.java

=====

```

package class037;

// LeetCode 226. Invert Binary Tree
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目描述: 给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。
// 翻转二叉树就是将每个节点的左右子树进行交换。
//
// 解题思路:
// 1. 使用递归深度优先搜索(DFS)
// 2. 对于每个节点，先递归翻转其左右子树
// 3. 然后交换当前节点的左右子树
// 4. 返回翻转后的根节点
//
// 时间复杂度: O(n) - n 为树中节点的数量，需要访问每个节点
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是翻转二叉树的标准方法

```

```
public class InvertBinaryTree {
```

```
// 二叉树节点定义
public static class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode() {}

    TreeNode(int val) {
        this.val = val;
    }

    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

// 提交如下的方法
public TreeNode invertTree(TreeNode root) {
    // 基本情况：空节点直接返回
    if (root == null) {
        return null;
    }

    // 递归翻转左右子树
    TreeNode left = invertTree(root.left);
    TreeNode right = invertTree(root.right);

    // 交换左右子树
    root.left = right;
    root.right = left;

    return root;
}

// 测试用例
public static void main(String[] args) {
    InvertBinaryTree solution = new InvertBinaryTree();

    // 测试用例 1:
}
```

```

// 原始树:
//      4
//      / \
//     2   7
//    / \ / \
//   1  3 6  9
//
// 翻转后:
//      4
//      / \
//     7   2
//    / \ / \
//   9  6 3  1

TreeNode root1 = new TreeNode(4);
root1.left = new TreeNode(2);
root1.right = new TreeNode(7);
root1.left.left = new TreeNode(1);
root1.left.right = new TreeNode(3);
root1.right.left = new TreeNode(6);
root1.right.right = new TreeNode(9);

System.out.println("翻转前:");
printTree(root1);

TreeNode invertedRoot1 = solution.invertTree(root1);
System.out.println("翻转后:");
printTree(invertedRoot1);
}

```

```

// 辅助方法: 打印二叉树 (前序遍历)
private static void printTree(TreeNode root) {
    if (root == null) {
        System.out.print("null ");
        return;
    }
    System.out.print(root.val + " ");
    printTree(root.left);
    printTree(root.right);
}
}

=====

```

文件: InvertBinaryTree.py

```
=====
# LeetCode 226. Invert Binary Tree
# 题目链接: https://leetcode.cn/problems/invert-binary-tree/
# 题目描述: 给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。
# 翻转二叉树就是将每个节点的左右子树进行交换。
#
# 解题思路:
# 1. 使用递归深度优先搜索(DFS)
# 2. 对于每个节点，先递归翻转其左右子树
# 3. 然后交换当前节点的左右子树
# 4. 返回翻转后的根节点
#
# 时间复杂度: O(n) - n 为树中节点的数量，需要访问每个节点
# 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解: 是，这是翻转二叉树的标准方法
```

```
from typing import Optional
```

```
# 二叉树节点定义
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
class Solution:
```

```
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
```

```
        """

```

```
        翻转二叉树
    
```

```
Args:
```

```
    root: 二叉树的根节点
```

```
Returns:
```

```
    翻转后的根节点
"""

```

```
# 基本情况: 空节点直接返回
```

```
if root is None:
```

```
    return None
```

```
# 递归翻转左右子树
```

```
left = self.invertTree(root.left)
```

```
        right = self.invertTree(root.right)

        # 交换左右子树
        root.left = right
        root.right = left

    return root

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1:
    # 原始树:
    #      4
    #     / \
    #    2   7
    #   / \ / \
    #  1  3 6  9
    #
    # 翻转后:
    #      4
    #     / \
    #    7   2
    #   / \ / \
    #  9  6 3  1

    root1 = TreeNode(4)
    root1.left = TreeNode(2)
    root1.right = TreeNode(7)
    root1.left.left = TreeNode(1)
    root1.left.right = TreeNode(3)
    root1.right.left = TreeNode(6)
    root1.right.right = TreeNode(9)

    print("翻转前:")
    print_tree(root1)

    inverted_root1 = solution.invertTree(root1)
    print("\n翻转后:")
    print_tree(inverted_root1)

# 辅助方法: 打印二叉树 (前序遍历)
def print_tree(root: Optional[TreeNode]) -> None:
```

```
if root is None:  
    print("null", end=" ")  
    return  
print(root.val, end=" ")  
print_tree(root.left)  
print_tree(root.right)
```

```
if __name__ == "__main__":  
    main()
```

=====

文件: LowestCommonAncestor.java

=====

```
package class037;  
  
// LeetCode 236. Lowest Common Ancestor of a Binary Tree  
// 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/  
// 题目描述: 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。  
// 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。"  
//  
// 解题思路:  
// 1. 使用递归深度优先搜索(DFS)  
// 2. 对于每个节点, 递归检查其左右子树  
// 3. 如果当前节点是 p 或 q, 则直接返回当前节点  
// 4. 如果左右子树分别找到了 p 和 q, 则当前节点就是 LCA  
// 5. 如果只在一侧子树找到了 p 或 q, 则返回找到的节点  
//  
// 时间复杂度: O(n) - n 为树中节点的数量, 最坏情况下需要遍历所有节点  
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度  
// 是否为最优解: 是, 这是寻找 LCA 的标准方法
```

```
public class LowestCommonAncestor {
```

```
// 二叉树节点定义  
public static class TreeNode {  
    int val;  
    TreeNode left;  
    TreeNode right;
```

```
    TreeNode(int x) {  
        val = x;
```

```
}

}

// 提交如下的方法
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // 基本情况：空节点或找到 p 或 q
    if (root == null || root == p || root == q) {
        return root;
    }

    // 递归在左右子树中查找 p 和 q
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    // 如果左右子树都找到了，说明当前节点是 LCA
    if (left != null && right != null) {
        return root;
    }

    // 如果只在一侧找到了，返回找到的节点
    return left != null ? left : right;
}

// 测试用例
public static void main(String[] args) {
    LowestCommonAncestor solution = new LowestCommonAncestor();

    // 构造测试用例:
    //      3
    //      / \
    //      5   1
    //      / \ / \
    //      6  2 0  8
    //      / \
    //      7  4

    TreeNode root = new TreeNode(3);
    root.left = new TreeNode(5);
    root.right = new TreeNode(1);
    root.left.left = new TreeNode(6);
    root.left.right = new TreeNode(2);
    root.right.left = new TreeNode(0);
    root.right.right = new TreeNode(8);
    root.left.right.left = new TreeNode(7);
}
```

```

root.left.right.right = new TreeNode(4);

TreeNode p1 = root.left; // 节点 5
TreeNode q1 = root.right; // 节点 1
TreeNode result1 = solution.lowestCommonAncestor(root, p1, q1);
System.out.println("LCA(5, 1) = " + result1.val); // 应该输出 3

TreeNode p2 = root.left; // 节点 5
TreeNode q2 = root.left.right.right; // 节点 4
TreeNode result2 = solution.lowestCommonAncestor(root, p2, q2);
System.out.println("LCA(5, 4) = " + result2.val); // 应该输出 5
}
}

```

=====

文件: LowestCommonAncestor.py

=====

```

# LeetCode 236. Lowest Common Ancestor of a Binary Tree
# 题目链接: https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/
# 题目描述: 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。
# 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。”
#
# 解题思路:
# 1. 使用递归深度优先搜索(DFS)
# 2. 对于每个节点, 递归检查其左右子树
# 3. 如果当前节点是 p 或 q, 则直接返回当前节点
# 4. 如果左右子树分别找到了 p 和 q, 则当前节点就是 LCA
# 5. 如果只在一侧子树找到了 p 或 q, 则返回找到的节点
#
# 时间复杂度: O(n) - n 为树中节点的数量, 最坏情况下需要遍历所有节点
# 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
# 是否为最优解: 是, 这是寻找 LCA 的标准方法

```

```

from typing import Optional

# 二叉树节点定义
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left: Optional['TreeNode'] = None
        self.right: Optional['TreeNode'] = None

```

```
class Solution:
    def lowestCommonAncestor(self, root: Optional[TreeNode], p: TreeNode, q: TreeNode) ->
Optional[TreeNode]:
    """
    找到二叉树中两个指定节点的最近公共祖先
    """

    Args:
        root: 二叉树的根节点
        p: 目标节点 p
        q: 目标节点 q

    Returns:
        最近公共祖先节点
    """

    # 基本情况: 空节点或找到 p 或 q
    if root is None or root == p or root == q:
        return root

    # 递归在左右子树中查找 p 和 q
    left = self.lowestCommonAncestor(root.left, p, q)
    right = self.lowestCommonAncestor(root.right, p, q)

    # 如果左右子树都找到了, 说明当前节点是 LCA
    if left is not None and right is not None:
        return root

    # 如果只在一侧找到了, 返回找到的节点
    return left if left is not None else right

# 测试用例
def main():
    solution = Solution()

    # 构造测试用例:
    #      3
    #     / \
    #    5   1
    #   / \ / \
    #  6 2 0 8
    #  / \
    # 7 4
    root = TreeNode(3)
```

```

root.left = TreeNode(5)
root.right = TreeNode(1)
root.left.left = TreeNode(6)
root.left.right = TreeNode(2)
root.right.left = TreeNode(0)
root.right.right = TreeNode(8)
root.left.right.left = TreeNode(7)
root.left.right.right = TreeNode(4)

p1 = root.left # 节点 5
q1 = root.right # 节点 1
result1 = solution.lowestCommonAncestor(root, p1, q1)
if result1 is not None:
    print("LCA(5, 1) =", result1.val) # 应该输出 3

p2 = root.left      # 节点 5
q2 = root.left.right.right # 节点 4
result2 = solution.lowestCommonAncestor(root, p2, q2)
if result2 is not None:
    print("LCA(5, 4) =", result2.val) # 应该输出 5

if __name__ == "__main__":
    main()

```

---

文件: LowestCommonAncestorII.cpp

---

```

// LintCode 474. Lowest Common Ancestor II
// 题目链接: https://www.lintcode.com/problem/474/
// 给定一个二叉树中的节点 a 和 b, 找出他们的最近公共祖先
// 每个节点包含一个指向其父节点的指针
//
// 解题思路:
// 1. 由于每个节点都有父节点指针, 可以将问题转化为链表相交问题
// 2. 从节点 a 开始, 沿着父节点指针向上遍历, 记录所有访问过的节点
// 3. 从节点 b 开始, 沿着父节点指针向上遍历, 第一个已经在步骤 2 中访问过的节点就是 LCA
//
// 时间复杂度: O(h) - h 为树的高度
// 空间复杂度: O(h) - 需要存储从 a 到根节点的路径
// 是否为最优解: 是, 利用父节点指针可以避免遍历整棵树

// 不提交这个类

```

```

struct ParentTreeNode {
    int val;
    ParentTreeNode *parent, *left, *right;
};

// 提交如下的方法
ParentTreeNode * lowestCommonAncestorII(ParentTreeNode * root, ParentTreeNode * A, ParentTreeNode * B) {
    // 使用数组记录从节点 A 到根节点的路径（假设最多 1000 个节点）
    ParentTreeNode* path[1000];
    int pathLen = 0;

    // 从节点 A 开始向上遍历到根节点，记录路径上的所有节点
    ParentTreeNode* current = A;
    while (current != 0) { // 用 0 代替 NULL
        path[pathLen++] = current;
        current = current->parent;
    }

    // 从节点 B 开始向上遍历，第一个在 path 数组中的节点就是 LCA
    current = B;
    while (current != 0) { // 用 0 代替 NULL
        // 检查 current 是否在 path 数组中
        for (int i = 0; i < pathLen; i++) {
            if (path[i] == current) {
                return current; // 找到 LCA
            }
        }
        current = current->parent;
    }

    // 理论上不会执行到这里，因为 A 和 B 都在树中，肯定有公共祖先
    return 0; // 用 0 代替 NULL
}

```

```

// 测试代码示例（不提交）
/*
int main() {
    // 构造测试用例:
    //      4
    //      / \
    //      3   7
    //      / \

```

```
//      5   6
ParentTreeNode* node4 = new ParentTreeNode();
ParentTreeNode* node3 = new ParentTreeNode();
ParentTreeNode* node7 = new ParentTreeNode();
ParentTreeNode* node5 = new ParentTreeNode();
ParentTreeNode* node6 = new ParentTreeNode();

node4->val = 4;
node3->val = 3;
node7->val = 7;
node5->val = 5;
node6->val = 6;

// 设置父子关系
node3->parent = node4;
node7->parent = node4;
node5->parent = node7;
node6->parent = node7;

node4->left = node3;
node4->right = node7;
node7->left = node5;
node7->right = node6;

// 测试 LCA(3, 5) = 4
ParentTreeNode* result1 = lowestCommonAncestorII(node4, node3, node5);
// printf("LCA(3, 5) = %d\n", result1->val); // 应该输出 4

// 测试 LCA(5, 6) = 7
ParentTreeNode* result2 = lowestCommonAncestorII(node4, node5, node6);
// printf("LCA(5, 6) = %d\n", result2->val); // 应该输出 7

// 测试 LCA(6, 7) = 7
ParentTreeNode* result3 = lowestCommonAncestorII(node4, node6, node7);
// printf("LCA(6, 7) = %d\n", result3->val); // 应该输出 7

return 0;
}
```

=====

文件: LowestCommonAncestorII.java

```
=====
package class037;

import java.util.HashSet;
import java.util.Set;

// LintCode 474. Lowest Common Ancestor II
// 题目链接: https://www.lintcode.com/problem/474/
// 给定一个二叉树中的节点 a 和 b, 找出他们的最近公共祖先
// 每个节点包含一个指向其父节点的指针
//
// 解题思路:
// 1. 由于每个节点都有父节点指针, 可以将问题转化为链表相交问题
// 2. 从节点 a 开始, 沿着父节点指针向上遍历, 记录所有访问过的节点
// 3. 从节点 b 开始, 沿着父节点指针向上遍历, 第一个已经在步骤 2 中访问过的节点就是 LCA
//
// 时间复杂度: O(h) - h 为树的高度
// 空间复杂度: O(h) - 需要存储从 a 到根节点的路径
// 是否为最优解: 是, 利用父节点指针可以避免遍历整棵树
public class LowestCommonAncestorII {

    // 不提交这个类
    public static class ParentTreeNode {
        public int val;
        public ParentTreeNode parent, left, right;

        public ParentTreeNode(int val) {
            this.val = val;
        }
    }

    // 提交如下的方法
    public static ParentTreeNode lowestCommonAncestorII(ParentTreeNode root, ParentTreeNode a, ParentTreeNode b) {
        // 使用 HashSet 记录从节点 a 到根节点的路径
        Set<ParentTreeNode> visited = new HashSet<>();

        // 从节点 a 开始向上遍历到根节点, 记录路径上的所有节点
        ParentTreeNode current = a;
        while (current != null) {
            visited.add(current);
            current = current.parent;
        }

        // 从节点 b 开始向上遍历到根节点, 如果遇到之前访问过的节点, 那么它就是 LCA
        current = b;
        while (current != null) {
            if (visited.contains(current)) {
                return current;
            }
            current = current.parent;
        }
    }
}
```

```

// 从节点 b 开始向上遍历，第一个在 visited 集合中的节点就是 LCA
current = b;
while (current != null) {
    if (visited.contains(current)) {
        return current; // 找到 LCA
    }
    current = current.parent;
}

// 理论上不会执行到这里，因为 a 和 b 都在树中，肯定有公共祖先
return null;
}

// 测试用例
public static void main(String[] args) {
    // 构造测试用例：
    //      4
    //      / \
    //      3   7
    //      / \
    //      5   6
    ParentTreeNode node4 = new ParentTreeNode(4);
    ParentTreeNode node3 = new ParentTreeNode(3);
    ParentTreeNode node7 = new ParentTreeNode(7);
    ParentTreeNode node5 = new ParentTreeNode(5);
    ParentTreeNode node6 = new ParentTreeNode(6);

    // 设置父子关系
    node3.parent = node4;
    node7.parent = node4;
    node5.parent = node7;
    node6.parent = node7;

    node4.left = node3;
    node4.right = node7;
    node7.left = node5;
    node7.right = node6;

    // 测试 LCA(3, 5) = 4
    ParentTreeNode result1 = lowestCommonAncestorII(node4, node3, node5);
    System.out.println("LCA(3, 5) = " + result1.val); // 应该输出 4
}

```

```

// 测试 LCA(5, 6) = 7
ParentTreeNode result2 = lowestCommonAncestorII(node4, node5, node6);
System.out.println("LCA(5, 6) = " + result2.val); // 应该输出 7

// 测试 LCA(6, 7) = 7
ParentTreeNode result3 = lowestCommonAncestorII(node4, node6, node7);
System.out.println("LCA(6, 7) = " + result3.val); // 应该输出 7
}

}
=====
```

文件: LowestCommonAncestorII.py

```

# LintCode 474. Lowest Common Ancestor II
# 题目链接: https://www.lintcode.com/problem/474/
# 给定一个二叉树中的节点 a 和 b, 找出他们的最近公共祖先
# 每个节点包含一个指向其父节点的指针
#
# 解题思路:
# 1. 由于每个节点都有父节点指针, 可以将问题转化为链表相交问题
# 2. 从节点 a 开始, 沿着父节点指针向上遍历, 记录所有访问过的节点
# 3. 从节点 b 开始, 沿着父节点指针向上遍历, 第一个已经在步骤 2 中访问过的节点就是 LCA
#
# 时间复杂度: O(h) - h 为树的高度
# 空间复杂度: O(h) - 需要存储从 a 到根节点的路径
# 是否为最优解: 是, 利用父节点指针可以避免遍历整棵树
```

```
from typing import Optional
```

```

class ParentTreeNode:
    def __init__(self, val):
        self.val = val
        self.parent: Optional['ParentTreeNode'] = None
        self.left: Optional['ParentTreeNode'] = None
        self.right: Optional['ParentTreeNode'] = None

    def lowestCommonAncestorII(self, A: Optional[ParentTreeNode], B: Optional[ParentTreeNode]) -> Optional[ParentTreeNode]:
        """
        @param root: The root of the tree
        @param A: node in the tree
        @param B: node in the tree
        """
        if A == B:
            return A
```

```

@param B: node in the tree
@return: The lowest common ancestor of A and B
"""
# 使用集合记录从节点 A 到根节点的路径
visited = set()

# 从节点 A 开始向上遍历到根节点，记录路径上的所有节点
current = A
while current is not None:
    visited.add(current)
    current = current.parent

# 从节点 B 开始向上遍历，第一个在 visited 集合中的节点就是 LCA
current = B
while current is not None:
    if current in visited:
        return current # 找到 LCA
    current = current.parent

# 理论上不会执行到这里，因为 A 和 B 都在树中，肯定有公共祖先
return None

# 测试用例
if __name__ == "__main__":
    # 构造测试用例:
    #      4
    #     / \
    #    3   7
    #   /   \
    #  5   6
    node4 = ParentTreeNode(4)
    node3 = ParentTreeNode(3)
    node7 = ParentTreeNode(7)
    node5 = ParentTreeNode(5)
    node6 = ParentTreeNode(6)

    # 设置父子关系
    node3.parent = node4
    node7.parent = node4
    node5.parent = node7
    node6.parent = node7

    node4.left = node3

```

```

node4.right = node7
node7.left = node5
node7.right = node6

# 测试 LCA(3, 5) = 4
result1 = lowestCommonAncestorII(node4, node3, node5)
if result1:
    print(f'LCA(3, 5) = {result1.val}') # 应该输出 4

# 测试 LCA(5, 6) = 7
result2 = lowestCommonAncestorII(node4, node5, node6)
if result2:
    print(f'LCA(5, 6) = {result2.val}') # 应该输出 7

# 测试 LCA(6, 7) = 7
result3 = lowestCommonAncestorII(node4, node6, node7)
if result3:
    print(f'LCA(6, 7) = {result3.val}') # 应该输出 7

```

=====

文件: PathSum.cpp

```

// LeetCode 112. Path Sum
// 题目链接: https://leetcode.cn/problems/path-sum/
// 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
//
// 解题思路:
// 1. 使用递归深度优先搜索(DFS)
// 2. 从根节点开始，每访问一个节点，将目标和减去当前节点的值
// 3. 当到达叶子节点时，检查剩余的目标和是否等于当前节点的值
// 4. 递归检查左右子树是否存在满足条件的路径
//
// 时间复杂度: O(n) - n 为树中节点的数量，最坏情况下需要遍历所有节点
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
// 是否为最优解: 是，这是解决此类路径问题的标准方法

```

```

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
}

```

```

TreeNode() : val(0), left(nullptr), right(nullptr) {}
TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    bool hasPathSum(TreeNode* root, int targetSum) {
        // 边界情况: 空树
        if (root == nullptr) {
            return false;
        }

        // 到达叶子节点, 检查路径和是否等于目标和
        if (root->left == nullptr && root->right == nullptr) {
            return root->val == targetSum;
        }

        // 递归检查左右子树, 目标和减去当前节点的值
        return hasPathSum(root->left, targetSum - root->val) ||
               hasPathSum(root->right, targetSum - root->val);
    }
};

// 测试代码示例 (不提交)
/*
#include <iostream>
using namespace std;

int main() {
    Solution solution;

    // 构造测试用例:
    //      5
    //     / \
    //    4   8
    //   /   / \
    //  11  13  4
    // / \       \
    //7  2       1

    TreeNode* root = new TreeNode(5);
    root->left = new TreeNode(4);
    root->right = new TreeNode(8);
}

```

```

root->left->left = new TreeNode(11);
root->right->left = new TreeNode(13);
root->right->right = new TreeNode(4);
root->left->left->left = new TreeNode(7);
root->left->left->right = new TreeNode(2);
root->right->right->right = new TreeNode(1);

// 测试 targetSum = 22, 应该返回 true (5->4->11->2)
bool result1 = solution.hasPathSum(root, 22);
cout << "targetSum=22: " << (result1 ? "true" : "false") << endl; // 应该输出 true

// 测试 targetSum = 26, 应该返回 true (5->8->13)
bool result2 = solution.hasPathSum(root, 26);
cout << "targetSum=26: " << (result2 ? "true" : "false") << endl; // 应该输出 true

// 测试 targetSum = 19, 应该返回 true (5->8->4->1)
bool result3 = solution.hasPathSum(root, 19);
cout << "targetSum=19: " << (result3 ? "true" : "false") << endl; // 应该输出 true

// 测试 targetSum = 10, 应该返回 false
bool result4 = solution.hasPathSum(root, 10);
cout << "targetSum=10: " << (result4 ? "true" : "false") << endl; // 应该输出 false

return 0;
}
*/
=====

文件: PathSum.java
=====

package class037;

import java.util.ArrayList;
import java.util.List;

// LeetCode 112. Path Sum
// 题目链接: https://leetcode.cn/problems/path-sum/
// 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
//
// 解题思路:

```

```
// 1. 使用递归深度优先搜索(DFS)
// 2. 从根节点开始, 每访问一个节点, 将目标和减去当前节点的值
// 3. 当到达叶子节点时, 检查剩余的目标和是否等于当前节点的值
// 4. 递归检查左右子树是否存在满足条件的路径
//
// 时间复杂度: O(n) - n 为树中节点的数量, 最坏情况下需要遍历所有节点
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
// 是否为最优解: 是, 这是解决此类路径问题的标准方法
public class PathSum {

    // 不提交这个类
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {
        }

        TreeNode(int val) {
            this.val = val;
        }

        TreeNode(int val, TreeNode left, TreeNode right) {
            this.val = val;
            this.left = left;
            this.right = right;
        }
    }

    // 提交如下的方法
    public boolean hasPathSum(TreeNode root, int targetSum) {
        // 边界情况: 空树
        if (root == null) {
            return false;
        }

        // 到达叶子节点, 检查路径和是否等于目标和
        if (root.left == null && root.right == null) {
            return root.val == targetSum;
        }

        // 递归检查左右子树, 目标和减去当前节点的值
    }
}
```

```
        return hasPathSum(root.left, targetSum - root.val) ||
               hasPathSum(root.right, targetSum - root.val);
    }

// 测试用例
public static void main(String[] args) {
    PathSum solution = new PathSum();

    // 构造测试用例:
    //      5
    //      / \
    //      4   8
    //      /   / \
    //     11  13  4
    //     / \       \
    //    7   2       1

    TreeNode root = new TreeNode(5);
    root.left = new TreeNode(4);
    root.right = new TreeNode(8);
    root.left.left = new TreeNode(11);
    root.right.left = new TreeNode(13);
    root.right.right = new TreeNode(4);
    root.left.left.left = new TreeNode(7);
    root.left.left.right = new TreeNode(2);
    root.right.right.right = new TreeNode(1);

    // 测试 targetSum = 22, 应该返回 true (5->4->11->2)
    boolean result1 = solution.hasPathSum(root, 22);
    System.out.println("targetSum=22: " + result1); // 应该输出 true

    // 测试 targetSum = 26, 应该返回 true (5->8->13)
    boolean result2 = solution.hasPathSum(root, 26);
    System.out.println("targetSum=26: " + result2); // 应该输出 true

    // 测试 targetSum = 19, 应该返回 true (5->8->4->1)
    boolean result3 = solution.hasPathSum(root, 19);
    System.out.println("targetSum=19: " + result3); // 应该输出 true

    // 测试 targetSum = 10, 应该返回 false
    boolean result4 = solution.hasPathSum(root, 10);
    System.out.println("targetSum=10: " + result4); // 应该输出 false
}
```

文件: PathSum.py

```
=====
# LeetCode 112. Path Sum
# 题目链接: https://leetcode.cn/problems/path-sum/
# 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
#
# 解题思路：
# 1. 使用递归深度优先搜索(DFS)
# 2. 从根节点开始，每访问一个节点，将目标和减去当前节点的值
# 3. 当到达叶子节点时，检查剩余的目标和是否等于当前节点的值
# 4. 递归检查左右子树是否存在满足条件的路径
#
# 时间复杂度：O(n) - n 为树中节点的数量，最坏情况下需要遍历所有节点
# 空间复杂度：O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解：是，这是解决此类路径问题的标准方法
```

```
from typing import Optional
```

```
# Definition for a binary tree node.
```

```
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```
def hasPathSum(root: Optional[TreeNode], targetSum: int) -> bool:
```

```
    """

```

```
    判断二叉树中是否存在从根到叶子节点的路径，其节点值之和等于目标和

```

Args:

root: 二叉树的根节点  
targetSum: 目标和

Returns:

如果存在满足条件的路径返回 True，否则返回 False

```
    """

```

```
# 边界情况：空树
if root is None:
```

```
    return False

# 到达叶子节点，检查路径和是否等于目标和
if root.left is None and root.right is None:
    return root.val == targetSum

# 递归检查左右子树，目标和减去当前节点的值
return hasPathSum(root.left, targetSum - root.val) or \
    hasPathSum(root.right, targetSum - root.val)

# 测试用例
if __name__ == "__main__":
    # 构造测试用例：
    #      5
    #     / \
    #    4   8
    #   /   / \
    #  11  13  4
    # / \       \
    # 7   2       1
    root = TreeNode(5)
    root.left = TreeNode(4)
    root.right = TreeNode(8)
    root.left.left = TreeNode(11)
    root.right.left = TreeNode(13)
    root.right.right = TreeNode(4)
    root.left.left.left = TreeNode(7)
    root.left.left.right = TreeNode(2)
    root.right.right.right = TreeNode(1)

    # 测试 targetSum = 22，应该返回 true (5->4->11->2)
    result1 = hasPathSum(root, 22)
    print(f"targetSum=22: {result1}") # 应该输出 True

    # 测试 targetSum = 26，应该返回 true (5->8->13)
    result2 = hasPathSum(root, 26)
    print(f"targetSum=26: {result2}") # 应该输出 True

    # 测试 targetSum = 19，应该返回 true (5->8->4->1)
    result3 = hasPathSum(root, 19)
    print(f"targetSum=19: {result3}") # 应该输出 True

    # 测试 targetSum = 10，应该返回 false
```

```
result4 = hasPathSum(root, 10)
print(f"targetSum=10: {result4}") # 应该输出 False
```

---

文件: SerializeDeserializeBinaryTree.cpp

---

```
// LeetCode 297. Serialize and Deserialize Binary Tree
// 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
// 题目描述: 设计一个算法来序列化和反序列化二叉树。
// 序列化是将一个数据结构或者对象转换为连续的比特位的过程，进而可以将转换后的数据存储在一个文件或内存中，
// 同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。
//
// 解题思路:
// 1. 前序遍历序列化: 使用特殊字符表示空节点
// 2. 递归反序列化: 根据前序遍历顺序重建二叉树
// 3. 使用队列辅助反序列化: 更直观的迭代方法
//
// 时间复杂度:
// - 序列化: O(n) - 每个节点访问一次
// - 反序列化: O(n) - 每个节点处理一次
// 空间复杂度: O(n) - 需要存储序列化字符串或使用递归栈
// 是否为最优解: 是, 这是序列化二叉树的标准方法
```

```
#include <string>
#include <queue>
#include <iostream>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
};

class Codec {
private:
    // 序列化分隔符
    const string SEPARATOR = ",";
    const string NULL_NODE = "null";
```

```

public:
    // 方法 1: 前序遍历序列化 (递归)
    // 核心思想: 使用前序遍历, 空节点用"null"表示
    string serializePreorder(TreeNode* root) {
        if (root == nullptr) {
            return "";
        }

        stringstream ss;
        serializeHelper(root, ss);
        return ss.str();
    }

private:
    void serializeHelper(TreeNode* node, stringstream& ss) {
        if (node == nullptr) {
            ss << NULL_NODE << SEPARATOR;
            return;
        }

        // 前序遍历: 根->左->右
        ss << node->val << SEPARATOR;
        serializeHelper(node->left, ss);
        serializeHelper(node->right, ss);
    }

public:
    // 方法 2: 层序遍历序列化 (BFS)
    // 核心思想: 使用队列进行层序遍历, 更符合直观理解
    string serializeLevelOrder(TreeNode* root) {
        if (root == nullptr) {
            return "";
        }

        stringstream ss;
        queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();

```

```

    if (node == nullptr) {
        ss << NULL_NODE << SEPARATOR;
        continue;
    }

    ss << node->val << SEPARATOR;
    q.push(node->left);
    q.push(node->right);
}

return ss.str();
}

// 反序列化方法（前序遍历版本）
TreeNode* deserializePreorder(string data) {
    if (data.empty())
        return nullptr;

    queue<string> nodes;
    stringstream ss(data);
    string token;

    while (getline(ss, token, SEPARATOR[0])) {
        if (!token.empty())
            nodes.push(token);
    }
}

return deserializeHelper(nodes);
}

private:
TreeNode* deserializeHelper(queue<string>& nodes) {
    if (nodes.empty())
        return nullptr;

    string val = nodes.front();
    nodes.pop();

    if (val == NULL_NODE) {
        return nullptr;
    }
}

```

```
}

TreeNode* node = new TreeNode(stoi(val));
node->left = deserializeHelper(nodes);
node->right = deserializeHelper(nodes);

return node;
}

public:

// 反序列化方法 (层序遍历版本)
TreeNode* deserializeLevelOrder(string data) {
    if (data.empty())
        return nullptr;

    queue<string> nodes;
    stringstream ss(data);
    string token;

    while (getline(ss, token, SEPARATOR[0])) {
        if (!token.empty()) {
            nodes.push(token);
        }
    }

    if (nodes.empty())
        return nullptr;

    string rootVal = nodes.front();
    nodes.pop();

    if (rootVal == NULL_NODE) {
        return nullptr;
    }

    TreeNode* root = new TreeNode(stoi(rootVal));
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty() && !nodes.empty()) {
        TreeNode* node = q.front();
        q.pop();

        int leftVal = stoi(nodes.front());
        nodes.pop();

        if (leftVal != -1) {
            node->left = new TreeNode(leftVal);
            q.push(node->left);
        }

        int rightVal = stoi(nodes.front());
        nodes.pop();

        if (rightVal != -1) {
            node->right = new TreeNode(rightVal);
            q.push(node->right);
        }
    }
}
```

```

q.pop();

// 处理左子节点
if (!nodes.empty()) {
    string leftVal = nodes.front();
    nodes.pop();

    if (leftVal != NULL_NODE) {
        node->left = new TreeNode(stoi(leftVal));
        q.push(node->left);
    }
}

// 处理右子节点
if (!nodes.empty()) {
    string rightVal = nodes.front();
    nodes.pop();

    if (rightVal != NULL_NODE) {
        node->right = new TreeNode(stoi(rightVal));
        q.push(node->right);
    }
}

return root;
}

// 提交如下的方法（使用层序遍历版本，更直观）
string serialize(TreeNode* root) {
    return serializeLevelOrder(root);
}

TreeNode* deserialize(string data) {
    return deserializeLevelOrder(data);
}

// 测试用例
// int main() {
//     Codec codec;
//     //
//     // 测试用例 1:

```

```

//      //      1
//      //      / \
//      //      2   3
//      //      / \
//      //      4   5
// TreeNode* root1 = new TreeNode(1);
// root1->left = new TreeNode(2);
// root1->right = new TreeNode(3);
// root1->right->left = new TreeNode(4);
// root1->right->right = new TreeNode(5);
//
// // 序列化
// string serialized = codec.serialize(root1);
// // 反序列化
// TreeNode* deserialized = codec.deserialize(serialized);
//
// // 验证结果...
//
// return 0;
// }

```

=====

文件: SerializeDeserializeBinaryTree.java

=====

```

package class037;

import java.util.*;

// LeetCode 297. Serialize and Deserialize Binary Tree
// 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
// 题目描述: 设计一个算法来序列化和反序列化二叉树。
// 序列化是将一个数据结构或者对象转换为连续的比特位的过程，进而可以将转换后的数据存储在一个文件或内存中，
// 同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。
//
// 解题思路:
// 1. 前序遍历序列化: 使用特殊字符表示空节点
// 2. 递归反序列化: 根据前序遍历顺序重建二叉树
// 3. 使用队列辅助反序列化: 更直观的迭代方法
//
// 时间复杂度:
// - 序列化: O(n) - 每个节点访问一次

```

```
// - 反序列化: O(n) - 每个节点处理一次
// 空间复杂度: O(n) - 需要存储序列化字符串或使用递归栈
// 是否为最优解: 是, 这是序列化二叉树的标准方法

// 补充题目:
// 1. LeetCode 449. Serialize and Deserialize BST - BST 的序列化与反序列化
// 2. LeetCode 428. Serialize and Deserialize N-ary Tree - N 叉树的序列化
// 3. 剑指 Offer 37. 序列化二叉树 - 与 LeetCode 297 相同
```

```
public class SerializeDeserializeBinaryTree {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode(int x) {
            val = x;
        }
    }

    // 序列化分隔符
    private static final String SEPARATOR = ",";
    private static final String NULL_NODE = "null";

    // 方法 1: 前序遍历序列化 (递归)
    // 核心思想: 使用前序遍历, 空节点用"null"表示
    public String serializePreorder(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        serializeHelper(root, sb);
        return sb.toString();
    }

    private void serializeHelper(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append(NULL_NODE).append(SEPARATOR);
            return;
        }

        // 前序遍历: 根->左->右
        sb.append(node.val).append(SEPARATOR);
        serializeHelper(node.left, sb);
        serializeHelper(node.right, sb);
    }
}
```

```
    serializeHelper(node.right, sb);
}

// 方法2：层序遍历序列化（BFS）
// 核心思想：使用队列进行层序遍历，更符合直观理解
public String serializeLevelOrder(TreeNode root) {
    if (root == null) {
        return "";
    }

    StringBuilder sb = new StringBuilder();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();

        if (node == null) {
            sb.append(NULL_NODE).append(SEPARATOR);
            continue;
        }

        sb.append(node.val).append(SEPARATOR);
        queue.offer(node.left);
        queue.offer(node.right);
    }

    return sb.toString();
}

// 反序列化方法（前序遍历版本）
public TreeNode deserializePreorder(String data) {
    if (data == null || data.isEmpty()) {
        return null;
    }

    String[] nodes = data.split(SEPARATOR);
    Queue<String> queue = new LinkedList<>(Arrays.asList(nodes));
    return deserializeHelper(queue);
}

private TreeNode deserializeHelper(Queue<String> queue) {
    if (queue.isEmpty()) {

```

```
        return null;
    }

    String val = queue.poll();
    if (val.equals(NULL_NODE)) {
        return null;
    }

    TreeNode node = new TreeNode(Integer.parseInt(val));
    node.left = deserializeHelper(queue);
    node.right = deserializeHelper(queue);

    return node;
}

// 反序列化方法 (层序遍历版本)
public TreeNode deserializeLevelOrder(String data) {
    if (data == null || data.isEmpty()) {
        return null;
    }

    String[] nodes = data.split(SEPARATOR);
    if (nodes.length == 0) {
        return null;
    }

    TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    int index = 1;
    while (!queue.isEmpty() && index < nodes.length) {
        TreeNode node = queue.poll();

        // 处理左子节点
        if (index < nodes.length && !nodes[index].equals(NULL_NODE)) {
            node.left = new TreeNode(Integer.parseInt(nodes[index]));
            queue.offer(node.left);
        }
        index++;

        // 处理右子节点
        if (index < nodes.length && !nodes[index].equals(NULL_NODE)) {
```

```

        node.right = new TreeNode(Integer.parseInt(nodes[index]));
        queue.offer(node.right);
    }
    index++;
}

return root;
}

// 提交如下的方法（使用层序遍历版本，更直观）
public String serialize(TreeNode root) {
    return serializeLevelOrder(root);
}

public TreeNode deserialize(String data) {
    return deserializeLevelOrder(data);
}

// 测试用例
public static void main(String[] args) {
    SerializeDeserializeBinaryTree codec = new SerializeDeserializeBinaryTree();

    // 测试用例 1:
    //      1
    //      / \
    //     2   3
    //     / \
    //    4   5
    TreeNode root1 = new TreeNode(1);
    root1.left = new TreeNode(2);
    root1.right = new TreeNode(3);
    root1.right.left = new TreeNode(4);
    root1.right.right = new TreeNode(5);

    // 序列化
    String serialized = codec.serialize(root1);
    System.out.println("序列化结果: " + serialized);
    // 应该输出类似: "1,2,3,null,null,4,5,null,null,null,null,"

    // 反序列化
    TreeNode deserialized = codec.deserialize(serialized);
    System.out.println("反序列化验证:");
    printLevelOrder(deserialized); // 应该输出与原树相同的结构
}

```

```

// 测试用例 2: 空树
TreeNode root2 = null;
String serialized2 = codec.serialize(root2);
System.out.println("空树序列化: " + serialized2);
TreeNode deserialized2 = codec.deserialize(serialized2);
System.out.println("空树反序列化验证: " + (deserialized2 == null));

// 测试用例 3: 单节点树
TreeNode root3 = new TreeNode(1);
String serialized3 = codec.serialize(root3);
System.out.println("单节点树序列化: " + serialized3);
TreeNode deserialized3 = codec.deserialize(serialized3);
System.out.println("单节点树反序列化验证: " + (deserialized3.val == 1));

// 补充题目测试: BST 的序列化与反序列化
System.out.println("\n==== 补充题目测试: BST 序列化与反序列化 ====");
TreeNode bstRoot = new TreeNode(2);
bstRoot.left = new TreeNode(1);
bstRoot.right = new TreeNode(3);

String bstSerialized = serializeBST(bstRoot);
System.out.println("BST 序列化: " + bstSerialized);
TreeNode bstDeserialized = deserializeBST(bstSerialized);
System.out.println("BST 反序列化验证: " + isValidBST(bstDeserialized));
}

// 辅助方法: 层序遍历打印二叉树 (用于验证)
private static void printLevelOrder(TreeNode root) {
    if (root == null) {
        System.out.println("空树");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    while (!queue.isEmpty()) {
        int levelSize = queue.size();
        List<String> level = new ArrayList<>();

        for (int i = 0; i < levelSize; i++) {
            TreeNode node = queue.poll();

```

```

        level.add(String.valueOf(node.val));

        if (node.left != null) {
            queue.offer(node.left);
        }
        if (node.right != null) {
            queue.offer(node.right);
        }
    }

    System.out.println(String.join(" ", level));
}
}

// 补充题目: LeetCode 449. Serialize and Deserialize BST
// BST 的序列化可以利用 BST 的性质进行优化
public static String serializeBST(TreeNode root) {
    if (root == null) {
        return "";
    }

    StringBuilder sb = new StringBuilder();
    serializeBSTHelper(root, sb);
    return sb.toString();
}

private static void serializeBSTHelper(TreeNode node, StringBuilder sb) {
    if (node == null) {
        return;
    }

    sb.append(node.val).append(SEPARATOR);
    serializeBSTHelper(node.left, sb);
    serializeBSTHelper(node.right, sb);
}

public static TreeNode deserializeBST(String data) {
    if (data == null || data.isEmpty()) {
        return null;
    }

    String[] nodes = data.split(SEPARATOR);
    Queue<Integer> queue = new LinkedList<>();

```

```
for (String node : nodes) {
    queue.offer(Integer.parseInt(node));
}

return deserializeBSTHelper(queue, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private static TreeNode deserializeBSTHelper(Queue<Integer> queue, int lower, int upper) {
    if (queue.isEmpty()) {
        return null;
    }

    int val = queue.peek();
    if (val < lower || val > upper) {
        return null;
    }

    queue.poll();
    TreeNode node = new TreeNode(val);
    node.left = deserializeBSTHelper(queue, lower, val);
    node.right = deserializeBSTHelper(queue, val, upper);

    return node;
}

// 验证 BST 是否有效
private static boolean isValidBST(TreeNode root) {
    return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
}

private static boolean isValidBSTHelper(TreeNode node, long lower, long upper) {
    if (node == null) {
        return true;
    }

    if (node.val <= lower || node.val >= upper) {
        return false;
    }

    return isValidBSTHelper(node.left, lower, node.val) &&
           isValidBSTHelper(node.right, node.val, upper);
}
```

文件: SerializeDeserializeBinaryTree.py

```
# LeetCode 297. Serialize and Deserialize Binary Tree
# 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
# 题目描述: 设计一个算法来序列化和反序列化二叉树。
# 序列化是将一个数据结构或者对象转换为连续的比特位的过程，进而可以将转换后的数据存储在一个文件或内存中，
# 同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。
#
# 解题思路:
# 1. 前序遍历序列化: 使用特殊字符表示空节点
# 2. 递归反序列化: 根据前序遍历顺序重建二叉树
# 3. 使用队列辅助反序列化: 更直观的迭代方法
#
# 时间复杂度:
# - 序列化: O(n) - 每个节点访问一次
# - 反序列化: O(n) - 每个节点处理一次
# 空间复杂度: O(n) - 需要存储序列化字符串或使用递归栈
# 是否为最优解: 是, 这是序列化二叉树的标准方法
```

```
from typing import Optional, List, Deque, Any
from collections import deque
```

```
# 二叉树节点定义
```

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left: Optional[TreeNode] = None
        self.right: Optional[TreeNode] = None
```

```
class Codec:
```

```
    # 序列化分隔符
    SEPARATOR = ","
    NULL_NODE = "null"
```

```
    def serializePreorder(self, root: Optional[TreeNode]) -> str:
        """
```

```
        前序遍历序列化 (递归)
```

```
Args:
```

root: 二叉树的根节点

Returns:

序列化后的字符串

"""

if root is None:

    return ""

result = []

self.\_serialize\_helper(root, result)

return self.SEPARATOR.join(result)

def \_serialize\_helper(self, node: Optional[TreeNode], result: List[str]) -> None:

"""

递归序列化辅助函数

Args:

node: 当前节点

result: 结果列表

"""

if node is None:

    result.append(self.NULL\_NODE)

    return

# 前序遍历: 根->左->右

result.append(str(node.val))

self.\_serialize\_helper(node.left, result)

self.\_serialize\_helper(node.right, result)

def serializeLevelOrder(self, root: Optional[TreeNode]) -> str:

"""

层序遍历序列化 (BFS)

Args:

root: 二叉树的根节点

Returns:

序列化后的字符串

"""

if root is None:

    return ""

result = []

```
queue: Deque[Optional[TreeNode]] = deque([root])

while queue:
    node = queue.popleft()

    if node is None:
        result.append(self.NULL_NODE)
        continue

    result.append(str(node.val))
    queue.append(node.left)
    queue.append(node.right)

return self.SEPARATOR.join(result)
```

```
def deserializePreorder(self, data: str) -> Optional[TreeNode]:
```

```
"""
```

```
前序遍历反序列化
```

Args:

data: 序列化字符串

Returns:

反序列化后的二叉树根节点

```
"""
```

```
if not data:
```

```
    return None
```

```
nodes = deque(data.split(self.SEPARATOR))
return self._deserialize_helper(nodes)
```

```
def _deserialize_helper(self, nodes: Deque[str]) -> Optional[TreeNode]:
```

```
"""
```

```
递归反序列化辅助函数
```

Args:

nodes: 节点值队列

Returns:

构建的子树根节点

```
"""
```

```
if not nodes:
```

```
    return None
```

```
val = nodes.popleft()
if val == self.NULL_NODE:
    return None

node = TreeNode(int(val))
left_node = self._deserialize_helper(nodes)
right_node = self._deserialize_helper(nodes)
node.left = left_node
node.right = right_node

return node
```

```
def deserializeLevelOrder(self, data: str) -> Optional[TreeNode]:
    """
    层序遍历反序列化
    
```

Args:

    data: 序列化字符串

Returns:

    反序列化后的二叉树根节点
 """

```
if not data:
    return None
```

```
nodes = data.split(self.SEPARATOR)
if not nodes or nodes[0] == self.NULL_NODE:
    return None
```

```
root = TreeNode(int(nodes[0]))
queue: Deque[TreeNode] = deque([root])
index = 1
```

```
while queue and index < len(nodes):
    node = queue.popleft()

    # 处理左子节点
    if index < len(nodes) and nodes[index] != self.NULL_NODE:
        left_node = TreeNode(int(nodes[index]))
        node.left = left_node
        queue.append(left_node)

    index += 1
```

```
# 处理右子节点
    if index < len(nodes) and nodes[index] != self.NULL_NODE:
        right_node = TreeNode(int(nodes[index]))
        node.right = right_node
        queue.append(right_node)
    index += 1

return root
```

```
def serialize(self, root: Optional[TreeNode]) -> str:
```

```
    """

```

```
    序列化二叉树（使用层序遍历版本，更直观）
```

Args:

root: 二叉树的根节点

Returns:

序列化后的字符串

```
    """

```

```
    return self.serializeLevelOrder(root)
```

```
def deserialize(self, data: str) -> Optional[TreeNode]:
```

```
    """

```

```
    反序列化二叉树（使用层序遍历版本，更直观）
```

Args:

data: 序列化字符串

Returns:

反序列化后的二叉树根节点

```
    """

```

```
    return self.deserializeLevelOrder(data)
```

```
# 测试用例
```

```
def main():
```

```
    codec = Codec()
```

```
# 测试用例 1:
```

```
#     1
```

```
#     / \

```

```
#     2   3
```

```
#     / \
```

```

#       4   5
root1 = TreeNode(1)
root1.left = TreeNode(2)
root1.right = TreeNode(3)
root1.right.left = TreeNode(4)
root1.right.right = TreeNode(5)

# 序列化
serialized = codec.serialize(root1)
print(f"序列化结果: {serialized}")

# 反序列化
deserialized = codec.deserialize(serialized)
print("反序列化验证:")
print_level_order(deserialized) # 应该输出与原树相同的结构

# 测试用例 2: 空树
root2 = None
serialized2 = codec.serialize(root2)
print(f"空树序列化: {serialized2}")
deserialized2 = codec.deserialize(serialized2)
print(f"空树反序列化验证: {deserialized2 is None}")

```

```
def print_level_order(root: Optional[TreeNode]) -> None:
    """

```

层序遍历打印二叉树（用于验证）

Args:

root: 二叉树的根节点

"""

if root is None:

print("空树")

return

queue = deque([root])

while queue:

level\_size = len(queue)

level = []

for \_ in range(level\_size):

node = queue.popleft()

level.append(str(node.val))

```

        if node.left is not None:
            queue.append(node.left)
        if node.right is not None:
            queue.append(node.right)

    print(" ".join(level))

if __name__ == "__main__":
    main()
=====
```

文件: SymmetricTree.cpp

```

// LeetCode 101. Symmetric Tree
// 题目链接: https://leetcode.cn/problems/symmetric-tree/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 同时遍历左右子树，检查是否镜像对称
// 2. 迭代方法: 使用队列进行层序遍历，检查对称性
//
// 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
// 空间复杂度:
//   - 递归: O(h) - h 为树的高度，递归调用栈的深度
//   - 迭代: O(w) - w 为树的最大宽度，队列中最多存储一层的节点
// 是否为最优解: 是，这是检查对称二叉树的标准方法
```

```

#include <queue>
#include <algorithm>
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};
```

```

class Solution {
public:
    // 方法1：递归解法
    // 核心思想：同时遍历左右子树，检查左子树的左节点是否等于右子树的右节点，
    //           左子树的右节点是否等于右子树的左节点
    bool isSymmetricRecursive(TreeNode* root) {
        if (root == nullptr) {
            return true;
        }
        return isMirror(root->left, root->right);
    }

private:
    bool isMirror(TreeNode* left, TreeNode* right) {
        // 两个节点都为空，对称
        if (left == nullptr && right == nullptr) {
            return true;
        }
        // 一个为空一个不为空，不对称
        if (left == nullptr || right == nullptr) {
            return false;
        }
        // 节点值不相等，不对称
        if (left->val != right->val) {
            return false;
        }
        // 递归检查左子树的左节点和右子树的右节点，
        // 以及左子树的右节点和右子树的左节点
        return isMirror(left->left, right->right) && isMirror(left->right, right->left);
    }

public:
    // 方法2：迭代解法（BFS）
    // 核心思想：使用队列进行层序遍历，每次入队两个节点进行比较
    bool isSymmetricIterative(TreeNode* root) {
        if (root == nullptr) {
            return true;
        }

        queue<TreeNode*> q;
        q.push(root->left);
        q.push(root->right);

        while (!q.empty()) {
            TreeNode* left = q.front();
            q.pop();
            TreeNode* right = q.front();
            q.pop();

            if (left == nullptr && right == nullptr) {
                continue;
            }
            if (left == nullptr || right == nullptr) {
                return false;
            }
            if (left->val != right->val) {
                return false;
            }

            q.push(left->left);
            q.push(right->right);
            q.push(left->right);
            q.push(right->left);
        }

        return true;
    }
}

```

```
while (!q.empty()) {
    TreeNode* left = q.front();
    q.pop();
    TreeNode* right = q.front();
    q.pop();

    // 两个节点都为空，继续检查
    if (left == nullptr && right == nullptr) {
        continue;
    }

    // 一个为空一个不为空，不对称
    if (left == nullptr || right == nullptr) {
        return false;
    }

    // 节点值不相等，不对称
    if (left->val != right->val) {
        return false;
    }

    // 按对称顺序入队子节点
    q.push(left->left);
    q.push(right->right);
    q.push(left->right);
    q.push(right->left);
}

return true;
}

// 提交如下的方法（使用递归版本）
bool isSymmetric(TreeNode* root) {
    return isSymmetricRecursive(root);
}

// 测试用例
int main() {
    Solution solution;

    // 测试用例 1：对称二叉树
    //      1
    //     / \

```

```

//      2   2
//      / \ / \
//      3   4 4   3
TreeNode* root1 = new TreeNode(1);
root1->left = new TreeNode(2);
root1->right = new TreeNode(2);
root1->left->left = new TreeNode(3);
root1->left->right = new TreeNode(4);
root1->right->left = new TreeNode(4);
root1->right->right = new TreeNode(3);

bool result1 = solution.isSymmetric(root1);
cout << "测试用例 1 结果: " << (result1 ? "true" : "false") << endl; // 应该输出 true

// 测试用例 2: 不对称二叉树
//      1
//      / \
//      2   2
//      \   \
//      3   3
TreeNode* root2 = new TreeNode(1);
root2->left = new TreeNode(2);
root2->right = new TreeNode(2);
root2->left->right = new TreeNode(3);
root2->right->right = new TreeNode(3);

bool result2 = solution.isSymmetric(root2);
cout << "测试用例 2 结果: " << (result2 ? "true" : "false") << endl; // 应该输出 false

// 测试用例 3: 空树
TreeNode* root3 = nullptr;
bool result3 = solution.isSymmetric(root3);
cout << "测试用例 3 结果: " << (result3 ? "true" : "false") << endl; // 应该输出 true

// 测试用例 4: 单节点树
TreeNode* root4 = new TreeNode(1);
bool result4 = solution.isSymmetric(root4);
cout << "测试用例 4 结果: " << (result4 ? "true" : "false") << endl; // 应该输出 true

// 补充题目测试: LeetCode 100 - 相同的树
cout << "\n==== 补充题目测试: 相同的树 ===" << endl;
// 构造两棵相同的树
TreeNode* tree1 = new TreeNode(1);

```

```
tree1->left = new TreeNode(2);
tree1->right = new TreeNode(3);

TreeNode* tree2 = new TreeNode(1);
tree2->left = new TreeNode(2);
tree2->right = new TreeNode(3);

// 相同的树函数实现
auto isSameTree = [] (TreeNode* p, TreeNode* q) -> bool {
    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);
};

bool sameResult = isSameTree(tree1, tree2);
cout << "相同的树测试: " << (sameResult ? "true" : "false") << endl; // 应该输出 true

// 内存清理
delete root1->left->left;
delete root1->left->right;
delete root1->right->left;
delete root1->right->right;
delete root1->left;
delete root1->right;
delete root1;

delete root2->left->right;
delete root2->right->right;
delete root2->left;
delete root2->right;
delete root2;

delete root4;

delete tree1->left;
delete tree1->right;
delete tree1;

delete tree2->left;
delete tree2->right;
delete tree2;
```

```
    return 0;
}
```

---

文件: SymmetricTree.java

---

```
package class037;

import java.util.LinkedList;
import java.util.Queue;

// LeetCode 101. Symmetric Tree
// 题目链接: https://leetcode.cn/problems/symmetric-tree/
// 题目描述: 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。
//
// 解题思路:
// 1. 递归方法: 同时遍历左右子树，检查是否镜像对称
// 2. 迭代方法: 使用队列进行层序遍历，检查对称性
//
// 时间复杂度: O(n) - n 为树中节点的数量，每个节点访问一次
// 空间复杂度:
// - 递归: O(h) - h 为树的高度，递归调用栈的深度
// - 迭代: O(w) - w 为树的最大宽度，队列中最多存储一层的节点
// 是否为最优解: 是，这是检查对称二叉树的标准方法

// 补充题目:
// 1. LeetCode 100. Same Tree - 判断两棵树是否相同
// 2. LeetCode 226. Invert Binary Tree - 翻转二叉树
// 3. 牛客 NC16. 对称的二叉树 - 与 LeetCode 101 相同

public class SymmetricTree {

    // 二叉树节点定义
    public static class TreeNode {
        int val;
        TreeNode left;
        TreeNode right;

        TreeNode() {}

        TreeNode(int val) {
```

```

        this.val = val;
    }

TreeNode(int val, TreeNode left, TreeNode right) {
    this.val = val;
    this.left = left;
    this.right = right;
}
}

// 方法 1: 递归解法
// 核心思想: 同时遍历左右子树, 检查左子树的左节点是否等于右子树的右节点,
//           左子树的右节点是否等于右子树的左节点
public boolean isSymmetricRecursive(TreeNode root) {
    if (root == null) {
        return true;
    }
    return isMirror(root.left, root.right);
}

private boolean isMirror(TreeNode left, TreeNode right) {
    // 两个节点都为空, 对称
    if (left == null && right == null) {
        return true;
    }
    // 一个为空一个不为空, 不对称
    if (left == null || right == null) {
        return false;
    }
    // 节点值不相等, 不对称
    if (left.val != right.val) {
        return false;
    }
    // 递归检查左子树的左节点和右子树的右节点,
    // 以及左子树的右节点和右子树的左节点
    return isMirror(left.left, right.right) && isMirror(left.right, right.left);
}

// 方法 2: 迭代解法 (BFS)
// 核心思想: 使用队列进行层序遍历, 每次入队两个节点进行比较
public boolean isSymmetricIterative(TreeNode root) {
    if (root == null) {
        return true;
    }

```

```
}

Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root.left);
queue.offer(root.right);

while (!queue.isEmpty()) {
    TreeNode left = queue.poll();
    TreeNode right = queue.poll();

    // 两个节点都为空，继续检查
    if (left == null && right == null) {
        continue;
    }

    // 一个为空一个不为空，不对称
    if (left == null || right == null) {
        return false;
    }

    // 节点值不相等，不对称
    if (left.val != right.val) {
        return false;
    }

    // 按对称顺序入队子节点
    queue.offer(left.left);
    queue.offer(right.right);
    queue.offer(left.right);
    queue.offer(right.left);
}

return true;
}

// 提交如下的方法（使用递归版本）
public boolean isSymmetric(TreeNode root) {
    return isSymmetricRecursive(root);
}

// 测试用例
public static void main(String[] args) {
    SymmetricTree solution = new SymmetricTree();

    // 测试用例 1：对称二叉树
}
```

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

TreeNode root1 = new TreeNode(1);
root1.left = new TreeNode(2);
root1.right = new TreeNode(2);
root1.left.left = new TreeNode(3);
root1.left.right = new TreeNode(4);
root1.right.left = new TreeNode(4);
root1.right.right = new TreeNode(3);

boolean result1 = solution.isSymmetric(root1);
System.out.println("测试用例 1 结果: " + result1); // 应该输出 true
```

// 测试用例 2: 不对称二叉树

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

TreeNode root2 = new TreeNode(1);
root2.left = new TreeNode(2);
root2.right = new TreeNode(2);
root2.left.right = new TreeNode(3);
root2.right.right = new TreeNode(3);
```

boolean result2 = solution.isSymmetric(root2);

System.out.println("测试用例 2 结果: " + result2); // 应该输出 false

// 测试用例 3: 空树

```
TreeNode root3 = null;
boolean result3 = solution.isSymmetric(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出 true
```

// 测试用例 4: 单节点树

```
TreeNode root4 = new TreeNode(1);
boolean result4 = solution.isSymmetric(root4);
System.out.println("测试用例 4 结果: " + result4); // 应该输出 true
```

// 补充题目测试: LeetCode 100 - 相同的树

System.out.println("\n== 补充题目测试: 相同的树 ==");

```

// 构造两棵相同的树
TreeNode tree1 = new TreeNode(1);
tree1.left = new TreeNode(2);
tree1.right = new TreeNode(3);

TreeNode tree2 = new TreeNode(1);
tree2.left = new TreeNode(2);
tree2.right = new TreeNode(3);

boolean sameResult = isSameTree(tree1, tree2);
System.out.println("相同的树测试: " + sameResult); // 应该输出 true
}

// 补充题目: LeetCode 100. Same Tree
// 题目链接: https://leetcode.cn/problems/same-tree/
// 判断两棵树是否相同
public static boolean isSameTree(TreeNode p, TreeNode q) {
    // 两个节点都为空, 相同
    if (p == null && q == null) {
        return true;
    }
    // 一个为空一个不为空, 不同
    if (p == null || q == null) {
        return false;
    }
    // 节点值不相等, 不同
    if (p.val != q.val) {
        return false;
    }
    // 递归检查左右子树
    return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
}

```

=====

文件: SymmetricTree.py

=====

```

# LeetCode 101. Symmetric Tree
# 题目链接: https://leetcode.cn/problems/symmetric-tree/
# 题目描述: 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。
#

```

```

# 解题思路:
# 1. 递归方法: 同时遍历左右子树, 检查是否镜像对称
# 2. 迭代方法: 使用队列进行层序遍历, 检查对称性
#
# 时间复杂度: O(n) - n 为树中节点的数量, 每个节点访问一次
# 空间复杂度:
#   - 递归: O(h) - h 为树的高度, 递归调用栈的深度
#   - 迭代: O(w) - w 为树的最大宽度, 队列中最多存储一层的节点
# 是否为最优解: 是, 这是检查对称二叉树的标准方法

from typing import Optional
from collections import deque

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    # 方法 1: 递归解法
    # 核心思想: 同时遍历左右子树, 检查左子树的左节点是否等于右子树的右节点,
    #           左子树的右节点是否等于右子树的左节点
    def isSymmetricRecursive(self, root: Optional[TreeNode]) -> bool:
        """
        递归方法检查二叉树是否对称
        """

        Args:
            root: 二叉树的根节点

        Returns:
            如果二叉树对称返回 True, 否则返回 False
        """
        if root is None:
            return True
        return self._isMirror(root.left, root.right)

    def _isMirror(self, left: Optional[TreeNode], right: Optional[TreeNode]) -> bool:
        """
        递归辅助函数, 检查两个子树是否镜像对称
        """

        Args:

```

left: 左子树根节点  
right: 右子树根节点

Returns:

如果两个子树镜像对称返回 True, 否则返回 False

"""

# 两个节点都为空, 对称

if left is None and right is None:

    return True

# 一个为空一个不为空, 不对称

if left is None or right is None:

    return False

# 节点值不相等, 不对称

if left.val != right.val:

    return False

# 递归检查左子树的左节点和右子树的右节点,

# 以及左子树的右节点和右子树的左节点

return (self.\_isMirror(left.left, right.right) and

        self.\_isMirror(left.right, right.left))

# 方法 2: 迭代解法 (BFS)

# 核心思想: 使用队列进行层序遍历, 每次入队两个节点进行比较

def isSymmetricIterative(self, root: Optional[TreeNode]) -> bool:

"""

迭代方法检查二叉树是否对称

Args:

root: 二叉树的根节点

Returns:

如果二叉树对称返回 True, 否则返回 False

"""

if root is None:

    return True

queue = deque()

queue.append(root.left)

queue.append(root.right)

while queue:

    left = queue.popleft()

    right = queue.popleft()

```
# 两个节点都为空，继续检查
if left is None and right is None:
    continue
# 一个为空一个不为空，不对称
if left is None or right is None:
    return False
# 节点值不相等，不对称
if left.val != right.val:
    return False

# 按对称顺序入队子节点
queue.append(left.left)
queue.append(right.right)
queue.append(left.right)
queue.append(right.left)

return True

# 提交如下的方法（使用递归版本）
def isSymmetric(self, root: Optional[TreeNode]) -> bool:
    return self.isSymmetricRecursive(root)

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1：对称二叉树
    #
    #      1
    #     / \
    #    2   2
    #   / \ / \
    #  3  4 4  3
    root1 = TreeNode(1)
    root1.left = TreeNode(2)
    root1.right = TreeNode(2)
    root1.left.left = TreeNode(3)
    root1.left.right = TreeNode(4)
    root1.right.left = TreeNode(4)
    root1.right.right = TreeNode(3)

    result1 = solution.isSymmetric(root1)
    print(f"测试用例 1 结果: {result1}")  # 应该输出 True
```

```

# 测试用例 2: 不对称二叉树
#      1
#    / \
#   2   2
#   \   \
#     3   3
root2 = TreeNode(1)
root2.left = TreeNode(2)
root2.right = TreeNode(2)
root2.left.right = TreeNode(3)
root2.right.right = TreeNode(3)

result2 = solution.isSymmetric(root2)
print(f"测试用例 2 结果: {result2}")  # 应该输出 False

# 测试用例 3: 空树
root3 = None
result3 = solution.isSymmetric(root3)
print(f"测试用例 3 结果: {result3}")  # 应该输出 True

# 测试用例 4: 单节点树
root4 = TreeNode(1)
result4 = solution.isSymmetric(root4)
print(f"测试用例 4 结果: {result4}")  # 应该输出 True

# 补充题目测试: LeetCode 100 - 相同的树
print("\n==== 补充题目测试: 相同的树 ====")
# 构造两棵相同的树
tree1 = TreeNode(1)
tree1.left = TreeNode(2)
tree1.right = TreeNode(3)

tree2 = TreeNode(1)
tree2.left = TreeNode(2)
tree2.right = TreeNode(3)

def isSameTree(p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
    """
    判断两棵树是否相同
    """

Args:
    p: 第一棵树的根节点
    q: 第二棵树的根节点

```

Returns:

如果两棵树相同返回 True，否则返回 False

"""

```
if p is None and q is None:  
    return True  
  
if p is None or q is None:  
    return False  
  
if p.val != q.val:  
    return False  
  
return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

```
same_result = isSameTree(tree1, tree2)
```

```
print(f"相同的树测试: {same_result}") # 应该输出 True
```

```
if __name__ == "__main__":  
    main()
```

=====

文件: ValidateBinarySearchTree.cpp

=====

```
// LeetCode 98. Validate Binary Search Tree  
// 题目链接: https://leetcode.cn/problems/validate-binary-search-tree/  
// 题目描述: 给你一个二叉树的根节点 root ，判断其是否是一个有效的二叉搜索树。  
// 有效二叉搜索树定义如下：  
// 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。  
//  
// 解题思路：  
// 1. 使用递归方法，为每个节点设置上下界  
// 2. 对于根节点，上下界为无穷大和无穷小  
// 3. 对于左子树，上界更新为当前节点值  
// 4. 对于右子树，下界更新为当前节点值  
// 5. 递归检查每个节点是否满足上下界约束  
//  
// 时间复杂度: O(n) - n 为树中节点的数量，需要访问每个节点  
// 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度  
// 是否为最优解: 是，这是验证 BST 的标准方法
```

```
#include <climits>  
#include <cfloat>
```

```
using namespace std;

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * 验证二叉搜索树
     * @param root 二叉树的根节点
     * @return 是否为有效的 BST
     */
    bool isValidBST(TreeNode* root) {
        return isValidBST(root, (long)LONG_MIN, (long)LONG_MAX);
    }

private:
    /**
     * 递归验证 BST
     * @param node 当前节点
     * @param min 下界 (不包含)
     * @param max 上界 (不包含)
     * @return 是否为有效的 BST
     */
    bool isValidBST(TreeNode* node, long min, long max) {
        // 空节点是有效的 BST
        if (node == nullptr) {
            return true;
        }

        // 检查当前节点是否满足上下界约束
        if (node->val <= min || node->val >= max) {
            return false;
        }

        // 递归检查左右子树
        return isValidBST(node->left, min, node->val) && isValidBST(node->right, node->val, max);
    }
}
```

```
// 左子树的上界更新为当前节点值
// 右子树的下界更新为当前节点值
return isValidBST(node->left, min, node->val) &&
      isValidBST(node->right, node->val, max);
}

};

// 测试用例
// int main() {
//     Solution solution;

//     // 测试用例 1: 有效的 BST
//     //      2
//     //      / \
//     //      1   3
//     TreeNode* root1 = new TreeNode(2);
//     root1->left = new TreeNode(1);
//     root1->right = new TreeNode(3);
//     bool result1 = solution.isValidBST(root1);
//     // 应该输出 true

//     // 测试用例 2: 无效的 BST
//     //      5
//     //      / \
//     //      1   4
//     //      / \
//     //      3   6
//     TreeNode* root2 = new TreeNode(5);
//     root2->left = new TreeNode(1);
//     root2->right = new TreeNode(4);
//     root2->right->left = new TreeNode(3);
//     root2->right->right = new TreeNode(6);
//     bool result2 = solution.isValidBST(root2);
//     // 应该输出 false

//     // 测试用例 3: 无效的 BST (相同值)
//     //      1
//     //      / \
//     //      1   1
//     TreeNode* root3 = new TreeNode(1);
//     root3->left = new TreeNode(1);
//     root3->right = new TreeNode(1);
//     bool result3 = solution.isValidBST(root3);
```

```
//    // 应该输出 false
```

```
//    return 0;  
// }
```

```
=====
```

文件: ValidateBinarySearchTree.java

```
=====
```

```
package class037;
```

```
import java.util.ArrayList;  
import java.util.List;
```

```
// LeetCode 98. Validate Binary Search Tree
```

```
// 题目链接: https://leetcode.cn/problems/validate-binary-search-tree/
```

```
// 题目描述: 给你一个二叉树的根节点 root , 判断其是否是一个有效的二叉搜索树。
```

```
// 有效二叉搜索树定义如下:
```

```
// 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。
```

```
//
```

```
// 解题思路:
```

```
// 1. 使用递归方法, 为每个节点设置上下界
```

```
// 2. 对于根节点, 上下界为无穷大和无穷小
```

```
// 3. 对于左子树, 上界更新为当前节点值
```

```
// 4. 对于右子树, 下界更新为当前节点值
```

```
// 5. 递归检查每个节点是否满足上下界约束
```

```
//
```

```
// 时间复杂度: O(n) - n 为树中节点的数量, 需要访问每个节点
```

```
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
```

```
// 是否为最优解: 是, 这是验证 BST 的标准方法
```

```
// 补充题目:
```

```
// 1. LeetCode 530. Minimum Absolute Difference in BST - 二叉搜索树的最小绝对差
```

```
// 2. LintCode 95. Validate Binary Search Tree - 与 LeetCode 98 相同
```

```
// 3. 牛客 NC47. 寻找第 K 大的元素
```

```
// 二叉搜索树的核心思想和技巧:
```

```
// 1. 利用 BST 的中序遍历结果是升序的特性解决许多问题
```

```
// 2. 对于验证 BST, 使用上下界递归可以有效处理边界情况
```

```
// 3. BST 的查找、插入、删除操作都可以在 O(h) 时间内完成
```

```
// 4. 使用 Morris 遍历可以实现 O(n) 时间和 O(1) 空间的中序遍历
```

```
// 5. 对于第 K 大/小元素问题，可以利用中序遍历的特性
```

```
public class ValidateBinarySearchTree {  
  
    // 二叉树节点定义  
    public static class TreeNode {  
        int val;  
        TreeNode left;  
        TreeNode right;  
  
        TreeNode() {}  
  
        TreeNode(int val) {  
            this.val = val;  
        }  
  
        TreeNode(int val, TreeNode left, TreeNode right) {  
            this.val = val;  
            this.left = left;  
            this.right = right;  
        }  
    }  
  
    // 提交如下的方法  
    public boolean isValidBST(TreeNode root) {  
        return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);  
    }  
  
    /**  
     * 递归验证 BST  
     * @param node 当前节点  
     * @param min 下界 (不包含)  
     * @param max 上界 (不包含)  
     * @return 是否为有效的 BST  
     */  
    private boolean isValidBST(TreeNode node, long min, long max) {  
        // 空节点是有效的 BST  
        if (node == null) {  
            return true;  
        }  
  
        // 检查当前节点是否满足上下界约束  
        if (node.val <= min || node.val >= max) {
```

```
    return false;
}

// 递归检查左右子树
// 左子树的上界更新为当前节点值
// 右子树的下界更新为当前节点值
return isValidBST(node.left, min, node.val) &&
      isValidBST(node.right, node.val, max);
}

// 测试用例
public static void main(String[] args) {
    ValidateBinarySearchTree solution = new ValidateBinarySearchTree();

    // 测试用例 1: 有效的 BST
    //      2
    //      / \
    //     1   3
    TreeNode root1 = new TreeNode(2);
    root1.left = new TreeNode(1);
    root1.right = new TreeNode(3);
    boolean result1 = solution.isValidBST(root1);
    System.out.println("测试用例 1 结果: " + result1); // 应该输出 true

    // 测试用例 2: 无效的 BST
    //      5
    //      / \
    //     1   4
    //      / \
    //     3   6
    TreeNode root2 = new TreeNode(5);
    root2.left = new TreeNode(1);
    root2.right = new TreeNode(4);
    root2.right.left = new TreeNode(3);
    root2.right.right = new TreeNode(6);
    boolean result2 = solution.isValidBST(root2);
    System.out.println("测试用例 2 结果: " + result2); // 应该输出 false

    // 测试用例 3: 无效的 BST (相同值)
    //      1
    //      / \
    //     1   1
    TreeNode root3 = new TreeNode(1);
```

```

root3.left = new TreeNode(1);
root3.right = new TreeNode(1);
boolean result3 = solution.isValidBST(root3);
System.out.println("测试用例 3 结果: " + result3); // 应该输出 false

// 测试补充题目 1: LeetCode 530
System.out.println("\n 测试 LeetCode 530 - 最小绝对差:");
//      4
//      / \
//      2   6
//      / \
//      1   3
TreeNode root4 = new TreeNode(4);
root4.left = new TreeNode(2);
root4.right = new TreeNode(6);
root4.left.left = new TreeNode(1);
root4.left.right = new TreeNode(3);
System.out.println("最小绝对差: " + solution.getMinimumDifference(root4)); // 应该输出 1

// 测试补充题目 3: 牛客 NC47
System.out.println("\n 测试牛客 NC47 - 寻找第 K 大的元素:");
//      3
//      / \
//      1   4
//      \n      //      2
TreeNode root5 = new TreeNode(3);
root5.left = new TreeNode(1);
root5.right = new TreeNode(4);
root5.left.right = new TreeNode(2);
int k = 1;
System.out.println("第" + k + "大的元素: " + solution.kthLargest(root5, k)); // 应该输出 4
}

// 补充题目 1: LeetCode 530. Minimum Absolute Difference in BST
// 题目链接: https://leetcode.cn/problems/minimum-absolute-difference-in-bst/
// 题目描述: 给你一个二叉搜索树的根节点 root，返回树中任意两不同节点值之间的最小绝对差。
//
// 解题思路:
// 1. 利用二叉搜索树的中序遍历结果是升序的特性
// 2. 在中序遍历过程中，计算当前节点与前一个节点的差值
// 3. 维护一个最小值变量，记录最小的绝对差值
//
// 时间复杂度: O(n) - 每个节点只访问一次

```

```

// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
// 是否为最优解: 是, 利用 BST 特性的高效解法
private int minDiff = Integer.MAX_VALUE;
private TreeNode prev = null;

public int getMinimumDifference(TreeNode root) {
    minDiff = Integer.MAX_VALUE;
    prev = null;
    inorderTraverse(root);
    return minDiff;
}

private void inorderTraverse(TreeNode node) {
    if (node == null) {
        return;
    }

    // 中序遍历: 先左子树
    inorderTraverse(node.left);

    // 处理当前节点
    if (prev != null) {
        minDiff = Math.min(minDiff, node.val - prev.val);
    }
    prev = node;

    // 中序遍历: 后右子树
    inorderTraverse(node.right);
}

// 补充题目 3: 牛客 NC47. 寻找第 K 大的元素
// 题目链接: https://www.nowcoder.com/practice/ef068f602dde4d28aab2b210e859150a
// 题目描述: 给定一棵二叉搜索树, 请找出其中第 k 大的节点值。
//
// 解题思路:
// 1. 利用二叉搜索树的特性: 中序遍历的结果是升序排列
// 2. 我们可以进行逆中序遍历(右-根-左), 这样结果就是降序排列
// 3. 在遍历过程中计数, 当计数达到 k 时, 即为第 k 大的元素
//
// 时间复杂度: O(n) - 最坏情况下需要遍历所有节点
// 空间复杂度: O(h) - h 为树的高度, 递归调用栈的深度
// 是否为最优解: 是, 利用 BST 特性的高效解法
private int count = 0;

```

```

private int result = 0;

public int kthLargest(TreeNode root, int k) {
    count = 0;
    result = 0;
    reverseInorder(root, k);
    return result;
}

private void reverseInorder(TreeNode node, int k) {
    if (node == null || count >= k) {
        return;
    }

    // 逆中序遍历: 先右子树
    reverseInorder(node.right, k);

    // 处理当前节点
    count++;
    if (count == k) {
        result = node.val;
        return;
    }

    // 逆中序遍历: 后左子树
    reverseInorder(node.left, k);
}

// 补充: Morris 中序遍历实现 (O(n)时间, O(1)空间)
public List<Integer> morrisInorder(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    TreeNode current = root;

    while (current != null) {
        if (current.left == null) {
            // 没有左子树, 直接访问当前节点, 然后移动到右子树
            result.add(current.val);
            current = current.right;
        } else {
            // 找到当前节点的前驱节点 (左子树的最右节点)
            TreeNode predecessor = current.left;
            while (predecessor.right != null && predecessor.right != current) {
                predecessor = predecessor.right;
            }
        }
    }
}

```

```

        }

        if (predecessor.right == null) {
            // 第一次访问，建立线索，连接前驱节点的右指针到当前节点
            predecessor.right = current;
            current = current.left;
        } else {
            // 已经访问过，断开线索，访问当前节点，然后移动到右子树
            predecessor.right = null;
            result.add(current.val);
            current = current.right;
        }
    }

}

return result;
}
}

```

=====

文件: ValidateBinarySearchTree.py

```

# LeetCode 98. Validate Binary Search Tree
# 题目链接: https://leetcode.cn/problems/validate-binary-search-tree/
# 题目描述: 给你一个二叉树的根节点 root ，判断其是否是一个有效的二叉搜索树。
# 有效二叉搜索树定义如下：
# 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。
#
# 解题思路：
# 1. 使用递归方法，为每个节点设置上下界
# 2. 对于根节点，上下界为无穷大和无穷小
# 3. 对于左子树，上界更新为当前节点值
# 4. 对于右子树，下界更新为当前节点值
# 5. 递归检查每个节点是否满足上下界约束
#
# 时间复杂度: O(n) - n 为树中节点的数量，需要访问每个节点
# 空间复杂度: O(h) - h 为树的高度，递归调用栈的深度
# 是否为最优解：是，这是验证 BST 的标准方法

```

```
from typing import Optional
```

```
import sys

# 二叉树节点定义
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        """
        验证二叉搜索树

        Args:
            root: 二叉树的根节点

        Returns:
            是否为有效的 BST
        """
        return self._isValidBST(root, -sys.maxsize - 1, sys.maxsize)

    def _isValidBST(self, node: Optional[TreeNode], min_val: int, max_val: int) -> bool:
        """
        递归验证 BST

        Args:
            node: 当前节点
            min_val: 下界 (不包含)
            max_val: 上界 (不包含)

        Returns:
            是否为有效的 BST
        """
        # 空节点是有效的 BST
        if node is None:
            return True

        # 检查当前节点是否满足上下界约束
        if node.val <= min_val or node.val >= max_val:
            return False

        # 递归检查左右子树
        return self._isValidBST(node.left, min_val, node.val) and self._isValidBST(node.right, node.val, max_val)
```

```
# 左子树的上界更新为当前节点值
# 右子树的下界更新为当前节点值
return (self._isValidBST(node.left, min_val, node.val) and
        self._isValidBST(node.right, node.val, max_val))

# 测试用例
def main():
    solution = Solution()

    # 测试用例 1: 有效的 BST
    #
    #      2
    #      / \
    #      1   3
    root1 = TreeNode(2)
    root1.left = TreeNode(1)
    root1.right = TreeNode(3)
    result1 = solution.isValidBST(root1)
    print("测试用例 1 结果:", result1)  # 应该输出 True

    # 测试用例 2: 无效的 BST
    #
    #      5
    #      / \
    #      1   4
    #      / \
    #      3   6
    root2 = TreeNode(5)
    root2.left = TreeNode(1)
    root2.right = TreeNode(4)
    root2.right.left = TreeNode(3)
    root2.right.right = TreeNode(6)
    result2 = solution.isValidBST(root2)
    print("测试用例 2 结果:", result2)  # 应该输出 False

    # 测试用例 3: 无效的 BST (相同值)
    #
    #      1
    #      / \
    #      1   1
    root3 = TreeNode(1)
    root3.left = TreeNode(1)
    root3.right = TreeNode(1)
    result3 = solution.isValidBST(root3)
    print("测试用例 3 结果:", result3)  # 应该输出 False
```

```
if __name__ == "__main__":
    main()
=====

```