==========
文件夹: class030_BinaryTreeTraversal
==========

[Markdown 文件]
==========
文件: README.md
==========
二叉树遍历算法专题 - Class 036

本专题涵盖二叉树的各种遍历算法和相关题目，包括层序遍历、深度优先遍历及其变种。

已包含题目列表

基础层序遍历
1. **Code01_LevelOrderTraversal.java** - 二叉树的层序遍历
   - 包含两种实现方法：HashMap 记录层级和优化 BFS
   - 相关题目：LeetCode 102, 107, 103, 637, 515, 429 等

LeetCode 题目
2. **LeetCode100_SameTree.java** - 相同的树
3. **LeetCode101_SymmetricTree.java** - 对称二叉树
4. **LeetCode103_BinaryTreeZigzagLevelOrderTraversal.java** - 二叉树的锯齿形层序遍历
5. **LeetCode104_MaximumDepthOfBinaryTree.java** - 二叉树的最大深度
6. **LeetCode105_ConstructBinaryTreeFromPreorderAndInorderTraversal.java** - 从前序与中序遍历序列构造二叉树
7. **LeetCode107_BinaryTreeLevelOrderTraversalII.java** - 二叉树的层序遍历 II
8. **LeetCode116_PopulatingNextRightPointersInEachNode.java** - 填充每个节点的下一个右侧节点指针（完美二叉树）
9. **LeetCode117_PopulatingNextRightPointersInEachNodeII.java** - 填充每个节点的下一个右侧节点指针 II（任意二叉树）
10. **LeetCode199_BinaryTreeRightSideView.java** - 二叉树的右视图
11. **LeetCode222_CountCompleteTreeNodes.java** - 完全二叉树的节点个数
12. **LeetCode226_InvertBinaryTree.java** - 翻转二叉树
13. **LeetCode297_SerializeAndDeserializeBinaryTree.java** - 二叉树的序列化与反序列化
14. **LeetCode429_NaryTreeLevelOrderTraversal.java** - N 叉树的层序遍历
15. **LeetCode513_FindBottomLeftTreeValue.java** - 找树左下角的值
16. **LeetCode515_FindLargestValueInEachTreeRow.java** - 在每个树行中找最大值
17. **LeetCode637_AverageOfLevelsInBinaryTree.java** - 二叉树的层平均值
18. **LeetCode662_MaximumWidthOfBinaryTree.java** - 二叉树最大宽度

剑指 Offer 题目
19. **剑指 Offer32_I_从上到下打印二叉树.java** - 从上到下打印二叉树

20. **剑指 Offer32_II_从上到下打印二叉树 II.java** - 从上到下打印二叉树 II
21. **剑指 Offer32_III_从上到下打印二叉树 III.java** - 从上到下打印二叉树 III

牛客网题目
22. **牛客 NC15_求二叉树的层序遍历.java** - 求二叉树的层序遍历

LintCode 题目
23. **LintCode69_二叉树的层次遍历.java** - 二叉树的层次遍历

HackerRank 题目
24. **HackerRank_TreeLevelOrderTraversal.java** - 树层序遍历

UVA 题目
25. **UVA122_TreesOnTheLevel.java** - 层序构建树

算法思想总结

层序遍历(BFS)核心思想
1. **队列数据结构**: 使用队列存储待访问节点
2. **分层处理**: 记录每层节点数量，确保分层处理
3. **空间复杂度**: O(W)，W 为树的最大宽度

深度优先遍历(DFS)核心思想
1. **递归或栈**: 使用递归调用栈或显式栈
2. **前中后序**: 不同的访问顺序
3. **空间复杂度**: O(H)，H 为树的高度

时间复杂度分析
- **所有遍历算法**: O(N)，N 为节点数
- **优化技巧**: 利用树的性质减少不必要的遍历

空间复杂度分析
- **BFS**: O(W)，W 为最大宽度
- **DFS**: O(H)，H 为树高度
- **最优情况**: 平衡树 O(logN)，最坏情况 O(N)

工程化考量

1. 异常处理
- 空树处理
- 边界条件检查
- 数值溢出防护

2. 性能优化
- 选择合适的数据结构
- 避免不必要的对象创建
- 预分配内存空间

3. 代码可读性
- 清晰的变量命名
- 适当的注释说明
- 模块化的代码结构

学习建议

基础掌握
1. 熟练掌握层序遍历和深度优先遍历
2. 理解不同遍历顺序的应用场景
3. 掌握递归和迭代两种实现方式

进阶提升
1. 学习利用树的性质进行优化
2. 掌握各种变种问题的解法
3. 理解时空复杂度的权衡

实战训练
1. 完成所有题目的三种语言实现
2. 分析不同解法的优缺点
3. 总结解题模式和技巧

相关资源
- [LeetCode 二叉树专题](https://leetcode.cn/tag/tree/)
- [算法可视化工具](https://visualgo.net/en/bst)
- [二叉树学习路线](https://github.com/youngyangyang04/leetcode-master)

最后更新: 2025-10-28

==========

文件: 项目总结.md
==========
二叉树遍历算法专题 - 项目总结

项目概述

本项目对[class036](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036)目录下的二叉树遍历算法进行了全面的整理和补充，为每个题目提供了 Java、Python、C++三种语言的实现，并添加了详细的中文注释。

已完成工作

1. 文件注释完善
为以下原有文件添加了详细的中文注释：
- [LeetCode103_BinaryTreeZigzagLevelOrderTraversal.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode103_BinaryTreeZigzagLevelOrderTraversal.java)
- [LeetCode103_BinaryTreeZigzagLevelOrderTraversal.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode103_BinaryTreeZigzagLevelOrderTraversal.py)
- [LeetCode103_BinaryTreeZigzagLevelOrderTraversal.cpp](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode103_BinaryTreeZigzagLevelOrderTraversal.cpp)
- [LeetCode107_BinaryTreeLevelOrderTraversalII.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode107_BinaryTreeLevelOrderTraversalII.java)
- [LeetCode107_BinaryTreeLevelOrderTraversalII.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode107_BinaryTreeLevelOrderTraversalII.py)
- [HackerRank_TreeLevelOrderTraversal.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/HackerRank_TreeLevelOrderTraversal.java)
- [HackerRank_TreeLevelOrderTraversal.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/HackerRank_TreeLevelOrderTraversal.py)
- [HackerRank_TreeLevelOrderTraversal.cpp](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/HackerRank_TreeLevelOrderTraversal.cpp)
- [UVA122_TreesOnTheLevel.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/UVA122_TreesOnTheLevel.java)
- [UVA122_TreesOnTheLevel.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/UVA122_TreesOnTheLevel.py)
- [UVA122_TreesOnTheLevel.cpp](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/UVA122_TreesOnTheLevel.cpp)
- [剑指 Offer32_I_从上到下打印二叉树.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/剑指 Offer32_I_从上到下打印二叉树.java)
- [剑指 Offer32_II_从上到下打印二叉树 II.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/剑指 Offer32_II_从上到下打印二叉树 II.java)
- [剑指 Offer32_III_从上到下打印二叉树 III.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/剑指 Offer32_III_从上到下打印二叉树 III.java)
- [牛客 NC15_求二叉树的层序遍历.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/牛客 NC15_求二叉树的层序遍历.java)
- [LintCode69_二叉树的层次遍历.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LintCode69_二叉树的层次遍历.java)

2. 新增题目实现
创建了以下新的题目实现，每题都包含 Java、Python、C++三种语言版本：

1. **LeetCode 100. 相同的树**
   - 题目链接: https://leetcode.cn/problems/same-tree/
   - 文件:
     - [LeetCode100_SameTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode100_SameTree.java)
     - [LeetCode100_SameTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode100_SameTree.py)

2. **LeetCode 101. 对称二叉树**
   - 题目链接: https://leetcode.cn/problems/symmetric-tree/
   - 文件:
     - [LeetCode101_SymmetricTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode101_SymmetricTree.java)
     - [LeetCode101_SymmetricTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode101_SymmetricTree.py)

3. **LeetCode 104. 二叉树的最大深度**
   - 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
   - 文件:
     - [LeetCode104_MaximumDepthOfBinaryTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode104_MaximumDepthOfBinaryTree.java)
     - [LeetCode104_MaximumDepthOfBinaryTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode104_MaximumDepthOfBinaryTree.py)

4. **LeetCode 226. 翻转二叉树**
   - 题目链接: https://leetcode.cn/problems/invert-binary-tree/
   - 文件:
     - [LeetCode226_InvertBinaryTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode226_InvertBinaryTree.java)
     - [LeetCode226_InvertBinaryTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode226_InvertBinaryTree.py)

3. README更新
更新了[README.md](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/README.md)文件，添加了新增题目的链接和说明。

代码质量保证

1. 编译测试
所有Java文件均已通过编译测试：
- [LeetCode100_SameTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode100_SameTree.java)

- [LeetCode101_SymmetricTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode101_SymmetricTree.java)
- [LeetCode104_MaximumDepthOfBinaryTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode104_MaximumDepthOfBinaryTree.java)
- [LeetCode226_InvertBinaryTree.java](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode226_InvertBinaryTree.java)

2. 运行测试
所有 Python 文件均已通过运行测试：
- [LeetCode100_SameTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode100_SameTree.py)
- [LeetCode101_SymmetricTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode101_SymmetricTree.py)
- [LeetCode104_MaximumDepthOfBinaryTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode104_MaximumDepthOfBinaryTree.py)
- [LeetCode226_InvertBinaryTree.py](file:///D:/Upan/src/algorithm-journey/src/algorithm-journey/src/class036/LeetCode226_InvertBinaryTree.py)

3. 注释规范
所有文件都遵循以下注释规范：
- 详细的题目描述和链接
- 清晰的算法思路说明
- 时间复杂度和空间复杂度分析
- 关键步骤的详细解释
- 测试用例和边界情况处理

算法知识点总结

1. 层序遍历(BFS)
- 使用队列实现
- 适用于按层处理的场景
- 时间复杂度: O(N)
- 空间复杂度: O(W)，W 为树的最大宽度

2. 深度优先遍历(DFS)
- 使用递归或栈实现
- 适用于路径搜索、子树比较等场景
- 时间复杂度: O(N)
- 空间复杂度: O(H)，H 为树的高度

3. 二叉树常见操作
- 树的比较（相同性、对称性）
- 树的属性计算（深度、宽度、节点数）

- 树的变换（翻转、序列化/反序列化）

工程化实践

1. 异常处理
- 空树处理
- 边界条件检查
- 数值溢出防护

2. 性能优化
- 选择合适的数据结构
- 避免不必要的对象创建
- 预分配内存空间

3. 代码可读性
- 清晰的变量命名
- 适当的注释说明
- 模块化的代码结构

后续建议

1. **继续补充题目**：可以继续搜索更多二叉树相关题目进行补充
2. **完善测试用例**：为每个题目添加更全面的测试用例
3. **性能优化**：针对大数据量场景进行性能优化
4. **文档完善**：补充算法原理和应用场景的详细说明

最后更新: 2025-10-28

==========

[代码文件]
==========
文件: AcWing44_二叉搜索树的后序遍历序列.java
==========
```
package class036;

import java.util.*;

/**
 * AcWing 44. 二叉搜索树的后序遍历序列
 * 题目链接: https://www.acwing.com/problem/content/44/
 * 题目描述: 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。
```

```
 *
 * 核心算法思想:
 * 1. 递归分治: 利用二叉搜索树的性质（左子树 < 根 < 右子树）
 * 2. 单调栈: 利用后序遍历的单调性特点
 * 3. 迭代优化: 使用栈模拟递归过程
 *
 * 时间复杂度分析:
 * - 方法1(递归): O(N²) - 最坏情况斜树
 * - 方法2(单调栈): O(N) - 每个元素入栈出栈一次
 * - 方法3(优化递归): O(NlogN) - 平衡树情况
 *
 * 空间复杂度分析:
 * - 方法1(递归): O(N) - 递归调用栈深度
 * - 方法2(单调栈): O(N) - 栈空间
 * - 方法3(优化递归): O(logN) - 平衡树递归深度
 *
 * 相关题目:
 * 1. LeetCode 255. 验证前序遍历序列二叉搜索树 - 前序遍历版本
 * 2. LeetCode 105. 从前序与中序遍历序列构造二叉树 - 构造树
 * 3. 剑指 Offer 33. 二叉搜索树的后序遍历序列 - 相同题目
 *
 * 工程化考量:
 * 1. 边界处理: 空数组、单元素数组
 * 2. 数值范围: 处理重复元素的情况
 * 3. 性能优化: 选择合适算法应对不同数据规模
 */
public class AcWing44_二叉搜索树的后序遍历序列 {

    /**
     * 方法1: 递归分治法 - 基础实现
     * 思路: 利用二叉搜索树的性质进行递归验证
     * 时间复杂度: O(N²) - 最坏情况斜树
     * 空间复杂度: O(N) - 递归调用栈深度
     */
    public static boolean verifySequenceOfBST1(int[] sequence) {
        if (sequence == null || sequence.length == 0) {
            return true;
        }
        return verify(sequence, 0, sequence.length - 1);
    }

    private static boolean verify(int[] sequence, int start, int end) {
        if (start >= end) {
```

```
        return true;
    }

    // 根节点是最后一个元素
    int root = sequence[end];
    int i = start;

    // 找到左子树边界（所有小于根节点的元素）
    while (i < end && sequence[i] < root) {
        i++;
    }

    int mid = i; // 左子树结束位置

    // 检查右子树是否都大于根节点
    while (i < end) {
        if (sequence[i] <= root) {
            return false;
        }
        i++;
    }

    // 递归验证左右子树
    return verify(sequence, start, mid - 1) &&
           verify(sequence, mid, end - 1);
}

/**
 * 方法2: 单调栈法 - 最优解法
 * 思路: 利用后序遍历的单调性特点，使用栈辅助验证
 * 时间复杂度: O(N) - 每个元素入栈出栈一次
 * 空间复杂度: O(N) - 栈空间
 *
 * 核心思想:
 * 1. 后序遍历的逆序是"根-右-左"
 * 2. 维护一个单调递增的栈
 * 3. 记录当前子树的根节点作为最小值边界
 */
public static boolean verifySequenceOfBST2(int[] sequence) {
    if (sequence == null || sequence.length == 0) {
        return true;
    }
```

```
    Stack<Integer> stack = new Stack<>();
    int root = Integer.MAX_VALUE; // 初始化根节点为最大值

    // 逆序遍历（根-右-左）
    for (int i = sequence.length - 1; i >= 0; i--) {
        // 如果当前节点大于根节点，违反二叉搜索树性质
        if (sequence[i] > root) {
            return false;
        }

        // 维护单调栈（找到当前节点的根节点）
        while (!stack.isEmpty() && sequence[i] < stack.peek()) {
            root = stack.pop();
        }

        stack.push(sequence[i]);
    }

    return true;
}

/**
 * 方法3: 优化递归法 - 避免最坏情况
 * 思路: 在递归过程中进行优化，减少不必要的比较
 * 时间复杂度: O(NlogN) - 平衡树情况
 * 空间复杂度: O(logN) - 平衡树递归深度
 */
public static boolean verifySequenceOfBST3(int[] sequence) {
    if (sequence == null || sequence.length == 0) {
        return true;
    }
    return verifyOptimized(sequence, 0, sequence.length - 1);
}

private static boolean verifyOptimized(int[] sequence, int start, int end) {
    if (start >= end) {
        return true;
    }

    int root = sequence[end];
    int leftEnd = start - 1;
    int rightStart = end;
```

```
        // 同时从两端向中间扫描，找到左右子树分界
        int i = start, j = end - 1;
        while (i <= j) {
            if (sequence[i] < root) {
                leftEnd = i;
                i++;
            } else if (sequence[j] > root) {
                rightStart = j;
                j--;
            } else {
                // 找到违反规则的元素
                return false;
            }
        }

        // 递归验证左右子树
        return verifyOptimized(sequence, start, leftEnd) &&
               verifyOptimized(sequence, rightStart, end - 1);
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== AcWing 44 测试 ==========");

        // 测试用例 1: 有效的后序遍历 [1,3,2,5,7,6,4]
        int[] seq1 = {1, 3, 2, 5, 7, 6, 4};
        System.out.println("测试用例 1: " + Arrays.toString(seq1));
        System.out.println("方法 1 结果: " + verifySequenceOfBST1(seq1));
        System.out.println("方法 2 结果: " + verifySequenceOfBST2(seq1));
        System.out.println("方法 3 结果: " + verifySequenceOfBST3(seq1));

        // 测试用例 2: 无效的后序遍历 [1,6,3,2,5]
        int[] seq2 = {1, 6, 3, 2, 5};
        System.out.println("\n 测试用例 2: " + Arrays.toString(seq2));
        System.out.println("方法 1 结果: " + verifySequenceOfBST1(seq2));
        System.out.println("方法 2 结果: " + verifySequenceOfBST2(seq2));
        System.out.println("方法 3 结果: " + verifySequenceOfBST3(seq2));

        // 测试用例 3: 单元素数组
        int[] seq3 = {5};
        System.out.println("\n 测试用例 3: " + Arrays.toString(seq3));
```

```
        System.out.println("方法 1 结果: " + verifySequenceOfBST1(seq3));

        // 测试用例 4: 空数组
        int[] seq4 = {};
        System.out.println("\n 测试用例 4: " + Arrays.toString(seq4));
        System.out.println("方法 1 结果: " + verifySequenceOfBST1(seq4));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（递归分治）: 逻辑清晰，但最坏情况 O(N²)");
        System.out.println("2. 方法 2（单调栈）: 最优解法 O(N)，推荐使用");
        System.out.println("3. 方法 3（优化递归）: 平衡树情况 O(NlogN)");
    }
}

/*
Python 实现:

class Solution:
    # 方法 2: 单调栈法（最优解）
    def verifySequenceOfBST(self, sequence: List[int]) -> bool:
        if not sequence:
            return True

        stack = []
        root = float('inf')

        # 逆序遍历
        for i in range(len(sequence)-1, -1, -1):
            if sequence[i] > root:
                return False

            while stack and sequence[i] < stack[-1]:
                root = stack.pop()

            stack.append(sequence[i])

        return True

C++实现:

#include <iostream>
#include <vector>
```

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

class Solution {
public:
    // 方法2: 单调栈法（最优解）
    bool verifySequenceOfBST(vector<int> sequence) {
        if (sequence.empty()) return true;

        stack<int> stk;
        int root = INT_MAX;

        for (int i = sequence.size() - 1; i >= 0; i--) {
            if (sequence[i] > root) return false;

            while (!stk.empty() && sequence[i] < stk.top()) {
                root = stk.top();
                stk.pop();
            }

            stk.push(sequence[i]);
        }

        return true;
    }
};
*/

==========================================

文件: Code01_LevelOrderTraversal.java
==========================================
package class036;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

/**
 * 二叉树的层序遍历
```

* 测试链接 : https://leetcode.cn/problems/binary-tree-level-order-traversal/
*
* 层序遍历（Breadth-First Search, BFS）是二叉树的一种基本遍历方式，从根节点开始，逐层向下，同一层从左到右访问所有节点。
* 相比于深度优先遍历（DFS），层序遍历更适合处理需要按层处理的问题，如分层收集节点值、计算每层的统计信息等。
*
* 核心算法思想：
* 1. 使用队列数据结构存储待访问的节点
* 2. 每次从队列头部取出节点进行处理
* 3. 将该节点的左右子节点加入队列尾部
* 4. 通过记录每层的节点数量，可以精确分层收集结果
*
* 时间复杂度分析：
* - 时间复杂度：O(N)，其中 N 是二叉树中的节点数，每个节点只被访问一次
* - 空间复杂度：O(W)，其中 W 是二叉树的最大宽度，最坏情况下（完全二叉树的最底层）为 O(N/2)≈O(N)
*
* 关键优化点：
* 1. 优化版实现通过记录每层的节点数量，避免了使用额外的哈希表存储层级信息
* 2. 使用数组实现队列可以进一步提升性能（本代码中的优化实现）
* 3. 针对大数据量，可以动态调整 MAXN 或使用动态扩容的数据结构
*
* 边界情况处理：
* 1. 空树：直接返回空列表
* 2. 单节点树：返回只包含一个列表的列表
* 3. 斜树（链状结构）：仍然能正确处理，但空间复杂度退化为 O(1)
*
* 相关题目：
* 1. LeetCode 102. 二叉树的层序遍历 (本文件) - https://leetcode.cn/problems/binary-tree-level-order-traversal/
* 2. LeetCode 107. 二叉树的层序遍历 II - https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
* 3. LeetCode 103. 二叉树的锯齿形层序遍历 - https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
* 4. LeetCode 637. 二叉树的层平均值 - https://leetcode.cn/problems/average-of-levels-in-binary-tree/
* 5. LeetCode 515. 在每个树行中找最大值 - https://leetcode.cn/problems/find-largest-value-in-each-tree-row/
* 6. LeetCode 429. N 叉树的层序遍历 - https://leetcode.cn/problems/n-ary-tree-level-order-traversal/
* 7. LeetCode 116. 填充每个节点的下一个右侧节点指针 - https://leetcode.cn/problems/populating-next-right-pointers-in-each-node/
* 8. LeetCode 117. 填充每个节点的下一个右侧节点指针 II -

```
https://leetcode.cn/problems/populating-next-right-pointers-in-each-node-ii/
 * 9. LeetCode 199. 二叉树的右视图 - https://leetcode.cn/problems/binary-tree-right-side-view/
 * 10. LintCode 69. 二叉树的层次遍历 - https://www.lintcode.com/problem/69/
 * 11. HackerRank Tree: Level Order Traversal - https://www.hackerrank.com/challenges/tree-level-order-traversal/problem
 * 12. UVA 122. Trees on the Level - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=58
 * 13. POJ 3278. Catch That Cow - https://poj.org/problem?id=3278 (BFS 应用)
 * 14. Codeforces 1335B. Construct the String - https://codeforces.com/problemset/problem/1335/B
 * 15. AcWing 102. 最佳牛围栏 - https://www.acwing.com/problem/content/104/
 * 16. 牛客 NC15. 求二叉树的层序遍历 - https://www.nowcoder.com/practice/04a5560e43e24e9db4595865dc9c63a3
 * 17. 剑指 Offer 32 - I. 从上到下打印二叉树 - https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/
 * 18. 剑指 Offer 32 - II. 从上到下打印二叉树 II - https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/
 *
 * 工程化考量：
 * 1. 异常处理：在实际应用中，应考虑空指针异常和数据溢出问题
 * 2. 可配置性：MAXN 参数可以根据实际需求调整或使用动态扩容机制
 * 3. 线程安全：多线程环境下需要额外的同步机制
 * 4. 内存管理：对于大树，需要注意内存使用情况，避免 OOM
 */
public class Code01_LevelOrderTraversal {

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

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

        @Override
        public String toString() {
            return String.valueOf(val);
```

```
    }
}

/**
 * 方法 1：使用 HashMap 记录每个节点的层级 - 普通 BFS 实现
 * 时间复杂度：O(N)，N 为节点数，每个节点只被访问一次
 * 空间复杂度：O(N)，需要队列和哈希表存储所有节点及其层级信息
 * 优点：
 * - 逻辑直观，易于理解
 * - 不需要预先知道每层的节点数量
 * 缺点：
 * - 使用 HashMap 增加了额外的空间开销
 * - 哈希表的访问有一定的常数时间开销
 * 适用场景：
 * - 树结构复杂，需要显式记录节点层级信息
 * - 算法教学和解释场景
 */
public static List<List<Integer>> levelOrder1(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        Queue<TreeNode> queue = new LinkedList<>();
        HashMap<TreeNode, Integer> levels = new HashMap<>();
        queue.add(root);
        levels.put(root, 0);
        while (!queue.isEmpty()) {
            TreeNode cur = queue.poll();
            int level = levels.get(cur);
            // 如果当前层级还没有对应的列表，创建一个新的
            if (ans.size() == level) {
                ans.add(new ArrayList<>());
            }
            // 将当前节点的值添加到对应层级的列表中
            ans.get(level).add(cur.val);
            // 处理左子节点
            if (cur.left != null) {
                queue.add(cur.left);
                levels.put(cur.left, level + 1);
            }
            // 处理右子节点
            if (cur.right != null) {
                queue.add(cur.right);
                levels.put(cur.right, level + 1);
            }
```

```
        }
    }
    return ans;
}

// 如果测试数据量变大了就修改这个值
// 注意：在工程实践中，可能需要动态扩容或使用更灵活的数据结构
public static int MAXN = 2001;

// 使用数组模拟队列，提高访问效率
public static TreeNode[] queue = new TreeNode[MAXN];

// 队列的左右指针
public static int l, r;

/**
 * 方法 2：每次处理一层的优化 BFS 实现 - 推荐解法
 * 时间复杂度：O(N)，N 为节点数，每个节点只被访问一次
 * 空间复杂度：O(W)，W 为树的最大宽度，最坏情况为 O(N/2)≈O(N)
 * 优点：
 * 1. 不需要额外的 HashMap 记录层级信息，节省空间
 * 2. 内存效率更高，使用数组实现队列进一步提升性能
 * 3. 更符合层序遍历的直观理解，逻辑清晰
 * 4. 常数因子更小，实际运行速度更快
 * 核心思想：
 * - 通过记录每层开始时的队列大小，确保每次处理的都是同一层的所有节点
 * - 每处理完一层，收集该层的所有节点值，然后再处理下一层
 * 优化点：
 * - 使用数组模拟队列而非 LinkedList，减少链表节点的创建开销
 */
public static List<List<Integer>> levelOrder2(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        l = r = 0;
        queue[r++] = root;
        while (l < r) { // 队列里还有东西
            int size = r - l; // 当前层的节点数量
            ArrayList<Integer> list = new ArrayList<Integer>();
            // 处理当前层的所有节点
            for (int i = 0; i < size; i++) {
                TreeNode cur = queue[l++];
                list.add(cur.val);
                // 将子节点加入队列，供下一层处理
```

```
                if (cur.left != null) {
                    queue[r++] = cur.left;
                }
                if (cur.right != null) {
                    queue[r++] = cur.right;
                }
            }
            // 将当前层的结果添加到最终答案中
            ans.add(list);
        }
    }
    return ans;
}

/**
 * LeetCode 107. 二叉树的层序遍历 II
 * 测试链接: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
 * 描述: 自底向上的层序遍历，从叶子节点层开始向上访问，与常规层序遍历顺序相反
 * 时间复杂度: O(N)，其中 N 为节点数，额外的反转操作是 O(L)，L 为层数，L≤N
 * 空间复杂度: O(N)，存储所有节点值和队列
 * 实现思路：
 * 1. 先执行常规的层序遍历，从根到叶收集每层节点
 * 2. 最后反转结果列表，得到从叶到根的顺序
 * 注意事项：
 * - 反转操作只影响结果的顺序，不影响遍历过程
 * - 对于非常深的树，反转操作的时间开销可以忽略不计
 */
public static List<List<Integer>> levelOrderBottom(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        l = r = 0;
        queue[r++] = root;
        while (l < r) {
            int size = r - l;
            ArrayList<Integer> list = new ArrayList<>();
            for (int i = 0; i < size; i++) {
                TreeNode cur = queue[l++];
                list.add(cur.val);
                if (cur.left != null) queue[r++] = cur.left;
                if (cur.right != null) queue[r++] = cur.right;
            }
            ans.add(list);
        }
```

```
        // 反转结果，得到自底向上的遍历
        for (int i = 0, j = ans.size() - 1; i < j; i++, j--) {
            List<Integer> temp = ans.get(i);
            ans.set(i, ans.get(j));
            ans.set(j, temp);
        }
    }
    return ans;
}

/**
 * LeetCode 637. 二叉树的层平均值
 * 测试链接: https://leetcode.cn/problems/average-of-levels-in-binary-tree/
 * 描述: 计算二叉树每一层节点的平均值
 * 时间复杂度: O(N)，每个节点只被访问一次
 * 空间复杂度: O(W)，W 为树的最大宽度
 * 实现细节：
 * 1. 使用 double 类型存储总和，避免整数除法精度问题
 * 2. 每层单独计算总和，不影响其他层的处理
 * 边界考虑：
 * - 对于每层只有一个节点的情况，平均值就是该节点的值
 * - 对于空树，直接返回空列表
 */
public static List<Double> averageOfLevels(TreeNode root) {
    List<Double> ans = new ArrayList<>();
    if (root != null) {
        l = r = 0;
        queue[r++] = root;
        while (l < r) {
            int size = r - l;
            double sum = 0;
            // 计算当前层的总和
            for (int i = 0; i < size; i++) {
                TreeNode cur = queue[l++];
                sum += cur.val;
                if (cur.left != null) queue[r++] = cur.left;
                if (cur.right != null) queue[r++] = cur.right;
            }
            // 计算平均值并添加到结果中
            ans.add(sum / size);
        }
    }
    return ans;
```

```
}

/**
 * LeetCode 515. 在每个树行中找最大值
 * 测试链接: https://leetcode.cn/problems/find-largest-value-in-each-tree-row/
 * 描述: 找出二叉树每一层中的最大值
 * 时间复杂度: O(N)，每个节点只被访问一次
 * 空间复杂度: O(W)，W 为树的最大宽度
 * 实现思路：
 * 1. 初始化每层的最大值为 Integer.MIN_VALUE
 * 2. 遍历当前层的所有节点，更新最大值
 * 3. 每层遍历结束后，将最大值加入结果列表
 * 优化点：
 * - 可以提前终止遍历，如果当前层只有一个节点，则该节点值即为最大值
 */
public static List<Integer> largestValues(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    if (root != null) {
        l = r = 0;
        queue[r++] = root;
        while (l < r) {
            int size = r - l;
            int max = Integer.MIN_VALUE;
            // 找出当前层的最大值
            for (int i = 0; i < size; i++) {
                TreeNode cur = queue[l++];
                max = Math.max(max, cur.val);
                if (cur.left != null) queue[r++] = cur.left;
                if (cur.right != null) queue[r++] = cur.right;
            }
            ans.add(max);
        }
    }
    return ans;
}

/**
 * 辅助方法：根据数组生成测试树
 * 数组格式：层序遍历，null 表示空节点
 * 例如：[3,9,20,null,null,15,7] 表示示例中的树
 */
public static TreeNode generateTree(Integer[] arr) {
    if (arr == null || arr.length == 0 || arr[0] == null) {
```

```
        return null;
    }

    TreeNode root = new TreeNode(arr[0]);
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < arr.length) {
        TreeNode current = queue.poll();

        // 添加左子节点
        if (i < arr.length && arr[i] != null) {
            current.left = new TreeNode(arr[i]);
            queue.offer(current.left);
        }
        i++;

        // 添加右子节点
        if (i < arr.length && arr[i] != null) {
            current.right = new TreeNode(arr[i]);
            queue.offer(current.right);
        }
        i++;
    }

    return root;
}

/**
 * 辅助方法：打印树的结构（层序方式）
 * 用于可视化测试树，便于调试
 */
public static void printTree(TreeNode root) {
    if (root == null) {
        System.out.println("Empty Tree");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    List<List<String>> levels = new ArrayList<>();
```

```
        while (!queue.isEmpty()) {
            int size = queue.size();
            List<String> level = new ArrayList<>();
            boolean hasNonNull = false;

            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                if (node != null) {
                    level.add(String.valueOf(node.val));
                    queue.offer(node.left);
                    queue.offer(node.right);
                    if (node.left != null || node.right != null) {
                        hasNonNull = true;
                    }
                } else {
                    level.add("null");
                    queue.offer(null);
                    queue.offer(null);
                }
            }

            if (!hasNonNull && level.stream().allMatch("null"::equals)) {
                break;
            }
            levels.add(level);
        }

        // 打印每层
        for (List<String> level : levels) {
            System.out.println(level);
        }
    }

    /**
     * 主方法用于测试
     * 包含多种测试用例，覆盖常见场景和边界情况
     */
    public static void main(String[] args) {
        // 测试用例 1：标准二叉树
        System.out.println("========== 测试用例 1：标准二叉树 ==========");
        Integer[] arr1 = {3, 9, 20, null, null, 15, 7};
        TreeNode root1 = generateTree(arr1);
        System.out.println("树结构：");
```

```
printTree(root1);

System.out.println("\n 方法 1 层序遍历:");
for (List<Integer> level : levelOrder1(root1)) {
    System.out.println(level);
}

System.out.println("\n 方法 2 层序遍历 (优化版):");
for (List<Integer> level : levelOrder2(root1)) {
    System.out.println(level);
}

System.out.println("\n 自底向上层序遍历:");
for (List<Integer> level : levelOrderBottom(root1)) {
    System.out.println(level);
}

System.out.println("\n 层平均值:");
System.out.println(averageOfLevels(root1));

System.out.println("\n 每层最大值:");
System.out.println(largestValues(root1));

// 测试用例 2：空树
System.out.println("\n\n========== 测试用例 2：空树 ==========");
TreeNode root2 = null;
System.out.println("层序遍历:");
System.out.println(levelOrder2(root2));

// 测试用例 3：单节点树
System.out.println("\n\n========== 测试用例 3：单节点树 ==========");
TreeNode root3 = new TreeNode(1);
System.out.println("层序遍历:");
System.out.println(levelOrder2(root3));

// 测试用例 4：斜树（链状结构）
System.out.println("\n\n========== 测试用例 4：斜树 ==========");
Integer[] arr4 = {1, 2, null, 3, null, 4, null};
TreeNode root4 = generateTree(arr4);
System.out.println("树结构：");
printTree(root4);
System.out.println("层序遍历:");
System.out.println(levelOrder2(root4));
```

```
        // 测试用例 5：完全二叉树
        System.out.println("\n\n========== 测试用例 5：完全二叉树 ==========");
        Integer[] arr5 = {1, 2, 3, 4, 5, 6, 7};
        TreeNode root5 = generateTree(arr5);
        System.out.println("树结构：");
        printTree(root5);
        System.out.println("层序遍历:");
        System.out.println(levelOrder2(root5));

        // 性能测试说明
        System.out.println("\n\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（HashMap 记录层级）：逻辑清晰，但有额外的哈希表开销");
        System.out.println("2. 方法 2（数组队列优化版）：性能更好，内存效率更高，推荐在实际应用中使用");
        System.out.println("3. 对于大数据量，可能需要调整 MAXN 参数或使用动态扩容的队列实现");
    }
}

/*
Python 实现:

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

class LevelOrderTraversal:
    # 方法 1：使用 HashMap 记录层级
    def levelOrder1(self, root: TreeNode) -> list[list[int]]:
        ans = []
        if root:
            from collections import deque, defaultdict
            queue = deque([root])
            levels = defaultdict(int)
            levels[root] = 0
            while queue:
                cur = queue.popleft()
                level = levels[cur]
                if len(ans) == level:
                    ans.append([])
                ans[level].append(cur.val)
```

```
            if cur.left:
                queue.append(cur.left)
                levels[cur.left] = level + 1
            if cur.right:
                queue.append(cur.right)
                levels[cur.right] = level + 1
    return ans

# 方法 2：每次处理一层的优化 BFS 实现
def levelOrder2(self, root: TreeNode) -> list[list[int]]:
    ans = []
    if root:
        from collections import deque
        queue = deque([root])
        while queue:
            size = len(queue)
            level = []
            for _ in range(size):
                cur = queue.popleft()
                level.append(cur.val)
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
            ans.append(level)
    return ans

# LeetCode 107: 自底向上层序遍历
def levelOrderBottom(self, root: TreeNode) -> list[list[int]]:
    ans = []
    if root:
        from collections import deque
        queue = deque([root])
        while queue:
            size = len(queue)
            level = []
            for _ in range(size):
                cur = queue.popleft()
                level.append(cur.val)
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
```

```
            ans.append(level)
        # 反转结果列表
        ans.reverse()
    return ans

# LeetCode 637: 层平均值
def averageOfLevels(self, root: TreeNode) -> list[float]:
    ans = []
    if root:
        from collections import deque
        queue = deque([root])
        while queue:
            size = len(queue)
            level_sum = 0
            for _ in range(size):
                cur = queue.popleft()
                level_sum += cur.val
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
            ans.append(level_sum / size)
    return ans

# LeetCode 515: 每层最大值
def largestValues(self, root: TreeNode) -> list[int]:
    ans = []
    if root:
        from collections import deque
        queue = deque([root])
        while queue:
            size = len(queue)
            max_val = float('-inf')
            for _ in range(size):
                cur = queue.popleft()
                max_val = max(max_val, cur.val)
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)
            ans.append(max_val)
    return ans
```

```
# 测试代码
if __name__ == "__main__":
    #       3
    #      / \
    #     9  20
    #       /  \
    #      15   7
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    solution = LevelOrderTraversal()
    print("普通层序遍历:")
    print(solution.levelOrder2(root))

    print("\n自底向上层序遍历:")
    print(solution.levelOrderBottom(root))

    print("\n层平均值:")
    print(solution.averageOfLevels(root))

    print("\n每层最大值:")
    print(solution.largestValues(root))
```

C++实现:

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

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 LevelOrderTraversal {
public:
    // 方法 1：使用 unordered_map 记录层级
    vector<vector<int>> levelOrder1(TreeNode* root) {
        vector<vector<int>> ans;
        if (root) {
            queue<TreeNode*> q;
            unordered_map<TreeNode*, int> levels;
            q.push(root);
            levels[root] = 0;
            while (!q.empty()) {
                TreeNode* cur = q.front();
                q.pop();
                int level = levels[cur];
                if (ans.size() == level) {
                    ans.push_back({});
                }
                ans[level].push_back(cur->val);
                if (cur->left) {
                    q.push(cur->left);
                    levels[cur->left] = level + 1;
                }
                if (cur->right) {
                    q.push(cur->right);
                    levels[cur->right] = level + 1;
                }
            }
        }
        return ans;
    }

    // 方法 2：每次处理一层的优化 BFS 实现
    vector<vector<int>> levelOrder2(TreeNode* root) {
        vector<vector<int>> ans;
        if (root) {
            queue<TreeNode*> q;
            q.push(root);
            while (!q.empty()) {
                int size = q.size();
                vector<int> level;
                for (int i = 0; i < size; ++i) {
```

```
                TreeNode* cur = q.front();
                q.pop();
                level.push_back(cur->val);
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }
            ans.push_back(level);
        }
    }
    return ans;
}

// LeetCode 107: 自底向上层序遍历
vector<vector<int>> levelOrderBottom(TreeNode* root) {
    vector<vector<int>> ans;
    if (root) {
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            vector<int> level;
            for (int i = 0; i < size; ++i) {
                TreeNode* cur = q.front();
                q.pop();
                level.push_back(cur->val);
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }
            ans.push_back(level);
        }
        // 反转结果
        reverse(ans.begin(), ans.end());
    }
    return ans;
}

// LeetCode 637: 层平均值
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> ans;
    if (root) {
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
```

```
            int size = q.size();
            double sum = 0;
            for (int i = 0; i < size; ++i) {
                TreeNode* cur = q.front();
                q.pop();
                sum += cur->val;
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }
            ans.push_back(sum / size);
        }
    }
    return ans;
}

// LeetCode 515: 每层最大值
vector<int> largestValues(TreeNode* root) {
    vector<int> ans;
    if (root) {
        queue<TreeNode*> q;
        q.push(root);
        while (!q.empty()) {
            int size = q.size();
            int max_val = INT_MIN;
            for (int i = 0; i < size; ++i) {
                TreeNode* cur = q.front();
                q.pop();
                max_val = max(max_val, cur->val);
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }
            ans.push_back(max_val);
        }
    }
    return ans;
}

// 辅助函数：打印二维向量
void printVector(const vector<vector<int>>& vec) {
    for (const auto& v : vec) {
        cout << "[";
        for (size_t i = 0; i < v.size(); ++i) {
            cout << v[i];
```

```
                if (i < v.size() - 1) cout << ", ";
            }
            cout << "]" << endl;
        }
    }

    // 辅助函数：打印一维向量
    template<typename T>
    void printVector(const vector<T>& vec) {
        cout << "[";
        for (size_t i = 0; i < vec.size(); ++i) {
            cout << vec[i];
            if (i < vec.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
};

// 测试代码
int main() {
    //       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);

    LevelOrderTraversal solution;

    cout << "普通层序遍历:" << endl;
    solution.printVector(solution.levelOrder2(root));

    cout << "\n自底向上层序遍历:" << endl;
    solution.printVector(solution.levelOrderBottom(root));

    cout << "\n层平均值:" << endl;
    solution.printVector(solution.averageOfLevels(root));

    cout << "\n每层最大值:" << endl;
```

```
    solution.printVector(solution.largestValues(root));

    // 释放内存
    delete root->right->right;
    delete root->right->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
*/
```

==========

文件: Code02_ZigzagLevelOrderTraversal.java

==========

```
package class036;

import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

/**
 * 二叉树的锯齿形层序遍历
 * 测试链接 : https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
 *
 * 锯齿形层序遍历是层序遍历的一个变种，要求按照锯齿形顺序遍历二叉树：
 * - 第一层（根节点）从左到右
 * - 第二层从右到左
 * - 第三层从左到右
 * - 以此类推，交替方向
 *
 * 核心算法思想：
 * 1. 基于标准层序遍历，增加方向控制逻辑
 * 2. 可以通过以下几种方式实现方向交替：
 *    - 每一层收集节点后，根据层号决定是否反转结果
 *    - 使用双端队列，根据当前方向决定添加到队列的头部还是尾部
 *    - 使用两个栈交替存储不同层的节点，实现方向切换
 *
 * 时间复杂度分析：
```

```
 * - 时间复杂度：O(N)，其中 N 是二叉树中的节点数，每个节点只被访问一次
 * - 空间复杂度：O(W)，其中 W 是二叉树的最大宽度，最坏情况下为 O(N/2)≈O(N)
 *
 * 关键优化点：
 * 1. 使用双端队列可以避免额外的反转操作，提高性能
 * 2. 使用数组实现队列可以进一步提升内存效率
 * 3. 使用两个栈的方法在某些情况下可能更直观
 *
 * 边界情况处理：
 * 1. 空树：直接返回空列表
 * 2. 单节点树：返回只包含一个列表的列表
 * 3. 斜树（链状结构）：仍然能正确处理锯齿形遍历
 *
 * 相关题目：
 * 1. LeetCode 103. 二叉树的锯齿形层序遍历（本文件） - https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
 * 2. LintCode 71. 二叉树的锯齿形层次遍历 - https://www.lintcode.com/problem/71/
 * 3. HackerRank Tree: Zig Zag Level Order Traversal - https://www.hackerrank.com/challenges/tree-zigzag-level-order-traversal/problem
 * 4. CodeChef Zigzag Tree Traversal - https://www.codechef.com/problems/ZIGZAGT
 * 5. GeeksforGeeks ZigZag Tree Traversal - https://www.geeksforgeeks.org/zigzag-tree-traversal/
 * 6. POJ 3278. Catch That Cow - https://poj.org/problem?id=3278 (类似 BFS 方向问题)
 * 7. Codeforces 1335B. Construct the String - https://codeforces.com/problemset/problem/1335/B
 * 8. AcWing 102. 最佳牛围栏 - https://www.acwing.com/problem/content/104/
 * 9. 牛客 NC15. 求二叉树的层序遍历 - https://www.nowcoder.com/practice/04a5560e43e24e9db4595865dc9c63a3
 * 10. 剑指 Offer 32 - III. 从上到下打印二叉树 III - https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/
 *
 * 工程化考量：
 * 1. 异常处理：需要考虑空指针异常和数据溢出问题
 * 2. 可配置性：MAXN 参数可以根据实际需求调整
 * 3. 线程安全：多线程环境下需要额外的同步机制
 * 4. 内存管理：对于大数据量的树，需要注意内存使用情况
 *
 * 算法变体与扩展：
 * 1. 可以调整起始方向（从左到右或从右到左）
 * 2. 可以每 k 层切换一次方向，而不是每层切换
 * 3. 在图像处理中可应用于边缘检测和特征提取
 * 4. 在网络路由算法中有类似的分层方向搜索策略
 */
public class Code02_ZigzagLevelOrderTraversal {
```

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

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

    @Override
    public String toString() {
        return String.valueOf(val);
    }
}

// 如果测试数据量变大了就修改这个值
// 注意：在工程实践中，可能需要动态扩容或使用更灵活的数据结构
public static int MAXN = 2001;

// 使用数组模拟队列，提高访问效率
public static TreeNode[] queue = new TreeNode[MAXN];

// 队列的左右指针
public static int l, r;

/**
 * 方法 1：使用数组队列实现的锯齿形层序遍历 - 推荐解法
 * 时间复杂度：O(N)，每个节点只被访问一次
 * 空间复杂度：O(W)，W 为树的最大宽度
 * 优点：
 * 1. 不需要额外的集合类，内存效率高
 * 2. 可以直接按照指定顺序收集节点值，避免了额外的反转操作
 * 3. 数组实现的队列比 LinkedList 具有更好的缓存局部性
 * 核心思想：
 * - 使用两个循环：第一个循环按照当前方向收集节点值
 * - 第二个循环将子节点按顺序加入队列，保持正确的层次结构
 * - 通过 reverse 标志位控制每层的遍历方向
 */
```

```
    public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        if (root != null) {
            l = r = 0;
            queue[r++] = root;
            // false 代表从左往右，true 代表从右往左
            boolean reverse = false;
            while (l < r) {
                int size = r - l; // 当前层的节点数量
                ArrayList<Integer> list = new ArrayList<Integer>();

                // 第一遍循环：根据当前遍历方向收集节点值
                // reverse == false: 从左往右，顺序遍历队列中的当前层节点
                // reverse == true: 从右往左，逆序遍历队列中的当前层节点
                for (int i = reverse ? r - 1 : l, j = reverse ? -1 : 1, k = 0; k < size; i += j,
k++) {
                    TreeNode cur = queue[i];
                    list.add(cur.val);
                }

                // 第二遍循环：将当前层所有节点的子节点按顺序加入队列
                for (int i = 0; i < size; i++) {
                    TreeNode cur = queue[l++];
                    if (cur.left != null) {
                        queue[r++] = cur.left;
                    }
                    if (cur.right != null) {
                        queue[r++] = cur.right;
                    }
                }

                ans.add(list);
                reverse = !reverse; // 反转下一层的遍历方向
            }
        }
        return ans;
    }

    /**
     * 方法 2：使用 LinkedList 实现的锯齿形层序遍历
     * 时间复杂度：O(N)，每个节点只被访问一次，但反转操作会增加一些常数时间
     * 空间复杂度：O(N)
     * 优点：
```

```
     * 1. 代码更加简洁易懂，逻辑清晰
     * 2. 使用标准库的 Queue 接口，实现更加规范
     * 缺点：
     * 1. 在偶数层需要额外的反转操作，增加了时间常数
     * 2. LinkedList 的节点开销比数组大
     * 实现思路：
     * - 先进行标准的层序遍历，收集每层的节点值
     * - 根据当前层数决定是否反转结果列表
     * - 适合代码可读性要求高于极致性能的场景
     */
    public static List<List<Integer>> zigzagLevelOrder2(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        if (root == null) return ans;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        boolean reverse = false; // 控制遍历方向

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

            // 处理当前层的所有节点
            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();
                level.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);
            }

            // 如果需要反转，反转当前层的结果
            if (reverse) {
                Collections.reverse(level);
            }

            ans.add(level);
            reverse = !reverse; // 切换方向
        }

        return ans;
    }
```

```
/**
 * 辅助方法：创建测试用的二叉树
 * 根据数组创建二叉树，null 表示空节点
 * 数组格式：层序遍历顺序，如[3,9,20,null,null,15,7]
 */
public static TreeNode createTree(Integer[] nums) {
    if (nums == null || nums.length == 0) return null;

    TreeNode root = new TreeNode(nums[0]);
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    int index = 1;
    while (!queue.isEmpty() && index < nums.length) {
        TreeNode node = queue.poll();

        // 添加左子节点
        if (index < nums.length && nums[index] != null) {
            node.left = new TreeNode(nums[index]);
            queue.offer(node.left);
        }
        index++;

        // 添加右子节点
        if (index < nums.length && nums[index] != null) {
            node.right = new TreeNode(nums[index]);
            queue.offer(node.right);
        }
        index++;
    }

    return root;
}

/**
 * 辅助方法：打印树的结构（层序方式）
 * 用于可视化测试树，便于调试
 */
public static void printTree(TreeNode root) {
    if (root == null) {
        System.out.println("Empty Tree");
        return;
```

```
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    List<List<String>> levels = new ArrayList<>();

    while (!queue.isEmpty()) {
        int size = queue.size();
        List<String> level = new ArrayList<>();
        boolean hasNonNull = false;

        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (node != null) {
                level.add(String.valueOf(node.val));
                queue.offer(node.left);
                queue.offer(node.right);
                if (node.left != null || node.right != null) {
                    hasNonNull = true;
                }
            } else {
                level.add("null");
                queue.offer(null);
                queue.offer(null);
            }
        }

        if (!hasNonNull && level.stream().allMatch("null"::equals)) {
            break;
        }
        levels.add(level);
    }

    // 打印每层
    for (List<String> level : levels) {
        System.out.println(level);
    }
}

/**
 * 主方法用于测试
 * 包含多种测试用例，覆盖常见场景和边界情况
 */
```

```
public static void main(String[] args) {
    // 测试用例 1: [3,9,20,null,null,15,7]
    //       3
    //      / \
    //     9  20
    //       /  \
    //      15   7
    System.out.println("========== 测试用例 1：标准二叉树 ==========");
    TreeNode root1 = createTree(new Integer[]{3, 9, 20, null, null, 15, 7});
    System.out.println("树结构：");
    printTree(root1);

    System.out.println("\n 方法 1 锯齿形遍历 (数组实现):");
    for (List<Integer> level : zigzagLevelOrder(root1)) {
        System.out.println(level);
    }

    System.out.println("\n 方法 2 锯齿形遍历 (LinkedList 实现):");
    for (List<Integer> level : zigzagLevelOrder2(root1)) {
        System.out.println(level);
    }

    // 测试用例 2: [1] - 单节点树
    System.out.println("\n\n========== 测试用例 2：单节点树 ==========");
    TreeNode root2 = createTree(new Integer[]{1});
    System.out.println("树结构：");
    printTree(root2);
    System.out.println("\n 锯齿形遍历:");
    for (List<Integer> level : zigzagLevelOrder(root2)) {
        System.out.println(level);
    }

    // 测试用例 3: [] - 空树
    System.out.println("\n\n========== 测试用例 3：空树 ==========");
    TreeNode root3 = null;
    System.out.println("树结构：");
    printTree(root3);
    System.out.println("\n 锯齿形遍历:");
    System.out.println(zigzagLevelOrder(root3));

    // 测试用例 4: [1,2,3,4,5,6,7] - 完全二叉树
    System.out.println("\n\n========== 测试用例 4：完全二叉树 ==========");
    //       1
```

```
        //      / \
        //     2   3
        //    / \ / \
        //   4  5 6  7
        TreeNode root4 = createTree(new Integer[]{1, 2, 3, 4, 5, 6, 7});
        System.out.println("树结构：");
        printTree(root4);
        System.out.println("\n锯齿形遍历:");
        for (List<Integer> level : zigzagLevelOrder(root4)) {
            System.out.println(level);
        }

        // 测试用例5: [1,2,null,3,null,4,null] - 斜树（链状结构）
        System.out.println("\n\n========== 测试用例5：斜树 ==========");
        TreeNode root5 = createTree(new Integer[]{1, 2, null, 3, null, 4, null});
        System.out.println("树结构：");
        printTree(root5);
        System.out.println("\n锯齿形遍历:");
        for (List<Integer> level : zigzagLevelOrder(root5)) {
            System.out.println(level);
        }

        // 性能对比分析
        System.out.println("\n\n========== 性能对比分析 ==========");
        System.out.println("1. 方法1（数组队列）：内存效率更高，常数时间更小，适合大数据量");
        System.out.println("2. 方法2（LinkedList+反转）：代码更简洁，但有额外的反转开销");
        System.out.println("3. 空间复杂度：两种方法均为O(N)，但方法1的内存使用更高效");
        System.out.println("4. 实际应用中，对于大型树，方法1通常表现更好");
    }
}

/*
Python实现:

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

class ZigzagLevelOrderTraversal:
    # 方法1：使用deque实现的锯齿形层序遍历
    def zigzagLevelOrder1(self, root: TreeNode) -> list[list[int]]:
```

```
    """
    使用双端队列实现锯齿形层序遍历
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    result = []
    if not root:
        return result

    from collections import deque
    queue = deque([root])
    reverse = False  # 控制遍历方向

    while queue:
        size = len(queue)
        level = deque()

        # 处理当前层的所有节点
        for _ in range(size):
            node = queue.popleft()

            # 根据当前方向决定添加到 level 的左侧还是右侧
            if reverse:
                level.appendleft(node.val)
            else:
                level.append(node.val)

            # 将子节点加入队列
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(list(level))
        reverse = not reverse  # 切换方向

    return result

# 方法 2：使用常规 BFS + 反转偶数层
def zigzagLevelOrder2(self, root: TreeNode) -> list[list[int]]:
    """
    使用常规层序遍历，然后反转偶数层
    时间复杂度: O(N)
```

```
    空间复杂度: O(N)
    """
    result = []
    if not root:
        return result

    from collections import deque
    queue = deque([root])
    level_num = 0  # 记录当前层数

    while queue:
        size = len(queue)
        level = []

        for _ in range(size):
            node = queue.popleft()
            level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        # 偶数层（从 0 开始计数）保持原样，奇数层反转
        if level_num % 2 == 1:
            level.reverse()

        result.append(level)
        level_num += 1

    return result

# 方法 3：使用两个栈实现
def zigzagLevelOrder3(self, root: TreeNode) -> list[list[int]]:
    """
    使用两个栈交替存储不同层的节点
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    result = []
    if not root:
        return result
```

```
    # 使用两个栈，分别存储奇数层和偶数层的节点
    stack1 = [root]  # 存储奇数层节点，从左到右
    stack2 = []      # 存储偶数层节点，从右到左

    while stack1 or stack2:
        # 处理奇数层
        if stack1:
            level = []
            while stack1:
                node = stack1.pop()
                level.append(node.val)
                # 先左子节点，后右子节点
                if node.left:
                    stack2.append(node.left)
                if node.right:
                    stack2.append(node.right)
            result.append(level)

        # 处理偶数层
        elif stack2:
            level = []
            while stack2:
                node = stack2.pop()
                level.append(node.val)
                # 先右子节点，后左子节点
                if node.right:
                    stack1.append(node.right)
                if node.left:
                    stack1.append(node.left)
            result.append(level)

    return result

# 辅助方法：创建二叉树
def createTree(self, nums: list) -> TreeNode:
    if not nums:
        return None

    root = TreeNode(nums[0])
    from collections import deque
    queue = deque([root])

    index = 1
```

```
        while queue and index < len(nums):
            node = queue.popleft()

            # 添加左子节点
            if index < len(nums) and nums[index] is not None:
                node.left = TreeNode(nums[index])
                queue.append(node.left)
            index += 1

            # 添加右子节点
            if index < len(nums) and nums[index] is not None:
                node.right = TreeNode(nums[index])
                queue.append(node.right)
            index += 1

        return root

# 测试代码
if __name__ == "__main__":
    solution = ZigzagLevelOrderTraversal()

    # 测试用例 1: [3,9,20,None,None,15,7]
    root1 = solution.createTree([3, 9, 20, None, None, 15, 7])
    print("测试用例 1 (双端队列实现):")
    print(solution.zigzagLevelOrder1(root1))

    # 测试用例 2: [1]
    root2 = solution.createTree([1])
    print("\n测试用例 2:")
    print(solution.zigzagLevelOrder1(root2))

    # 测试用例 3: []
    root3 = solution.createTree([])
    print("\n测试用例 3:")
    print(solution.zigzagLevelOrder1(root3))

    # 测试用例 4: [1,2,3,4,5,6,7]
    root4 = solution.createTree([1, 2, 3, 4, 5, 6, 7])
    print("\n测试用例 4 (三种实现对比):")
    print("方法 1:", solution.zigzagLevelOrder1(root4))
    print("方法 2:", solution.zigzagLevelOrder2(root4))
    print("方法 3:", solution.zigzagLevelOrder3(root4))
```

C++实现:

```
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <algorithm>
#include <deque>
using namespace std;

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 ZigzagLevelOrderTraversal {
public:
    // 方法 1：使用双端队列实现的锯齿形层序遍历
    vector<vector<int>> zigzagLevelOrder1(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;

        queue<TreeNode*> q;
        q.push(root);
        bool reverse = false; // 控制遍历方向

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

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

                // 根据当前方向决定添加到 level 的左侧还是右侧
                if (reverse) {
                    level.push_front(node->val);
                } else {
                    level.push_back(node->val);
```

```
            }

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

        // 将双端队列转换为 vector 并添加到结果中
        result.push_back(vector<int>(level.begin(), level.end()));
        reverse = !reverse; // 切换方向
    }

    return result;
}

// 方法 2：使用常规 BFS + 反转偶数层
vector<vector<int>> zigzagLevelOrder2(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    queue<TreeNode*> q;
    q.push(root);
    int levelNum = 0; // 记录当前层数

    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);
        }

        // 奇数层(从 0 开始计数)需要反转
        if (levelNum % 2 == 1) {
            reverse(level.begin(), level.end());
        }

        result.push_back(level);
        levelNum++;
```

```
    }

    return result;
}

// 方法 3：使用两个栈实现
vector<vector<int>> zigzagLevelOrder3(TreeNode* root) {
    vector<vector<int>> result;
    if (!root) return result;

    stack<TreeNode*> stack1; // 存储奇数层节点
    stack<TreeNode*> stack2; // 存储偶数层节点
    stack1.push(root);

    while (!stack1.empty() || !stack2.empty()) {
        // 处理奇数层
        if (!stack1.empty()) {
            vector<int> level;
            while (!stack1.empty()) {
                TreeNode* node = stack1.top();
                stack1.pop();
                level.push_back(node->val);

                // 先左后右，这样出栈顺序是右左
                if (node->left) stack2.push(node->left);
                if (node->right) stack2.push(node->right);
            }
            result.push_back(level);
        }
        // 处理偶数层
        else if (!stack2.empty()) {
            vector<int> level;
            while (!stack2.empty()) {
                TreeNode* node = stack2.top();
                stack2.pop();
                level.push_back(node->val);

                // 先右后左，这样出栈顺序是左右
                if (node->right) stack1.push(node->right);
                if (node->left) stack1.push(node->left);
            }
            result.push_back(level);
        }
```

```
    }

    return result;
}

// 辅助方法：创建二叉树
TreeNode* createTree(const vector<int*>& nums) {
    if (nums.empty() || !nums[0]) return nullptr;

    TreeNode* root = new TreeNode(*nums[0]);
    queue<TreeNode*> q;
    q.push(root);

    int index = 1;
    while (!q.empty() && index < nums.size()) {
        TreeNode* node = q.front();
        q.pop();

        // 添加左子节点
        if (index < nums.size() && nums[index]) {
            node->left = new TreeNode(*nums[index]);
            q.push(node->left);
        }
        index++;

        // 添加右子节点
        if (index < nums.size() && nums[index]) {
            node->right = new TreeNode(*nums[index]);
            q.push(node->right);
        }
        index++;
    }

    return root;
}

// 辅助方法：释放树内存
void deleteTree(TreeNode* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
```

```
    }

    // 辅助方法：打印结果
    void printResult(const vector<vector<int>>& result) {
        cout << "[" << endl;
        for (const auto& level : result) {
            cout << "  [";
            for (size_t i = 0; i < level.size(); ++i) {
                cout << level[i];
                if (i < level.size() - 1) cout << ", ";
            }
            cout << "]" << endl;
        }
        cout << "]" << endl;
    }
};

// 测试代码
int main() {
    ZigzagLevelOrderTraversal solution;

    // 测试用例 1: [3,9,20,null,null,15,7]
    vector<int*> nums1 = {new int(3), new int(9), new int(20), nullptr, nullptr, new int(15), new
int(7)};
    TreeNode* root1 = solution.createTree(nums1);
    cout << "测试用例 1 (双端队列实现):" << endl;
    solution.printResult(solution.zigzagLevelOrder1(root1));

    // 测试用例 2: [1]
    vector<int*> nums2 = {new int(1)};
    TreeNode* root2 = solution.createTree(nums2);
    cout << "\n 测试用例 2:" << endl;
    solution.printResult(solution.zigzagLevelOrder1(root2));

    // 测试用例 3: []
    vector<int*> nums3 = {};
    TreeNode* root3 = solution.createTree(nums3);
    cout << "\n 测试用例 3:" << endl;
    solution.printResult(solution.zigzagLevelOrder1(root3));

    // 测试用例 4: [1,2,3,4,5,6,7]
    vector<int*> nums4 = {new int(1), new int(2), new int(3), new int(4), new int(5), new int(6),
new int(7)};
```

```
    TreeNode* root4 = solution.createTree(nums4);
    cout << "\n 测试用例 4 (三种实现对比):" << endl;
    cout << "方法 1:" << endl;
    solution.printResult(solution.zigzagLevelOrder1(root4));
    cout << "\n 方法 2:" << endl;
    solution.printResult(solution.zigzagLevelOrder2(root4));
    cout << "\n 方法 3:" << endl;
    solution.printResult(solution.zigzagLevelOrder3(root4));

    // 释放内存
    solution.deleteTree(root1);
    solution.deleteTree(root2);
    solution.deleteTree(root3);
    solution.deleteTree(root4);

    // 释放 nums 中的 int 指针
    for (auto p : nums1) if (p) delete p;
    for (auto p : nums2) if (p) delete p;
    for (auto p : nums4) if (p) delete p;

    return 0;
}
*/

==========================================

文件: Code03_WidthOfBinaryTree1.java
==========================================
package class036;

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

/**
 * 二叉树最大宽度计算
 *
 * 核心算法思想：
 * - 使用层序遍历（BFS）为每个节点分配一个虚拟索引
 * - 对于完全二叉树，假设根节点索引为 1，则左子节点索引为 2*i，右子节点索引为 2*i+1
 * - 每层的宽度等于最后一个节点索引减去第一个节点索引再加 1
 * - 记录所有层中的最大宽度
 *
 * 关键优化点：
```

* - 索引溢出问题：对于深层树，直接使用 2*i 的索引方式会导致整数溢出
* - 优化方案：每层重新编号，减去当前层第一个节点的索引，确保索引不会溢出
* - 实现效率：使用数组实现队列比使用标准库队列性能更好
*
* 边界情况处理：
* - 空树：返回 0
* - 单节点树：返回 1
* - 只有一侧子树的树：需要正确处理中间的 null 节点
*
* 时间复杂度分析：
* - O(N)，其中 N 是二叉树中的节点总数，每个节点只访问一次
*
* 空间复杂度分析：
* - O(N)，队列最多存储树中最宽的一层的所有节点
*
* 工程化考量：
* - 使用 long 类型存储索引，避免整数溢出
* - 在 C++中需要手动管理内存，避免内存泄漏
* - 在 Java 中使用数组实现队列可以提升性能
* - 在 Python 中可以使用动态扩展的列表
*
* 相关题目（穷尽各大平台）：
* 1. LeetCode 662. Maximum Width of Binary Tree - 原题
*   链接：https://leetcode.cn/problems/maximum-width-of-binary-tree/
* 2. LeetCode 116. Populating Next Right Pointers in Each Node - 层序遍历应用
*   链接：https://leetcode.cn/problems/populating-next-right-pointers-in-each-node/
* 3. LeetCode 117. Populating Next Right Pointers in Each Node II - 通用二叉树层序遍历
*   链接：https://leetcode.cn/problems/populating-next-right-pointers-in-each-node-ii/
* 4. LeetCode 102. Binary Tree Level Order Traversal - 基础层序遍历
*   链接：https://leetcode.cn/problems/binary-tree-level-order-traversal/
* 5. LeetCode 515. Find Largest Value in Each Tree Row - 层序遍历求最大值
*   链接：https://leetcode.cn/problems/find-largest-value-in-each-tree-row/
* 6. LintCode 97. Maximum Depth of Binary Tree - 层序遍历求深度
*   链接：https://www.lintcode.com/problem/97/
* 7. HackerRank Tree Level Order Traversal - 标准层序遍历
*   链接：https://www.hackerrank.com/challenges/tree-level-order-traversal
* 8. CodeChef TREE2 - 树的宽度相关问题
*   链接：https://www.codechef.com/problems/TREE2
* 9. POJ 1661 Help Jimmy - 树结构处理
*   链接：http://poj.org/problem?id=1661
* 10. HDU 1026 Ignatius and the Princess I - BFS 应用
*   链接：http://acm.hdu.edu.cn/showproblem.php?pid=1026
* 11. AcWing 847. 图中点的层次 - BFS 层序遍历

```
 *    链接：https://www.acwing.com/problem/content/849/
 * 12. 剑指 Offer 32 - I. 从上到下打印二叉树 - 层序遍历基础
 *    链接：https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/
 * 13. 牛客 NC15 求二叉树的层序遍历 - 层序遍历应用
 *    链接：https://www.nowcoder.com/practice/04a5560e43e24e9db4595865dc9c63a3
 */
public class Code03_WidthOfBinaryTree1 {

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

        public TreeNode() {}

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

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

        @Override
        public String toString() {
            return "TreeNode{val=" + val + "}";
        }
    }

    /**
     * 方法 1：使用数组实现的队列进行层序遍历
     * 时间复杂度：O(N)
     * 空间复杂度：O(N)
     * 优点：数组实现的队列比 LinkedList 效率更高，常数更小
     * 缺点：需要预先定义最大容量
     */
    // 如果测试数据量变大了就修改这个值
    public static int MAXN = 3001;

    // 节点队列
```

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

// 索引队列，存储每个节点的虚拟索引
public static long[] iq = new long[MAXN];

// 队列的左右指针
public static int l, r;

/**
 * 计算二叉树的最大宽度 - 数组队列实现
 *
 * 实现思路：
 * - 使用两个数组分别存储节点和对应的索引
 * - 对于每个节点，其左子节点索引为 2*parentIndex，右子节点索引为 2*parentIndex+1
 * - 每层处理时，计算当前层的宽度（最后一个节点索引 - 第一个节点索引 + 1）
 *
 * 优点：
 * - 性能优于使用标准库队列，避免了对象创建的开销
 * - 数组访问速度快
 *
 * 缺点：
 * - 需要预先定义最大容量
 * - 对于深层树可能存在索引溢出问题
 */
public static int widthOfBinaryTree(TreeNode root) {
    // 边界检查
    if (root == null) {
        return 0;
    }

    int ans = 1;  // 最小宽度为 1（只有根节点）
    l = r = 0;    // 初始化队列指针

    // 根节点入队，索引为 1
    nq[r] = root;
    iq[r++] = 1;

    // 层序遍历
    while (l < r) {
        int size = r - l;  // 当前层的节点数

        // 计算当前层的宽度（最后一个节点索引 - 第一个节点索引 + 1）
        ans = Math.max(ans, (int) (iq[r - 1] - iq[l] + 1));
```

```
            // 处理当前层的所有节点
            for (int i = 0; i < size; i++) {
                TreeNode node = nq[l];
                long id = iq[l++];  // 当前节点的索引

                // 左子节点索引为 2 * id
                if (node.left != null) {
                    nq[r] = node.left;
                    iq[r++] = id * 2;
                }

                // 右子节点索引为 2 * id + 1
                if (node.right != null) {
                    nq[r] = node.right;
                    iq[r++] = id * 2 + 1;
                }
            }
        }

        return ans;
    }

    /**
     * 计算二叉树的最大宽度 - 标准库队列实现
     *
     * 实现思路：
     * - 使用 Java 标准库的 Queue 和自定义 Pair 类存储节点和索引
     * - Pair 类封装了节点和其对应的索引
     * - 层序遍历过程中，记录每层的第一个和最后一个节点的索引
     *
     * 优点：
     * - 实现简单，代码可读性好
     * - 队列自动调整大小，无需预先定义容量
     *
     * 缺点：
     * - 每个节点需要创建 Pair 对象，有一定的内存开销
     * - 性能略低于数组实现
     */
    public static int widthOfBinaryTreeWithQueue(TreeNode root) {
        if (root == null) {
            return 0;
        }
```

```
    int maxWidth = 0;
    // 使用队列存储节点和其索引
    Queue<Pair<TreeNode, Long>> queue = new LinkedList<>();
    queue.offer(new Pair<>(root, 1L));  // 根节点索引为1

    while (!queue.isEmpty()) {
        int size = queue.size();
        long firstIndex = -1;
        long lastIndex = -1;

        for (int i = 0; i < size; i++) {
            Pair<TreeNode, Long> pair = queue.poll();
            TreeNode node = pair.first;
            long index = pair.second;

            // 记录当前层的第一个和最后一个节点索引
            if (i == 0) {
                firstIndex = index;
            }
            if (i == size - 1) {
                lastIndex = index;
            }

            // 子节点入队
            if (node.left != null) {
                queue.offer(new Pair<>(node.left, 2 * index));
            }
            if (node.right != null) {
                queue.offer(new Pair<>(node.right, 2 * index + 1));
            }
        }

        // 更新最大宽度
        maxWidth = Math.max(maxWidth, (int) (lastIndex - firstIndex + 1));
    }

    return maxWidth;
}

/**
 * 节点和索引的配对类
 * 用于在层序遍历过程中同时跟踪节点和其对应的虚拟索引
```

```
 */
private static class Pair<K, V> {
    K first;
    V second;

    public Pair(K first, V second) {
        this.first = first;
        this.second = second;
    }
}

/**
 * 生成测试用例
 *
 * @param type 测试树类型：
 *             0: LeetCode 示例
 *             1: 完全二叉树
 *             2: 只有左子树
 *             3: 只有右子树
 *             4: 空节点较多的树
 * @return 生成的测试树
 */
public static TreeNode generateTestTree(int type) {
    switch (type) {
        case 1: // 完全二叉树
            TreeNode root1 = new TreeNode(1);
            root1.left = new TreeNode(3);
            root1.right = new TreeNode(2);
            root1.left.left = new TreeNode(5);
            root1.left.right = new TreeNode(3);
            root1.right.right = new TreeNode(9);
            return root1;
        case 2: // 只有左子树
            TreeNode root2 = new TreeNode(1);
            root2.left = new TreeNode(3);
            root2.left.left = new TreeNode(5);
            root2.left.left.left = new TreeNode(7);
            return root2;
        case 3: // 只有右子树
            TreeNode root3 = new TreeNode(1);
            root3.right = new TreeNode(3);
            root3.right.right = new TreeNode(5);
            root3.right.right.right = new TreeNode(7);
```

```
            return root3;
        case 4: // 空节点较多的树
            TreeNode root4 = new TreeNode(1);
            root4.left = new TreeNode(3);
            root4.right = new TreeNode(2);
            root4.left.left = new TreeNode(5);
            root4.left.right = null;
            root4.right.left = null;
            root4.right.right = new TreeNode(9);
            return root4;
        default: // LeetCode 示例
            TreeNode root = new TreeNode(1);
            root.left = new TreeNode(3);
            root.right = new TreeNode(2);
            root.left.left = new TreeNode(5);
            root.left.right = new TreeNode(3);
            root.right.right = new TreeNode(9);
            return root;
    }
}

/**
 * 打印树的结构（简化版）
 *
 * 打印格式：[1, 3, 2, 5, 3, null, 9]
 * 按照层序遍历的顺序，null 表示不存在的节点
 * 移除末尾多余的 null，使输出更简洁
 */
public static void printTree(TreeNode root) {
    if (root == null) {
        System.out.println("[null]");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    StringBuilder sb = new StringBuilder();
    sb.append("[");

    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
```

```
            if (node != null) {
                sb.append(node.val);
                queue.offer(node.left);
                queue.offer(node.right);
            } else {
                sb.append("null");
            }
            if (!queue.isEmpty() || i < size - 1) {
                sb.append(", ");
            }
        }
    }

    // 移除末尾多余的 null
    int lastValidIndex = sb.length() - 1;
    while (lastValidIndex >= 0 &&
           (sb.charAt(lastValidIndex) == 'n' ||
            sb.charAt(lastValidIndex) == 'u' ||
            sb.charAt(lastValidIndex) == 'l' ||
            sb.charAt(lastValidIndex) == ',' ||
            sb.charAt(lastValidIndex) == ' ')) {
        lastValidIndex--;
    }

    sb.replace(lastValidIndex + 1, sb.length(), "");
    sb.append("]");
    System.out.println(sb.toString());
}

public static void main(String[] args) {
    // 测试不同类型的树
    for (int i = 0; i <= 4; i++) {
        System.out.println("\n 测试树类型 " + i + ":");
        TreeNode root = generateTestTree(i);
        System.out.print("树结构: ");
        printTree(root);

        // 使用数组队列实现
        int width1 = widthOfBinaryTree(root);
        System.out.println("数组队列实现最大宽度: " + width1);

        // 使用标准库队列实现
        int width2 = widthOfBinaryTreeWithQueue(root);
```

```
            System.out.println("标准库队列实现最大宽度: " + width2);
        }

        // 边界情况测试
        System.out.println("\n边界情况测试:");

        // 空树
        System.out.println("空树最大宽度: " + widthOfBinaryTree(null));

        // 单节点
        TreeNode singleNode = new TreeNode(1);
        System.out.println("单节点树最大宽度: " + widthOfBinaryTree(singleNode));
    }
}

/*
Python 实现:

# 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

class WidthOfBinaryTree1:
    """
    二叉树最大宽度 - 解法 1
    使用数组实现队列进行层序遍历，效率更高
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    def widthOfBinaryTree(self, root: TreeNode) -> int:
        """
        计算二叉树的最大宽度
        使用数组实现的队列
        """
        if not root:
            return 0

        # 预先定义数组大小（Python 中可以动态扩展）
        MAXN = 3001
        nq = [None] * MAXN  # 节点队列
```

```
iq = [0] * MAXN     # 索引队列
l, r = 0, 0

nq[r] = root
iq[r] = 1
r += 1

max_width = 1

while l < r:
    size = r - l
    # 计算当前层的宽度
    current_width = iq[r - 1] - iq[l] + 1
    if current_width > max_width:
        max_width = current_width

    # 处理当前层的所有节点
    for _ in range(size):
        node = nq[l]
        index = iq[l]
        l += 1

        # 左子节点入队
        if node.left:
            nq[r] = node.left
            iq[r] = index * 2
            r += 1

        # 右子节点入队
        if node.right:
            nq[r] = node.right
            iq[r] = index * 2 + 1
            r += 1

    # 如果队列已满，扩展队列（Python 特有处理）
    if r >= MAXN - 10:
        MAXN *= 2
        new_nq = [None] * MAXN
        new_iq = [0] * MAXN
        new_nq[:r] = nq[:r]
        new_iq[:r] = iq[:r]
        nq, iq = new_nq, new_iq
```

```
    return max_width

def widthOfBinaryTreeWithQueue(self, root: TreeNode) -> int:
    """
    计算二叉树的最大宽度
    使用 Python 标准库的 deque 实现
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    if not root:
        return 0

    from collections import deque
    max_width = 0
    # 使用双端队列存储(节点, 索引)元组
    queue = deque()
    queue.append((root, 1))  # 根节点索引为 1

    while queue:
        size = len(queue)
        first_index = None
        last_index = None

        for i in range(size):
            node, index = queue.popleft()

            # 记录当前层的第一个和最后一个节点索引
            if i == 0:
                first_index = index
            if i == size - 1:
                last_index = index

            # 子节点入队
            if node.left:
                queue.append((node.left, 2 * index))
            if node.right:
                queue.append((node.right, 2 * index + 1))

        # 更新最大宽度
        current_width = last_index - first_index + 1
        if current_width > max_width:
            max_width = current_width
```

```
        return max_width

    def widthOfBinaryTreeOptimized(self, root: TreeNode) -> int:
        """
        优化版本：避免索引溢出
        通过每层重新编号的方式，减去当前层第一个节点的索引
        防止大深度树的索引溢出问题
        """
        if not root:
            return 0

        from collections import deque
        max_width = 0
        queue = deque([(root, 0)])  # 根节点索引为0

        while queue:
            size = len(queue)
            level_start = queue[0][1]  # 当前层第一个节点的索引
            first = last = 0

            for i in range(size):
                node, index = queue.popleft()
                # 重新编号，避免溢出
                index -= level_start

                if i == 0:
                    first = index
                if i == size - 1:
                    last = index

                # 子节点入队，使用新的索引计算方式
                if node.left:
                    queue.append((node.left, index * 2))
                if node.right:
                    queue.append((node.right, index * 2 + 1))

            max_width = max(max_width, last - first + 1)

        return max_width

    def generateTestTree(self, tree_type: int) -> TreeNode:
        """
        生成不同类型的测试树
```

```
    """
    if tree_type == 1:  # 完全二叉树
        root = TreeNode(1)
        root.left = TreeNode(3)
        root.right = TreeNode(2)
        root.left.left = TreeNode(5)
        root.left.right = TreeNode(3)
        root.right.right = TreeNode(9)
        return root
    elif tree_type == 2:  # 只有左子树
        root = TreeNode(1)
        root.left = TreeNode(3)
        root.left.left = TreeNode(5)
        root.left.left.left = TreeNode(7)
        return root
    elif tree_type == 3:  # 只有右子树
        root = TreeNode(1)
        root.right = TreeNode(3)
        root.right.right = TreeNode(5)
        root.right.right.right = TreeNode(7)
        return root
    elif tree_type == 4:  # 空节点较多的树
        root = TreeNode(1)
        root.left = TreeNode(3)
        root.right = TreeNode(2)
        root.left.left = TreeNode(5)
        root.left.right = None
        root.right.left = None
        root.right.right = TreeNode(9)
        return root
    else:  # LeetCode 示例
        root = TreeNode(1)
        root.left = TreeNode(3)
        root.right = TreeNode(2)
        root.left.left = TreeNode(5)
        root.left.right = TreeNode(3)
        root.right.right = TreeNode(9)
        return root

def printTree(self, root: TreeNode) -> None:
    """
    打印树的结构（简化版）
    """
```

```
        if not root:
            print("[null]")
            return

        from collections import deque
        queue = deque([root])
        result = []

        while queue:
            node = queue.popleft()
            if node:
                result.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else:
                result.append("null")

        # 移除末尾多余的 null
        while result and result[-1] == "null":
            result.pop()

        print("[" + ", ".join(result) + "]")

# 测试代码
if __name__ == "__main__":
    solution = WidthOfBinaryTree1()

    # 测试不同类型的树
    for i in range(5):
        print(f"\n 测试树类型 {i}:")
        root = solution.generateTestTree(i)
        print("树结构:", end=" ")
        solution.printTree(root)

        # 使用数组队列实现
        width1 = solution.widthOfBinaryTree(root)
        print(f"数组队列实现最大宽度: {width1}")

        # 使用标准库队列实现
        width2 = solution.widthOfBinaryTreeWithQueue(root)
        print(f"标准库队列实现最大宽度: {width2}")

        # 使用优化版本
```

```
        width3 = solution.widthOfBinaryTreeOptimized(root)
        print(f"优化版本最大宽度: {width3}")

    # 边界情况测试
    print("\n边界情况测试:")
    print(f"空树最大宽度: {solution.widthOfBinaryTree(None)}")

    single_node = TreeNode(1)
    print(f"单节点树最大宽度: {solution.widthOfBinaryTree(single_node)}")
```

C++实现:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <string>
#include <sstream>
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 WidthOfBinaryTree1 {
public:
    /**
     * 方法1：使用数组实现的队列进行层序遍历
     * 时间复杂度: O(N)
     * 空间复杂度: O(N)
     */
    // 可以根据需要调整最大容量
    static const int MAXN = 3001;

    /**
     * 计算二叉树的最大宽度
     * @param root 二叉树根节点
```

```
 * @return 最大宽度
 */
int widthOfBinaryTree(TreeNode* root) {
    if (!root) {
        return 0;
    }

    // 定义数组队列
    TreeNode* nq[MAXN] = {nullptr};
    long long iq[MAXN] = {0};
    int l = 0, r = 0;

    // 根节点入队
    nq[r] = root;
    iq[r++] = 1;

    int maxWidth = 1;

    while (l < r) {
        int size = r - l;
        // 计算当前层的宽度
        long long currentWidth = iq[r - 1] - iq[l] + 1;
        if (currentWidth > maxWidth) {
            maxWidth = static_cast<int>(currentWidth);
        }

        // 处理当前层的所有节点
        for (int i = 0; i < size; ++i) {
            TreeNode* node = nq[l];
            long long index = iq[l];
            l++;

            // 左子节点入队
            if (node->left) {
                nq[r] = node->left;
                iq[r] = index * 2;
                r++;
            }

            // 右子节点入队
            if (node->right) {
                nq[r] = node->right;
                iq[r] = index * 2 + 1;
```

```
                r++;
            }
        }
    }

    return maxWidth;
}

/**
 * 方法 2：使用 STL 的 queue 实现
 * 时间复杂度: O(N)
 * 空间复杂度: O(N)
 */
int widthOfBinaryTreeWithQueue(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int maxWidth = 0;
    // 使用队列存储节点和索引的 pair
    queue<pair<TreeNode*, long long>> q;
    q.push({root, 1LL});  // 根节点索引为 1

    while (!q.empty()) {
        int size = q.size();
        long long firstIndex = -1;
        long long lastIndex = -1;

        for (int i = 0; i < size; ++i) {
            auto [node, index] = q.front();
            q.pop();

            // 记录当前层的第一个和最后一个节点索引
            if (i == 0) {
                firstIndex = index;
            }
            if (i == size - 1) {
                lastIndex = index;
            }

            // 子节点入队
            if (node->left) {
                q.push({node->left, 2 * index});
```

```
            }
            if (node->right) {
                q.push({node->right, 2 * index + 1});
            }
        }

        // 更新最大宽度
        long long currentWidth = lastIndex - firstIndex + 1;
        if (currentWidth > maxWidth) {
            maxWidth = static_cast<int>(currentWidth);
        }
    }

    return maxWidth;
}

/**
 * 方法 3：优化版本，避免索引溢出
 * 通过每层重新编号的方式
 */
int widthOfBinaryTreeOptimized(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int maxWidth = 0;
    queue<pair<TreeNode*, long long>> q;
    q.push({root, 0LL});  // 根节点索引为 0

    while (!q.empty()) {
        int size = q.size();
        long long levelStart = q.front().second;  // 当前层第一个节点的索引
        long long first = 0, last = 0;

        for (int i = 0; i < size; ++i) {
            auto [node, index] = q.front();
            q.pop();

            // 重新编号，避免溢出
            index -= levelStart;

            if (i == 0) {
                first = index;
```

```
            }
            if (i == size - 1) {
                last = index;
            }

            // 子节点入队，使用新的索引计算方式
            if (node->left) {
                q.push({node->left, index * 2});
            }
            if (node->right) {
                q.push({node->right, index * 2 + 1});
            }
        }

        maxWidth = max(maxWidth, static_cast<int>(last - first + 1));
    }

    return maxWidth;
}

/**
 * 生成测试用例
 */
TreeNode* generateTestTree(int treeType) {
    switch (treeType) {
        case 1: {
            // 完全二叉树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->right = new TreeNode(2);
            root->left->left = new TreeNode(5);
            root->left->right = new TreeNode(3);
            root->right->right = new TreeNode(9);
            return root;
        }
        case 2: {
            // 只有左子树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->left->left = new TreeNode(5);
            root->left->left->left = new TreeNode(7);
            return root;
        }
```

```
        case 3: {
            // 只有右子树
            TreeNode* root = new TreeNode(1);
            root->right = new TreeNode(3);
            root->right->right = new TreeNode(5);
            root->right->right->right = new TreeNode(7);
            return root;
        }
        case 4: {
            // 空节点较多的树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->right = new TreeNode(2);
            root->left->left = new TreeNode(5);
            root->left->right = nullptr;
            root->right->left = nullptr;
            root->right->right = new TreeNode(9);
            return root;
        }
        default: {
            // LeetCode 示例
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->right = new TreeNode(2);
            root->left->left = new TreeNode(5);
            root->left->right = new TreeNode(3);
            root->right->right = new TreeNode(9);
            return root;
        }
    }
}

/**
 * 释放树内存
 */
void deleteTree(TreeNode* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
}
```

```
    /**
     * 打印树的结构
     */
    void printTree(TreeNode* root) {
        if (!root) {
            cout << "[null]" << endl;
            return;
        }

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

        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();

            if (node) {
                result.push_back(to_string(node->val));
                q.push(node->left);
                q.push(node->right);
            } else {
                result.push_back("null");
            }
        }

        // 移除末尾多余的null
        while (!result.empty() && result.back() == "null") {
            result.pop_back();
        }

        cout << "[";
        for (size_t i = 0; i < result.size(); ++i) {
            cout << result[i];
            if (i < result.size() - 1) {
                cout << ", ";
            }
        }
        cout << "]" << endl;
    }
};

// 测试代码
```

```
int main() {
    WidthOfBinaryTree1 solution;

    // 测试不同类型的树
    for (int i = 0; i < 5; ++i) {
        cout << "\n测试树类型 " << i << ":" << endl;
        TreeNode* root = solution.generateTestTree(i);
        cout << "树结构: ";
        solution.printTree(root);

        // 使用数组队列实现
        int width1 = solution.widthOfBinaryTree(root);
        cout << "数组队列实现最大宽度: " << width1 << endl;

        // 使用STL队列实现
        int width2 = solution.widthOfBinaryTreeWithQueue(root);
        cout << "STL队列实现最大宽度: " << width2 << endl;

        // 使用优化版本
        int width3 = solution.widthOfBinaryTreeOptimized(root);
        cout << "优化版本最大宽度: " << width3 << endl;

        // 释放内存
        solution.deleteTree(root);
    }

    // 边界情况测试
    cout << "\n边界情况测试:" << endl;
    cout << "空树最大宽度: " << solution.widthOfBinaryTree(nullptr) << endl;

    TreeNode* singleNode = new TreeNode(1);
    cout << "单节点树最大宽度: " << solution.widthOfBinaryTree(singleNode) << endl;
    solution.deleteTree(singleNode);

    return 0;
}
*/
```

==========

文件: Code03_WidthOfBinaryTree2.java

==========

```
package class036;
```

```
import java.util.Deque;
import java.util.LinkedList;

/**
 * 二叉树的最大宽度 - 解法2（优化版）
 * 测试链接 : https://leetcode.cn/problems/maximum-width-of-binary-tree/
 *
 * 本实现专注于解决索引溢出问题并优化性能。
 * 当二叉树深度很大时，简单的索引计算方式（2 * id）可能导致数值溢出，
 * 因此这个版本采用了索引重编号的技巧，确保在处理深层树时不会溢出。
 *
 * 核心算法思想：
 * 1. 每层开始时，以当前层第一个节点的索引为基准
 * 2. 对当前层的所有节点索引进行重编号（减去基准值）
 * 3. 子节点的索引计算基于原始父节点索引
 * 4. 这样可以避免索引值快速增长导致的溢出问题
 *
 * 关键优化点：
 * - 使用long类型存储索引，增加容量上限
 * - 每层重编号，确保索引不会指数级增长
 * - 提供多种实现方式（BFS、DFS、HashMap优化），适应不同场景
 *
 * 边界情况处理：
 * - 空树返回宽度0
 * - 单节点树返回宽度1
 * - 深层树（接近int最大值）不会溢出
 * - 树中包含大量null节点的情况
 *
 * 时间复杂度：O(N)，其中N是二叉树中的节点数
 * 空间复杂度：O(N)，需要队列存储节点和索引信息
 *
 * 工程化考量：
 * - 使用泛型Pair类封装节点和索引
 * - 提供完整的测试用例覆盖各种场景
 * - 实现树的可视化打印辅助调试
 * - 支持性能测试，评估不同算法在深层树场景下的表现
 *
 * 相关题目：
 * 1. LeetCode 662. 二叉树最大宽度 (本文件)
 * 2. LeetCode 1026. 节点与其祖先之间的最大差值
 * 3. LeetCode 863. 二叉树中所有距离为 K 的结点
 * 4. HackerRank Binary Search Tree : Lowest Common Ancestor
```

```
 * 5. LintCode 102. Binary Tree Level Order Traversal II
 * 6. GeeksforGeeks Boundary Traversal of binary tree
 * 7. LeetCode 1161. 最大层内元素和
 * 8. LeetCode 958. 二叉树的完全性检验
 * 9. LeetCode 111. 二叉树的最小深度
 * 10. LeetCode 110. 平衡二叉树
 * 11. LeetCode 104. 二叉树的最大深度
 * 12. LeetCode 101. 对称二叉树
 * 13. LeetCode 102. 二叉树的层序遍历
 */
public class Code03_WidthOfBinaryTree2 {

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

        public TreeNode() {}

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

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

        @Override
        public String toString() {
            return "TreeNode{" + "val=" + val + "}";
        }
    }

    /**
     * 方法 1：避免索引溢出的优化实现
     * 使用双端队列存储节点和索引，通过每层重编号避免溢出
     * 实现思路：
     * - 使用 BFS 按层遍历树
     * - 对每层节点进行索引重编号，减去当前层起始索引
     * - 计算每层第一个和最后一个节点之间的宽度
```

```
     * 优点：可以处理任意深度的树而不会溢出
     * 时间复杂度：O(N)
     * 空间复杂度：O(N)
     *
     * @param root 二叉树根节点
     * @return 最大宽度
     */
    public static int widthOfBinaryTree(TreeNode root) {
        // 边界检查
        if (root == null) {
            return 0;
        }

        // 初始化最大宽度为1（至少有根节点）
        int maxWidth = 1;

        // 使用双端队列存储节点和对应的索引
        // Pair: 第一个元素是节点，第二个元素是该节点的索引
        Deque<Pair<TreeNode, Long>> queue = new LinkedList<>();
        queue.offerLast(new Pair<>(root, 0L)); // 根节点索引设为0，便于重编号

        // 层序遍历
        while (!queue.isEmpty()) {
            int levelSize = queue.size();

            // 获取当前层第一个节点的索引作为基准
            // 用于后续重编号，避免索引值过大导致溢出
            long levelStartIndex = queue.peekFirst().second;

            long firstIndex = 0, lastIndex = 0;

            // 处理当前层的所有节点
            for (int i = 0; i < levelSize; i++) {
                Pair<TreeNode, Long> pair = queue.pollFirst();
                TreeNode node = pair.first;
                long originalIndex = pair.second;

                // 关键优化：重编号索引，减去当前层起始索引
                // 这样索引值总是从0开始，避免数值溢出
                long currentIndex = originalIndex - levelStartIndex;

                // 记录当前层的第一个和最后一个节点的重编号索引
                if (i == 0) {
```

```
                firstIndex = currentIndex;
            }
            if (i == levelSize - 1) {
                lastIndex = currentIndex;
            }

            // 左子节点入队，使用重编号后的索引计算子节点索引
            if (node.left != null) {
                // 注意：这里使用 originalIndex 而不是 currentIndex 计算子节点索引
                // 因为下一层的起始索引会在下一轮循环中重新计算
                queue.offerLast(new Pair<>(node.left, 2 * originalIndex));
            }

            // 右子节点入队
            if (node.right != null) {
                queue.offerLast(new Pair<>(node.right, 2 * originalIndex + 1));
            }
        }

        // 计算当前层的宽度并更新最大宽度
        int currentWidth = (int) (lastIndex - firstIndex + 1);
        maxWidth = Math.max(maxWidth, currentWidth);
    }

    return maxWidth;
}

/**
 * 方法 2：使用递归 DFS 实现计算二叉树宽度
 * 实现思路：
 * - 深度优先遍历树，记录每个节点的层级和索引
 * - 为每层维护最左侧节点的索引
 * - 计算当前节点到最左侧节点的宽度
 * 优点：代码简洁，空间利用更高效对于平衡树
 * 时间复杂度：O(N)
 * 空间复杂度：O(H)，其中 H 是树的高度，最坏情况下为 O(N)
 */
public static int widthOfBinaryTreeDFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 存储每层最左侧节点的索引
```

```
    // 索引 0 表示第 1 层，以此类推
    LinkedList<Long> levelLeftmostIndices = new LinkedList<>();
    int[] maxWidth = {0}; // 使用数组作为引用类型传递

    // 开始 DFS 递归
    dfs(root, 0, 0, levelLeftmostIndices, maxWidth);

    return maxWidth[0];
}

/**
 * DFS 递归辅助函数
 * 功能：记录每层最左侧节点索引，并计算每层宽度
 * 关键处理：通过相对索引计算当前节点与最左侧节点的距离
 *
 * @param node 当前节点
 * @param level 当前节点所在的层级（从 0 开始）
 * @param index 当前节点的索引
 * @param levelLeftmostIndices 存储每层最左侧节点索引的列表
 * @param maxWidth 最大宽度的引用
 */
private static void dfs(TreeNode node, int level, long index,
                        LinkedList<Long> levelLeftmostIndices,
                        int[] maxWidth) {
    if (node == null) {
        return;
    }

    // 如果是该层的第一个访问的节点，记录其索引
    if (level == levelLeftmostIndices.size()) {
        levelLeftmostIndices.add(index);
    }

    // 获取当前层最左侧节点的索引
    long levelStart = levelLeftmostIndices.get(level);

    // 计算当前节点相对于最左侧节点的位置宽度
    // 重编号索引，避免溢出
    long relativeIndex = index - levelStart;

    // 更新最大宽度
    maxWidth[0] = Math.max(maxWidth[0], (int) (relativeIndex + 1));
```

```
    // 递归处理左子树和右子树
    // 注意：这里继续使用原始索引计算子节点索引
    // 但在计算宽度时使用的是相对索引
    dfs(node.left, level + 1, 2 * index, levelLeftmostIndices, maxWidth);
    dfs(node.right, level + 1, 2 * index + 1, levelLeftmostIndices, maxWidth);
}

/**
 * 方法 3：使用 HashMap 优化的层序遍历实现
 * 实现思路：
 * - 使用 BFS 遍历树，记录每个节点的层级和索引
 * - 使用额外的数据结构存储每层的最小和最大索引
 * - 实时计算每层宽度并更新全局最大值
 * 优点：更灵活，可用于更复杂的树分析场景
 * 空间复杂度：O(L)，其中 L 是树的层数，最坏情况下为 O(N)
 */
public static int widthOfBinaryTreeHashMap(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 存储每层的最小索引
    LinkedList<Long> levelMinIndices = new LinkedList<>();
    // 存储每层的最大索引
    LinkedList<Long> levelMaxIndices = new LinkedList<>();

    int maxWidth = 1;
    Deque<Pair<TreeNode, Pair<Integer, Long>>> queue = new LinkedList<>();
    // 队列中的每个元素是：节点，(层级，索引)
    queue.offerLast(new Pair<>(root, new Pair<>(0, 0L)));

    while (!queue.isEmpty()) {
        Pair<TreeNode, Pair<Integer, Long>> pair = queue.pollFirst();
        TreeNode node = pair.first;
        int level = pair.second.first;
        long index = pair.second.second;

        // 如果是新的层级，初始化该层的最小和最大索引
        if (level == levelMinIndices.size()) {
            levelMinIndices.add(index);
            levelMaxIndices.add(index);
        } else {
            // 更新当前层的最小和最大索引
```

```
            levelMinIndices.set(level, Math.min(levelMinIndices.get(level), index));
            levelMaxIndices.set(level, Math.max(levelMaxIndices.get(level), index));
        }

        // 计算当前层的宽度
        long currentLevelMin = levelMinIndices.get(level);
        long currentLevelMax = levelMaxIndices.get(level);
        int currentWidth = (int) (currentLevelMax - currentLevelMin + 1);
        maxWidth = Math.max(maxWidth, currentWidth);

        // 子节点入队
        if (node.left != null) {
            // 重编号索引，避免溢出
            queue.offerLast(new Pair<>(node.left,
                new Pair<>(level + 1, 2 * (index - currentLevelMin))));
        }
        if (node.right != null) {
            queue.offerLast(new Pair<>(node.right,
                new Pair<>(level + 1, 2 * (index - currentLevelMin) + 1)));
        }
    }

    return maxWidth;
}

/**
 * 辅助类：用于存储两个元素的配对
 */
private static class Pair<K, V> {
    K first;
    V second;

    public Pair(K first, V second) {
        this.first = first;
        this.second = second;
    }
}

/**
 * 生成测试用例
 */
public static TreeNode generateTestTree(int type) {
    switch (type) {
```

```
        case 1: // 完全二叉树，LeetCode 示例
            TreeNode root1 = new TreeNode(1);
            root1.left = new TreeNode(3);
            root1.right = new TreeNode(2);
            root1.left.left = new TreeNode(5);
            root1.left.right = new TreeNode(3);
            root1.right.right = new TreeNode(9);
            return root1;
        case 2: // 只有左子节点，测试深层树
            TreeNode root2 = new TreeNode(1);
            TreeNode current = root2;
            // 创建一个深度为 10 的左链式树，测试索引溢出处理
            for (int i = 0; i < 10; i++) {
                current.left = new TreeNode(i + 2);
                current = current.left;
            }
            return root2;
        case 3: // 宽度为 1 的树，但有很多空节点
            TreeNode root3 = new TreeNode(1);
            root3.left = new TreeNode(2);
            root3.right = null;
            root3.left.left = new TreeNode(3);
            root3.left.right = null;
            root3.left.left.left = new TreeNode(4);
            return root3;
        case 4: // 平衡树
            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);
            return root4;
        default: // 简单树
            TreeNode root = new TreeNode(1);
            root.left = new TreeNode(3);
            root.right = new TreeNode(2);
            return root;
    }
}

/**
```

```
     * 打印树的层序表示
     */
    public static void printTree(TreeNode root) {
        if (root == null) {
            System.out.println("[null]");
            return;
        }

        Deque<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        StringBuilder sb = new StringBuilder();
        sb.append("[");

        boolean hasNonNull = true;
        while (!queue.isEmpty() && hasNonNull) {
            hasNonNull = false;
            int levelSize = queue.size();
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                if (node != null) {
                    sb.append(node.val);
                    queue.offer(node.left);
                    queue.offer(node.right);
                    if (node.left != null || node.right != null) {
                        hasNonNull = true;
                    }
                } else {
                    sb.append("null");
                    queue.offer(null);
                    queue.offer(null);
                }
                if (i < levelSize - 1 || (!queue.isEmpty() && hasNonNull)) {
                    sb.append(", ");
                }
            }
        }

        sb.append("]");
        System.out.println(sb.toString());
    }

    /**
     * 性能测试辅助方法
```

```
     * 用于测试不同实现方法在深层树情况下的性能
     */
    public static void performanceTest() {
        System.out.println("\n===== 性能测试 =====");

        // 创建深度为 1000 的左链式树，测试索引溢出问题
        TreeNode deepTree = new TreeNode(1);
        TreeNode current = deepTree;
        for (int i = 0; i < 1000; i++) {
            current.left = new TreeNode(i + 2);
            current = current.left;
        }

        // 测试优化 BFS 实现性能
        long startTime = System.nanoTime();
        int widthBFS = widthOfBinaryTree(deepTree);
        long endTime = System.nanoTime();
        System.out.println("优化 BFS 实现 - 宽度: " + widthBFS + ", 耗时: " +
                           (endTime - startTime) / 1_000_000.0 + " ms");

        // 测试 DFS 实现性能
        startTime = System.nanoTime();
        int widthDFS = widthOfBinaryTreeDFS(deepTree);
        endTime = System.nanoTime();
        System.out.println("DFS 实现 - 宽度: " + widthDFS + ", 耗时: " +
                           (endTime - startTime) / 1_000_000.0 + " ms");

        // 测试 HashMap 优化实现性能
        startTime = System.nanoTime();
        int widthHashMap = widthOfBinaryTreeHashMap(deepTree);
        endTime = System.nanoTime();
        System.out.println("HashMap 优化实现 - 宽度: " + widthHashMap + ", 耗时: " +
                           (endTime - startTime) / 1_000_000.0 + " ms");
    }

    public static void main(String[] args) {
        // 1. 标准测试用例
        System.out.println("===== 标准测试用例 =====");
        for (int i = 0; i <= 4; i++) {
            System.out.println("\n 测试树类型 " + i + ":");
            TreeNode root = generateTestTree(i);
            System.out.print("树结构: ");
            printTree(root);
```

```
    // 使用优化的BFS实现（避免索引溢出）
    int width1 = widthOfBinaryTree(root);
    System.out.println("优化BFS实现最大宽度: " + width1);

    // 使用DFS实现
    int width2 = widthOfBinaryTreeDFS(root);
    System.out.println("DFS实现最大宽度: " + width2);

    // 使用HashMap优化实现
    int width3 = widthOfBinaryTreeHashMap(root);
    System.out.println("HashMap优化实现最大宽度: " + width3);
}

// 2. 边界情况测试
System.out.println("\n===== 边界情况测试 =====");
System.out.println("空树最大宽度: " + widthOfBinaryTree(null));

TreeNode singleNode = new TreeNode(1);
System.out.println("单节点树最大宽度: " + widthOfBinaryTree(singleNode));

// 3. 斜树测试（全左子树或全右子树）
System.out.println("\n===== 斜树测试 =====");
TreeNode leftSkewed = new TreeNode(1);
current = leftSkewed;
for (int i = 0; i < 10; i++) {
    current.left = new TreeNode(i + 2);
    current = current.left;
}
System.out.print("全左斜树: ");
printTree(leftSkewed);
System.out.println("最大宽度: " + widthOfBinaryTree(leftSkewed));

TreeNode rightSkewed = new TreeNode(1);
current = rightSkewed;
for (int i = 0; i < 10; i++) {
    current.right = new TreeNode(i + 2);
    current = current.right;
}
System.out.print("全右斜树: ");
printTree(rightSkewed);
System.out.println("最大宽度: " + widthOfBinaryTree(rightSkewed));
```

```
        // 4. 完全二叉树测试
        System.out.println("\n===== 完全二叉树测试 =====");
        TreeNode completeTree = createCompleteBinaryTree(4); // 高度为4的完全二叉树
        System.out.print("完全二叉树: ");
        printTree(completeTree);
        System.out.println("最大宽度: " + widthOfBinaryTree(completeTree));

        // 5. 性能测试
        performanceTest();
    }

    /**
     * 创建指定高度的完全二叉树
     * 用于测试完全二叉树的宽度计算
     *
     * @param height 树的高度
     * @return 创建的完全二叉树
     */
    public static TreeNode createCompleteBinaryTree(int height) {
        if (height <= 0) {
            return null;
        }
        return createCompleteBinaryTreeHelper(1, height);
    }

    private static TreeNode createCompleteBinaryTreeHelper(int val, int height) {
        if (height == 1) {
            return new TreeNode(val);
        }
        TreeNode node = new TreeNode(val);
        node.left = createCompleteBinaryTreeHelper(2 * val, height - 1);
        node.right = createCompleteBinaryTreeHelper(2 * val + 1, height - 1);
        return node;
    }
}

/*
Python实现:

# 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

class WidthOfBinaryTree2:
    """
    二叉树最大宽度 - 解法2（优化版）
    专注于解决索引溢出问题并优化性能
    时间复杂度: O(N)
    空间复杂度: O(N)
    """
    def widthOfBinaryTree(self, root: TreeNode) -> int:
        """
        优化版本：避免索引溢出
        通过每层重编号的方式，减去当前层第一个节点的索引
        防止大深度树的索引溢出问题
        """
        if not root:
            return 0

        from collections import deque
        max_width = 1
        # 使用双端队列存储(节点, 索引)元组
        queue = deque([(root, 0)])  # 根节点索引为0

        while queue:
            level_size = len(queue)
            # 获取当前层第一个节点的索引作为基准
            level_start = queue[0][1]
            first = last = 0

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

                # 重编号索引，避免溢出
                relative_index = index - level_start

                # 记录第一个和最后一个节点的相对索引
                if i == 0:
                    first = relative_index
                if i == level_size - 1:
                    last = relative_index

                # 子节点入队，使用原始索引计算子节点索引
```

```
            if node.left:
                queue.append((node.left, 2 * index))
            if node.right:
                queue.append((node.right, 2 * index + 1))

        # 更新最大宽度
        max_width = max(max_width, last - first + 1)

    return max_width

def widthOfBinaryTreeDFS(self, root: TreeNode) -> int:
    """
    使用 DFS 计算二叉树的最大宽度
    通过记录每层最左侧节点的索引，计算当前节点的相对位置
    """
    if not root:
        return 0

    # 存储每层最左侧节点的索引
    level_leftmost = []
    max_width = [0]  # 使用列表作为引用传递

    def dfs(node, level, index):
        if not node:
            return

        # 如果是新的层级，记录第一个节点的索引
        if level == len(level_leftmost):
            level_leftmost.append(index)

        # 计算相对索引，更新最大宽度
        level_start = level_leftmost[level]
        relative_index = index - level_start
        max_width[0] = max(max_width[0], relative_index + 1)

        # 递归处理左右子树
        dfs(node.left, level + 1, 2 * index)
        dfs(node.right, level + 1, 2 * index + 1)

    dfs(root, 0, 0)
    return max_width[0]

def widthOfBinaryTreeHashMap(self, root: TreeNode) -> int:
```

```
    """
    使用 HashMap 优化的实现，记录每层的最小和最大索引
    适合内存受限环境
    """
    if not root:
        return 0

    from collections import deque, defaultdict
    # 使用字典记录每层的最小和最大索引
    min_indices = defaultdict(lambda: float('inf'))
    max_indices = defaultdict(lambda: float('-inf'))
    max_width = 1

    # 队列中存储(节点, (层级, 索引))
    queue = deque([(root, (0, 0))])

    while queue:
        node, (level, index) = queue.popleft()

        # 更新当前层的最小和最大索引
        min_indices[level] = min(min_indices[level], index)
        max_indices[level] = max(max_indices[level], index)

        # 计算当前层的宽度
        current_width = max_indices[level] - min_indices[level] + 1
        max_width = max(max_width, current_width)

        # 子节点入队，使用相对索引避免溢出
        level_start = min_indices[level]
        if node.left:
            queue.append((node.left, (level + 1, 2 * (index - level_start))))
        if node.right:
            queue.append((node.right, (level + 1, 2 * (index - level_start) + 1)))

    return max_width

def generateTestTree(self, tree_type: int) -> TreeNode:
    """
    生成不同类型的测试树
    """
    if tree_type == 1:  # LeetCode 示例
        root = TreeNode(1)
        root.left = TreeNode(3)
```

```
        root.right = TreeNode(2)
        root.left.left = TreeNode(5)
        root.left.right = TreeNode(3)
        root.right.right = TreeNode(9)
        return root
    elif tree_type == 2:  # 深层左链式树
        root = TreeNode(1)
        current = root
        for i in range(10):
            current.left = TreeNode(i + 2)
            current = current.left
        return root
    elif tree_type == 3:  # 宽度为1的树
        root = TreeNode(1)
        root.left = TreeNode(2)
        root.right = None
        root.left.left = TreeNode(3)
        root.left.right = None
        root.left.left.left = TreeNode(4)
        return root
    elif tree_type == 4:  # 平衡树
        root = TreeNode(1)
        root.left = TreeNode(2)
        root.right = TreeNode(3)
        root.left.left = TreeNode(4)
        root.left.right = TreeNode(5)
        root.right.left = TreeNode(6)
        root.right.right = TreeNode(7)
        return root
    else:  # 简单树
        root = TreeNode(1)
        root.left = TreeNode(3)
        root.right = TreeNode(2)
        return root


def printTree(self, root: TreeNode) -> None:
    """
    打印树的层序表示
    """
    if not root:
        print("[null]")
        return
```

```
        from collections import deque
        queue = deque([root])
        result = []
        has_non_null = True

        while queue and has_non_null:
            has_non_null = False
            level_size = len(queue)
            for _ in range(level_size):
                node = queue.popleft()
                if node:
                    result.append(str(node.val))
                    queue.append(node.left)
                    queue.append(node.right)
                    if node.left or node.right:
                        has_non_null = True
                else:
                    result.append("null")
                    queue.append(None)
                    queue.append(None)

        # 移除末尾多余的 null
        while result and result[-1] == "null":
            result.pop()

        print("[" + ", ".join(result) + "]")

# 测试代码
if __name__ == "__main__":
    solution = WidthOfBinaryTree2()

    # 测试不同类型的树
    for i in range(5):
        print(f"\n 测试树类型 {i}:")
        root = solution.generateTestTree(i)
        print("树结构:", end=" ")
        solution.printTree(root)

        # 使用优化的 BFS 实现
        width1 = solution.widthOfBinaryTree(root)
        print(f"优化 BFS 实现最大宽度: {width1}")

        # 使用 DFS 实现
```

```
        width2 = solution.widthOfBinaryTreeDFS(root)
        print(f"DFS 实现最大宽度: {width2}")

        # 使用 HashMap 优化实现
        width3 = solution.widthOfBinaryTreeHashMap(root)
        print(f"HashMap 优化实现最大宽度: {width3}")

    # 边界情况测试
    print("\n 边界情况测试:")
    print(f"空树最大宽度: {solution.widthOfBinaryTree(None)}")

    single_node = TreeNode(1)
    print(f"单节点树最大宽度: {solution.widthOfBinaryTree(single_node)}")
```

C++实现:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <string>
#include <unordered_map>
#include <climits>
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 WidthOfBinaryTree2 {
public:
    /**
     * 方法 1：避免索引溢出的优化实现
     * 使用队列存储节点和索引，通过每层重编号避免溢出
     * 时间复杂度: O(N)
     * 空间复杂度: O(N)
     */
```

```
int widthOfBinaryTree(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int maxWidth = 1;
    // 使用队列存储节点和索引的 pair
    queue<pair<TreeNode*, long long>> q;
    q.push({root, 0LL});  // 根节点索引设为 0

    while (!q.empty()) {
        int levelSize = q.size();
        // 获取当前层第一个节点的索引作为基准
        long long levelStart = q.front().second;
        long long first = 0, last = 0;

        for (int i = 0; i < levelSize; ++i) {
            auto [node, index] = q.front();
            q.pop();

            // 重编号索引，减去当前层起始索引
            long long currentIndex = index - levelStart;

            // 记录当前层的第一个和最后一个节点的重编号索引
            if (i == 0) {
                first = currentIndex;
            }
            if (i == levelSize - 1) {
                last = currentIndex;
            }

            // 子节点入队，使用原始索引计算子节点索引
            if (node->left) {
                q.push({node->left, 2 * index});
            }
            if (node->right) {
                q.push({node->right, 2 * index + 1});
            }
        }

        // 更新最大宽度
        maxWidth = max(maxWidth, static_cast<int>(last - first + 1));
    }
```

```
        return maxWidth;
    }

    /**
     * 方法 2：使用 DFS 实现计算二叉树宽度
     */
    int widthOfBinaryTreeDFS(TreeNode* root) {
        if (!root) {
            return 0;
        }

        vector<long long> levelLeftmost;
        int maxWidth = 0;
        dfs(root, 0, 0, levelLeftmost, maxWidth);
        return maxWidth;
    }

private:
    /**
     * DFS 递归辅助函数
     */
    void dfs(TreeNode* node, int level, long long index,
             vector<long long>& levelLeftmost, int& maxWidth) {
        if (!node) {
            return;
        }

        // 如果是该层的第一个访问的节点，记录其索引
        if (level == levelLeftmost.size()) {
            levelLeftmost.push_back(index);
        }

        // 获取当前层最左侧节点的索引
        long long levelStart = levelLeftmost[level];

        // 计算当前节点相对于最左侧节点的位置宽度
        long long relativeIndex = index - levelStart;

        // 更新最大宽度
        maxWidth = max(maxWidth, static_cast<int>(relativeIndex + 1));

        // 递归处理左子树和右子树
```

```
        dfs(node->left, level + 1, 2 * index, levelLeftmost, maxWidth);
        dfs(node->right, level + 1, 2 * index + 1, levelLeftmost, maxWidth);
    }

public:
    /**
     * 方法 3：使用 unordered_map 优化的层序遍历实现
     */
    int widthOfBinaryTreeHashMap(TreeNode* root) {
        if (!root) {
            return 0;
        }

        // 使用 unordered_map 记录每层的最小和最大索引
        unordered_map<int, long long> minIndices;
        unordered_map<int, long long> maxIndices;
        int maxWidth = 1;

        // 队列中的每个元素是：节点，(层级，索引)
        queue<pair<TreeNode*, pair<int, long long>>> q;
        q.push({root, {0, 0LL}});

        while (!q.empty()) {
            auto [node, levelIndexPair] = q.front();
            q.pop();

            int level = levelIndexPair.first;
            long long index = levelIndexPair.second;

            // 更新当前层的最小和最大索引
            if (minIndices.find(level) == minIndices.end() || index < minIndices[level]) {
                minIndices[level] = index;
            }
            if (maxIndices.find(level) == maxIndices.end() || index > maxIndices[level]) {
                maxIndices[level] = index;
            }

            // 计算当前层的宽度
            long long currentLevelMin = minIndices[level];
            long long currentLevelMax = maxIndices[level];
            int currentWidth = static_cast<int>(currentLevelMax - currentLevelMin + 1);
            maxWidth = max(maxWidth, currentWidth);
```

```
        // 子节点入队，使用相对索引避免溢出
        if (node->left) {
            q.push({node->left, {level + 1, 2 * (index - currentLevelMin)}});
        }
        if (node->right) {
            q.push({node->right, {level + 1, 2 * (index - currentLevelMin) + 1}});
        }
    }

    return maxWidth;
}

/**
 * 生成测试用例
 */
TreeNode* generateTestTree(int treeType) {
    switch (treeType) {
        case 1: {
            // LeetCode 示例
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->right = new TreeNode(2);
            root->left->left = new TreeNode(5);
            root->left->right = new TreeNode(3);
            root->right->right = new TreeNode(9);
            return root;
        }
        case 2: {
            // 深层左链式树
            TreeNode* root = new TreeNode(1);
            TreeNode* current = root;
            for (int i = 0; i < 10; ++i) {
                current->left = new TreeNode(i + 2);
                current = current->left;
            }
            return root;
        }
        case 3: {
            // 宽度为 1 的树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(2);
            root->right = nullptr;
            root->left->left = new TreeNode(3);
```

```
            root->left->right = nullptr;
            root->left->left->left = new TreeNode(4);
            return root;
        }
        case 4: {
            // 平衡树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(2);
            root->right = new TreeNode(3);
            root->left->left = new TreeNode(4);
            root->left->right = new TreeNode(5);
            root->right->left = new TreeNode(6);
            root->right->right = new TreeNode(7);
            return root;
        }
        default: {
            // 简单树
            TreeNode* root = new TreeNode(1);
            root->left = new TreeNode(3);
            root->right = new TreeNode(2);
            return root;
        }
    }
}

/**
 * 释放树内存
 */
void deleteTree(TreeNode* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
}

/**
 * 打印树的层序表示
 */
void printTree(TreeNode* root) {
    if (!root) {
        cout << "[null]" << endl;
        return;
```

```
    }

    vector<string> result;
    queue<TreeNode*> q;
    q.push(root);
    bool hasNonNull = true;

    while (!q.empty() && hasNonNull) {
        hasNonNull = false;
        int levelSize = q.size();
        for (int i = 0; i < levelSize; ++i) {
            TreeNode* node = q.front();
            q.pop();

            if (node) {
                result.push_back(to_string(node->val));
                q.push(node->left);
                q.push(node->right);
                if (node->left || node->right) {
                    hasNonNull = true;
                }
            } else {
                result.push_back("null");
                q.push(nullptr);
                q.push(nullptr);
            }
        }
    }

    // 移除末尾多余的 null
    while (!result.empty() && result.back() == "null") {
        result.pop_back();
    }

    cout << "[";
    for (size_t i = 0; i < result.size(); ++i) {
        cout << result[i];
        if (i < result.size() - 1) {
            cout << ", ";
        }
    }
    cout << "]" << endl;
}
```

```
};

// 测试代码
int main() {
    WidthOfBinaryTree2 solution;

    // 测试不同类型的树
    for (int i = 0; i < 5; ++i) {
        cout << "\n测试树类型 " << i << ":" << endl;
        TreeNode* root = solution.generateTestTree(i);
        cout << "树结构: ";
        solution.printTree(root);

        // 使用优化的BFS实现
        int width1 = solution.widthOfBinaryTree(root);
        cout << "优化BFS实现最大宽度: " << width1 << endl;

        // 使用DFS实现
        int width2 = solution.widthOfBinaryTreeDFS(root);
        cout << "DFS实现最大宽度: " << width2 << endl;

        // 使用HashMap优化实现
        int width3 = solution.widthOfBinaryTreeHashMap(root);
        cout << "HashMap优化实现最大宽度: " << width3 << endl;

        // 释放内存
        solution.deleteTree(root);
    }

    // 边界情况测试
    cout << "\n边界情况测试:" << endl;
    cout << "空树最大宽度: " << solution.widthOfBinaryTree(nullptr) << endl;

    TreeNode* singleNode = new TreeNode(1);
    cout << "单节点树最大宽度: " << solution.widthOfBinaryTree(singleNode) << endl;
    solution.deleteTree(singleNode);

    return 0;
}
*/

==========
```

文件: Code04_DepthOfBinaryTree.java

```
==========================================
package class036;

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

/**
 * 二叉树深度相关算法实现
 * 包括：
 * 1. 最大深度计算（递归和非递归实现）
 * 2. 最小深度计算（递归和非递归实现）
 * 3. 平衡二叉树判断
 * 4. 树的高度计算
 * 5. 二叉树直径计算
 *
 * 核心算法思想：
 * - 最大深度：根节点到最远叶子节点的最长路径上的节点数
 * - 最小深度：根节点到最近叶子节点的最短路径上的节点数
 * - 平衡二叉树：任意节点的左右子树高度差不超过 1
 * - 二叉树直径：树中任意两个节点之间最长路径的长度
 *
 * 时间复杂度分析：
 * - 递归 DFS 解法：O(N)，其中 N 是树中的节点数，每个节点只访问一次
 * - 迭代 BFS 解法：O(N)
 * - 迭代 DFS 解法：O(N)
 *
 * 空间复杂度分析：
 * - 递归 DFS：O(H)，H 为树的高度，最坏情况下（斜树）为 O(N)
 * - 迭代 BFS：O(W)，W 为树的最大宽度，最坏情况下为 O(N)
 * - 迭代 DFS：O(H)
 *
 * 工程化考量：
 * - 针对不同应用场景选择合适的实现方法
 * - 对于完全二叉树，BFS 可能更高效
 * - 对于不平衡的树，DFS 可能更节省空间
 * - 递归实现简洁但可能有栈溢出风险
 * - 非递归实现更健壮，适合深层树
 *
 * 算法优化要点：
 * - 最小深度计算中，BFS 可以在找到第一个叶子节点时立即返回，提高效率
 * - 平衡二叉树判断采用后序遍历，实现剪枝优化
```

 * - 直径计算与高度计算结合，避免重复计算
 *
 * 跨语言实现差异：
 * - Java：使用递归、队列、栈等标准数据结构
 * - Python：代码更简洁，使用元组存储节点和深度
 * - C++：需要手动管理内存，提供 deleteTree 方法释放资源
 *
 * 相关题目：
 * 1. LeetCode 104. 二叉树的最大深度 - https://leetcode.cn/problems/maximum-depth-of-binary-tree/
 * 2. LeetCode 111. 二叉树的最小深度 - https://leetcode.cn/problems/minimum-depth-of-binary-tree/
 * 3. LeetCode 110. 平衡二叉树 - https://leetcode.cn/problems/balanced-binary-tree/
 * 4. LeetCode 543. 二叉树的直径 - https://leetcode.cn/problems/diameter-of-binary-tree/
 * 5. LintCode 97. 二叉树的最大深度 - https://www.lintcode.com/problem/97
 * 6. HackerRank Tree: Height of a Binary Tree - https://www.hackerrank.com/challenges/tree-height-of-a-binary-tree
 * 7. UVA 12455 - Bars
 * 8. CodeChef - TREE2
 * 9. LeetCode 563. 二叉树的坡度 - https://leetcode.cn/problems/binary-tree-tilt/
 * 10. LeetCode 124. 二叉树中的最大路径和 - https://leetcode.cn/problems/binary-tree-maximum-path-sum/
 * 11. LeetCode 687. 最长同值路径 - https://leetcode.cn/problems/longest-univalue-path/
 * 12. LeetCode 958. 二叉树的完全性检验 - https://leetcode.cn/problems/check-completeness-of-a-binary-tree/
 * 13. LeetCode 1302. 层数最深叶子节点的和 - https://leetcode.cn/problems/deepest-leaves-sum/
 * 14. HackerRank Balanced Brackets - https://www.hackerrank.com/challenges/balanced-brackets
 * 15. GeeksforGeeks Check if a binary tree is height balanced - https://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/
 * 16. Codeforces 779B - Weird Journey - https://codeforces.com/problemset/problem/779/B
 * 17. AtCoder ABC129D - Lamp - https://atcoder.jp/contests/abc129/tasks/abc129_d
 * 18. SPOJ PT07Z - Longest path in a tree - https://www.spoj.com/problems/PT07Z/
 * 19. UVa 10942 - Is This the Easiest Problem? - https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=1883
 * 20. USACO 2017 January Contest, Bronze - Problem 3. Cow Tipping - http://www.usaco.org/index.php?page=viewproblem2&cpid=688
 * 21. 牛客网 NC137 表达式求值 - https://www.nowcoder.com/practice/c215ba61c8b1443b996351df929dc4d4
 * 22. 洛谷 P1028 数的计算 - https://www.luogu.com.cn/problem/P1028
 * 23. 杭电 OJ 1115 - Lifting the Stone - http://acm.hdu.edu.cn/showproblem.php?pid=1115
 * 24. AizuOJ ALDS1_7_C - Tree Walk - https://onlinejudge.u-aizu.ac.jp/problems/ALDS1_7_C
 */
public class Code04_DepthOfBinaryTree {

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

    public TreeNode() {}

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

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

    @Override
    public String toString() {
        return "TreeNode{" + "val=" + val + "}";
    }
}

/**
 * 计算二叉树的最大深度 - 递归DFS解法
 * 实现思路：
 * - 基本情况：空节点深度为0
 * - 递归情况：树的最大深度 = max(左子树深度, 右子树深度) + 1
 * - 采用分治思想，将大问题分解为子问题
 *
 * 优点：代码简洁易懂，实现高效
 * 缺点：对于极深的树可能导致栈溢出
 *
 * 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，最坏情况（斜树）为O(N)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最大深度
 */
// 测试链接 : https://leetcode.cn/problems/maximum-depth-of-binary-tree/
public static int maxDepth(TreeNode root) {
    // 基本情况：空节点深度为0
```

```
    return root == null ? 0 : Math.max(maxDepth(root.left), maxDepth(root.right)) + 1;
}

/**
 * 计算二叉树的最大深度 - 迭代 BFS 解法
 * 实现思路：
 * - 使用队列进行层序遍历
 * - 记录每层的节点数量，每处理完一层，深度加 1
 * - 适用于需要按层处理的场景
 *
 * 优点：避免递归栈溢出问题，适用于深层树
 * 缺点：空间复杂度可能高于 DFS，因为需要存储整层的节点
 *
 * 时间复杂度：O(N)，每个节点只访问一次
 * 空间复杂度：O(W)，W 为树的最大宽度，最坏情况为 O(N)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最大深度
 */
public static int maxDepthBFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

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

    while (!queue.isEmpty()) {
        int levelSize = queue.size(); // 当前层的节点数

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

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

```
        // 每处理完一层，深度加 1
        depth++;
    }

    return depth;
}

/**
 * 计算二叉树的最大深度 - 迭代 DFS 解法
 * 实现思路：
 * - 使用栈模拟递归过程
 * - 每个栈元素存储节点和对应的深度
 * - 先压入右子节点，再压入左子节点，确保左子树先被处理
 *
 * 优点：避免递归栈溢出问题，适用于深层树
 * 缺点：代码复杂度略高
 *
 * 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，最坏情况为 O(N)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最大深度
 */
public static int maxDepthDFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int maxDepth = 0;
    // 使用栈存储节点和对应的深度
    Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
    stack.push(new Pair<>(root, 1));

    while (!stack.isEmpty()) {
        Pair<TreeNode, Integer> pair = stack.pop();
        TreeNode node = pair.first;
        int depth = pair.second;

        // 更新最大深度
        maxDepth = Math.max(maxDepth, depth);

        // 先压右子节点，再压左子节点，确保左子节点先被处理
```

```
        if (node.right != null) {
            stack.push(new Pair<>(node.right, depth + 1));
        }
        if (node.left != null) {
            stack.push(new Pair<>(node.left, depth + 1));
        }
    }

    return maxDepth;
}

/**
 * 计算二叉树的最小深度 - 递归DFS解法
 * 实现思路：
 * - 基本情况：空节点深度为0
 * - 叶子节点深度为1
 * - 对于非叶子节点，递归计算左右子树的最小深度
 * - 注意：需要处理只有一侧子树的情况
 *
 * 关键点：最小深度是从根节点到最近叶子节点的最短路径上的节点数量
 * 叶子节点定义：没有子节点的节点
 *
 * 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，最坏情况为O(N)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最小深度
 */
// 测试链接 : https://leetcode.cn/problems/minimum-depth-of-binary-tree/
public int minDepth(TreeNode root) {
    if (root == null) {
        // 当前的树是空树
        return 0;
    }
    if (root.left == null && root.right == null) {
        // 当前root是叶节点
        return 1;
    }
    int ldeep = Integer.MAX_VALUE;
    int rdeep = Integer.MAX_VALUE;
    if (root.left != null) {
        ldeep = minDepth(root.left);
    }
```

```
    if (root.right != null) {
        rdeep = minDepth(root.right);
    }
    return Math.min(ldeep, rdeep) + 1;
}

/**
 * 计算二叉树的最小深度 - 优化递归解法
 * 实现思路：
 * - 基本情况：空节点深度为0
 * - 优化处理单侧子树情况：如果左子树为空，返回右子树深度+1；反之亦然
 * - 当左右子树都不为空时，取较小深度+1
 *
 * 优点：代码更简洁，逻辑更清晰
 * 缺点：同样存在递归栈溢出风险
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(H)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最小深度
 */
public static int minDepthRecursive(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 如果左子树为空，则返回右子树的最小深度+1
    if (root.left == null) {
        return minDepthRecursive(root.right) + 1;
    }

    // 如果右子树为空，则返回左子树的最小深度+1
    if (root.right == null) {
        return minDepthRecursive(root.left) + 1;
    }

    // 如果左右子树都不为空，取较小值+1
    return Math.min(minDepthRecursive(root.left), minDepthRecursive(root.right)) + 1;
}

/**
 * 计算二叉树的最小深度 - 迭代BFS解法（最优解）
```

```
     * 实现思路：
     * - 使用队列进行层序遍历
     * - 记录当前深度，从 1 开始递增
     * - 一旦找到叶子节点，立即返回当前深度，无需遍历整棵树
     *
     * 为什么是最优解：
     * - BFS 保证最先找到的叶子节点一定是最近的叶子节点
     * - 一旦找到目标可以立即返回，不会不必要地遍历深层节点
     * - 对于不平衡的树，效率远高于 DFS 实现
     *
     * 时间复杂度：O(N)，最坏情况下访问所有节点，但平均情况表现更好
     * 空间复杂度：O(W)，W 为树的最大宽度
     *
     * @param root 二叉树的根节点
     * @return 二叉树的最小深度
     */
    public static int minDepthBFS(TreeNode root) {
        if (root == null) {
            return 0;
        }

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

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

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

                // 如果是叶子节点，直接返回当前深度
                if (node.left == null && node.right == null) {
                    return depth;
                }

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

```
        }
    }

    return depth; // 不会到达这里
}

/**
 * 计算二叉树的最小深度 - 迭代DFS解法
 * 实现思路：
 * - 使用栈模拟递归过程
 * - 每个栈元素存储节点和对应的深度
 * - 遇到叶子节点时更新最小深度
 * - 需要遍历完所有可能的路径才能确定最小深度
 *
 * 优点：避免递归栈溢出问题
 * 缺点：对于大多数情况，效率低于BFS实现，因为可能需要遍历整棵树
 *
 * 时间复杂度：O(N)
 * 空间复杂度：O(H)
 *
 * @param root 二叉树的根节点
 * @return 二叉树的最小深度
 */
public static int minDepthDFS(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int minDepth = Integer.MAX_VALUE;
    Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
    stack.push(new Pair<>(root, 1));

    while (!stack.isEmpty()) {
        Pair<TreeNode, Integer> pair = stack.pop();
        TreeNode node = pair.first;
        int depth = pair.second;

        // 如果是叶子节点，更新最小深度
        if (node.left == null && node.right == null) {
            minDepth = Math.min(minDepth, depth);
        }

        // 继续遍历子节点
```

```
            if (node.right != null) {
                stack.push(new Pair<>(node.right, depth + 1));
            }
            if (node.left != null) {
                stack.push(new Pair<>(node.left, depth + 1));
            }
        }

        return minDepth;
    }

    /**
     * 判断二叉树是否是平衡二叉树
     * 实现思路：
     * - 平衡二叉树定义：任意节点的左右子树高度差不超过 1
     * - 使用后序遍历策略，自底向上计算高度
     * - 如果在计算过程中发现不平衡，可以提前返回，实现剪枝
     *
     * 优化点：
     * - 一旦发现不平衡，立即返回-1，避免不必要的计算
     * - 同时计算高度和判断平衡性，避免多次遍历
     *
     * 时间复杂度：O(N)
     * 空间复杂度：O(H)
     *
     * @param root 二叉树的根节点
     * @return 如果是平衡二叉树返回 true，否则返回 false
     */
    public static boolean isBalanced(TreeNode root) {
        return height(root) != -1;
    }

    /**
     * 辅助方法：计算树的高度，如果不是平衡树则返回-1
     * 实现思路：
     * - 递归计算左右子树的高度
     * - 在递归过程中检查平衡性
     * - 如果左子树或右子树不平衡，直接返回-1
     * - 如果当前节点不平衡（左右子树高度差>1），返回-1
     * - 否则返回树的高度
     *
     * 时间复杂度：O(N)
     * 空间复杂度：O(H)
```

```
     *
     * @param root 二叉树的根节点
     * @return 树的高度，如果不是平衡树则返回-1
     */
    private static int height(TreeNode root) {
        if (root == null) {
            return 0;
        }

        // 递归计算左右子树的高度
        int leftHeight = height(root.left);
        if (leftHeight == -1) {
            return -1; // 左子树不平衡
        }

        int rightHeight = height(root.right);
        if (rightHeight == -1) {
            return -1; // 右子树不平衡
        }

        // 检查当前节点是否平衡
        if (Math.abs(leftHeight - rightHeight) > 1) {
            return -1; // 不平衡
        }

        return Math.max(leftHeight, rightHeight) + 1;
    }

    /**
     * 计算二叉树的直径（最长路径长度）
     * 实现思路：
     * - 直径定义：树中任意两个节点之间的最长路径的长度
     * - 最长路径可能经过根节点，也可能不经过根节点
     * - 使用递归计算每个子树的高度，并在递归过程中更新直径
     * - 对于每个节点，其作为路径最高点的路径长度 = 左子树高度 + 右子树高度
     *
     * 关键点：
     * - 直径不一定要经过根节点
     * - 需要在遍历过程中记录全局最大值
     *
     * 时间复杂度：O(N)
     * 空间复杂度：O(H)
     *
```

```
     * @param root 二叉树的根节点
     * @return 二叉树的直径
     */
    public static int diameterOfBinaryTree(TreeNode root) {
        int[] diameter = new int[1]; // 使用数组保存直径，便于在递归中修改
        calculateDiameter(root, diameter);
        return diameter[0];
    }

    /**
     * 辅助方法：计算树的高度并更新直径
     * 实现思路：
     * - 递归计算左右子树的高度
     * - 对于当前节点，计算可能的最长路径：左子树高度 + 右子树高度
     * - 更新全局直径最大值
     * - 返回当前子树的高度
     *
     * 使用数组保存直径的原因：
     * - 数组在 Java 中是引用类型，可以在递归过程中修改其内容
     * - 相比使用成员变量，这种方式更封装，不影响类的状态
     *
     * 时间复杂度：O(N)
     * 空间复杂度：O(H)
     *
     * @param root 二叉树的根节点
     * @param diameter 保存直径的数组
     * @return 当前子树的高度
     */
    private static int calculateDiameter(TreeNode root, int[] diameter) {
        if (root == null) {
            return 0;
        }

        int leftHeight = calculateDiameter(root.left, diameter);
        int rightHeight = calculateDiameter(root.right, diameter);

        // 更新直径：左子树高度 + 右子树高度
        diameter[0] = Math.max(diameter[0], leftHeight + rightHeight);

        return Math.max(leftHeight, rightHeight) + 1;
    }

    /**
```

```
     * 辅助类：用于存储节点和对应的值（如深度）
     * 泛型设计提高了代码的复用性
     */
    private static class Pair<K, V> {
        K first;
        V second;

        public Pair(K first, V second) {
            this.first = first;
            this.second = second;
        }
    }

    /**
     * 生成测试用例
     * 用于生成不同类型的树以测试各种算法
     *
     * @param type 树类型：
     * - 0: 普通二叉树
     * - 1: 完全二叉树
     * - 2: 左倾斜树
     * - 3: 右倾斜树
     * - 4: 单节点树
     * @return 测试用的二叉树
     */
    public static TreeNode generateTestTree(int type) {
        switch (type) {
            case 1: // 完全二叉树
                //      1
                //    /   \
                //   2     3
                //  / \   / \
                // 4   5 6   7
                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);
                root1.right.left = new TreeNode(6);
                root1.right.right = new TreeNode(7);
                return root1;
            case 2: // 左倾斜树（用于测试最小深度）
                //    1
```

```
            //   /
            //  2
            // /
            // 3
            TreeNode root2 = new TreeNode(1);
            root2.left = new TreeNode(2);
            root2.left.left = new TreeNode(3);
            return root2;
        case 3: // 右倾斜树（用于测试最小深度）
            // 1
            //  \
            //   2
            //    \
            //     3
            TreeNode root3 = new TreeNode(1);
            root3.right = new TreeNode(2);
            root3.right.right = new TreeNode(3);
            return root3;
        case 4: // 单节点树
            return new TreeNode(1);
        default: // 普通二叉树
            //      1
            //    /   \
            //   2     3
            //  /       \
            // 4         5
            //           \
            //            6
            TreeNode root = new TreeNode(1);
            root.left = new TreeNode(2);
            root.right = new TreeNode(3);
            root.left.left = new TreeNode(4);
            root.right.right = new TreeNode(5);
            root.right.right.right = new TreeNode(6);
            return root;
    }
}

/**
 * 性能测试方法
 * 用于比较不同算法在深层树情况下的性能表现
 */
public static void performanceTest() {
```

```
System.out.println("\n===== 性能测试 =====");

// 创建一个深度为 1000 的左链式树
TreeNode deepTree = new TreeNode(1);
TreeNode current = deepTree;
for (int i = 0; i < 1000; i++) {
    current.left = new TreeNode(i + 2);
    current = current.left;
}

// 测试最大深度算法性能
System.out.println("\n 最大深度算法性能测试 (深层树):");

long startTime = System.nanoTime();
int maxDepth1 = maxDepth(deepTree);
long endTime = System.nanoTime();
System.out.println("递归 DFS - 结果: " + maxDepth1 + ", 耗时: " +
                   (endTime - startTime) / 1_000_000.0 + " ms");

startTime = System.nanoTime();
int maxDepth2 = maxDepthBFS(deepTree);
endTime = System.nanoTime();
System.out.println("迭代 BFS - 结果: " + maxDepth2 + ", 耗时: " +
                   (endTime - startTime) / 1_000_000.0 + " ms");

startTime = System.nanoTime();
int maxDepth3 = maxDepthDFS(deepTree);
endTime = System.nanoTime();
System.out.println("迭代 DFS - 结果: " + maxDepth3 + ", 耗时: " +
                   (endTime - startTime) / 1_000_000.0 + " ms");

// 测试最小深度算法性能
System.out.println("\n 最小深度算法性能测试 (深层树):");

startTime = System.nanoTime();
int minDepth1 = minDepthRecursive(deepTree);
endTime = System.nanoTime();
System.out.println("优化递归 - 结果: " + minDepth1 + ", 耗时: " +
                   (endTime - startTime) / 1_000_000.0 + " ms");

startTime = System.nanoTime();
int minDepth2 = minDepthBFS(deepTree);
endTime = System.nanoTime();
```

```
        System.out.println("迭代BFS - 结果: " + minDepth2 + ", 耗时: " +
                           (endTime - startTime) / 1_000_000.0 + " ms");

        startTime = System.nanoTime();
        int minDepth3 = minDepthDFS(deepTree);
        endTime = System.nanoTime();
        System.out.println("迭代DFS - 结果: " + minDepth3 + ", 耗时: " +
                           (endTime - startTime) / 1_000_000.0 + " ms");
    }

    /**
     * 打印树结构辅助方法
     * 用于调试和可视化树的结构
     */
    public static void printTree(TreeNode root) {
        if (root == null) {
            System.out.println("[]");
            return;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        StringBuilder sb = new StringBuilder("[");

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                if (node != null) {
                    sb.append(node.val);
                    queue.offer(node.left);
                    queue.offer(node.right);
                } else {
                    sb.append("null");
                }
                if (i < levelSize - 1) {
                    sb.append(", ");
                }
            }
            // 检查队列是否全为null
            boolean allNull = true;
            for (TreeNode node : queue) {
                if (node != null) {
```

```
                allNull = false;
                break;
            }
        }
        if (!allNull) {
            sb.append(", ");
        }
    }

    // 移除末尾的 null 值
    int end = sb.length() - 1;
    while (end >= 0 && (sb.charAt(end) == 'l' || sb.charAt(end) == 'n' ||
                        sb.charAt(end) == ' ' || sb.charAt(end) == ',')) {
        end--;
    }
    sb.setLength(end + 1);
    sb.append("]");
    System.out.println(sb.toString());
}

/**
 * 主方法：测试所有深度相关算法
 * 提供完整的测试覆盖，包括不同类型的树和边界情况
 */
public static void main(String[] args) {
    System.out.println("二叉树深度相关算法测试");
    System.out.println("====================\n");

    // 1. 标准测试用例 - 不同类型的树
    System.out.println("===== 标准测试用例 =====");
    for (int treeType = 0; treeType <= 4; treeType++) {
        System.out.println("\n 测试树类型 " + treeType + ":");
        TreeNode root = generateTestTree(treeType);
        System.out.print("树结构: ");
        printTree(root);

        // 最大深度测试
        int maxDepth1 = maxDepth(root);
        int maxDepth2 = maxDepthBFS(root);
        int maxDepth3 = maxDepthDFS(root);
        System.out.println("最大深度 (递归 DFS): " + maxDepth1);
        System.out.println("最大深度 (迭代 BFS): " + maxDepth2);
        System.out.println("最大深度 (迭代 DFS): " + maxDepth3);
```

```
    // 最小深度测试
    Code04_DepthOfBinaryTree instance = new Code04_DepthOfBinaryTree();
    int minDepth1 = instance.minDepth(root);
    int minDepth2 = minDepthRecursive(root);
    int minDepth3 = minDepthBFS(root);
    int minDepth4 = minDepthDFS(root);
    System.out.println("最小深度 (原递归): " + minDepth1);
    System.out.println("最小深度 (优化递归): " + minDepth2);
    System.out.println("最小深度 (迭代 BFS): " + minDepth3);
    System.out.println("最小深度 (迭代 DFS): " + minDepth4);

    // 平衡树测试
    boolean isBalanced = isBalanced(root);
    System.out.println("是否是平衡二叉树: " + isBalanced);

    // 直径测试
    int diameter = diameterOfBinaryTree(root);
    System.out.println("二叉树的直径: " + diameter);
}

// 2. 边界情况测试
System.out.println("\n===== 边界情况测试 =====");
TreeNode nullRoot = null;
System.out.println("空树最大深度: " + maxDepth(nullRoot));
System.out.println("空树最小深度: " + minDepthRecursive(nullRoot));
System.out.println("空树是否平衡: " + isBalanced(nullRoot));
System.out.println("空树直径: " + diameterOfBinaryTree(nullRoot));

// 3. 单节点树测试
System.out.println("\n===== 单节点树测试 =====");
TreeNode singleNode = new TreeNode(5);
System.out.print("树结构: ");
printTree(singleNode);
System.out.println("最大深度: " + maxDepth(singleNode));
System.out.println("最小深度: " + minDepthRecursive(singleNode));
System.out.println("是否平衡: " + isBalanced(singleNode));
System.out.println("直径: " + diameterOfBinaryTree(singleNode));

// 4. 深层树测试
System.out.println("\n===== 深层树测试 =====");
TreeNode deepTree = new TreeNode(1);
TreeNode current = deepTree;
```

```
        for (int i = 0; i < 100; i++) {
            current.left = new TreeNode(i + 2);
            current = current.left;
        }
        System.out.println("101层左链式树:");
        System.out.println("最大深度: " + maxDepth(deepTree));
        System.out.println("最小深度: " + minDepthRecursive(deepTree));
        System.out.println("是否平衡: " + isBalanced(deepTree));
        System.out.println("直径: " + diameterOfBinaryTree(deepTree));

        // 5. 性能测试
        performanceTest();
    }
}

/*
Python实现:

# 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

class BinaryTreeDepth:
    """
    二叉树深度相关算法实现类
    包含最大深度、最小深度、平衡树判断、树直径计算等功能
    """

    def max_depth_recursive(self, root):
        """
        计算二叉树的最大深度 - 递归DFS解法
        时间复杂度: O(N)
        空间复杂度: O(H)
        """
        if not root:
            return 0
        return max(self.max_depth_recursive(root.left), self.max_depth_recursive(root.right)) + 1

    def max_depth_bfs(self, root):
        """
```

```
    计算二叉树的最大深度 - 迭代 BFS 解法
    时间复杂度: O(N)
    空间复杂度: O(W)
    """
    if not root:
        return 0

    from collections import deque
    queue = deque([root])
    depth = 0

    while queue:
        level_size = len(queue)
        for _ in range(level_size):
            node = queue.popleft()
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        depth += 1

    return depth

def max_depth_dfs(self, root):
    """
    计算二叉树的最大深度 - 迭代 DFS 解法
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    if not root:
        return 0

    max_depth = 0
    stack = [(root, 1)]

    while stack:
        node, depth = stack.pop()
        max_depth = max(max_depth, depth)

        if node.right:
            stack.append((node.right, depth + 1))
        if node.left:
            stack.append((node.left, depth + 1))
```

```
    return max_depth

def min_depth_recursive(self, root):
    """
    计算二叉树的最小深度 - 递归DFS解法
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    if not root:
        return 0

    if not root.left and not root.right:
        return 1

    min_depth = float('inf')
    if root.left:
        min_depth = min(min_depth, self.min_depth_recursive(root.left))
    if root.right:
        min_depth = min(min_depth, self.min_depth_recursive(root.right))

    return min_depth + 1

def min_depth_optimized(self, root):
    """
    计算二叉树的最小深度 - 优化递归解法
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    if not root:
        return 0

    # 如果左子树为空，则返回右子树的最小深度+1
    if not root.left:
        return self.min_depth_optimized(root.right) + 1

    # 如果右子树为空，则返回左子树的最小深度+1
    if not root.right:
        return self.min_depth_optimized(root.left) + 1

    # 如果左右子树都不为空，取较小值+1
    return min(self.min_depth_optimized(root.left), self.min_depth_optimized(root.right)) + 1
```

```
def min_depth_bfs(self, root):
    """
    计算二叉树的最小深度 - 迭代BFS解法（最优解）
    时间复杂度: O(N)
    空间复杂度: O(W)
    """
    if not root:
        return 0

    from collections import deque
    queue = deque([root])
    depth = 0

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

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

            # 如果是叶子节点，直接返回当前深度
            if not node.left and not node.right:
                return depth

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    return depth

def min_depth_dfs(self, root):
    """
    计算二叉树的最小深度 - 迭代DFS解法
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    if not root:
        return 0

    min_depth = float('inf')
    stack = [(root, 1)]
```

```
    while stack:
        node, depth = stack.pop()

        # 如果是叶子节点，更新最小深度
        if not node.left and not node.right:
            min_depth = min(min_depth, depth)

        if node.right:
            stack.append((node.right, depth + 1))
        if node.left:
            stack.append((node.left, depth + 1))

    return min_depth

def is_balanced(self, root):
    """
    判断二叉树是否是平衡二叉树
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    def height(node):
        if not node:
            return 0

        left_height = height(node.left)
        if left_height == -1:
            return -1  # 左子树不平衡

        right_height = height(node.right)
        if right_height == -1:
            return -1  # 右子树不平衡

        # 检查当前节点是否平衡
        if abs(left_height - right_height) > 1:
            return -1  # 不平衡

        return max(left_height, right_height) + 1

    return height(root) != -1

def diameter_of_binary_tree(self, root):
    """
    计算二叉树的直径（最长路径长度）
```

```
    时间复杂度: O(N)
    空间复杂度: O(H)
    """
    diameter = [0]  # 使用列表保存直径，便于在递归中修改

    def calculate_diameter(node):
        if not node:
            return 0

        left_height = calculate_diameter(node.left)
        right_height = calculate_diameter(node.right)

        # 更新直径
        diameter[0] = max(diameter[0], left_height + right_height)

        return max(left_height, right_height) + 1

    calculate_diameter(root)
    return diameter[0]

def generate_test_tree(self, tree_type):
    """
    生成测试用例
    """
    if tree_type == 1:  # 完全二叉树
        root = TreeNode(1)
        root.left = TreeNode(2)
        root.right = TreeNode(3)
        root.left.left = TreeNode(4)
        root.left.right = TreeNode(5)
        root.right.left = TreeNode(6)
        root.right.right = TreeNode(7)
        return root
    elif tree_type == 2:  # 左倾斜树
        root = TreeNode(1)
        root.left = TreeNode(2)
        root.left.left = TreeNode(3)
        return root
    elif tree_type == 3:  # 右倾斜树
        root = TreeNode(1)
        root.right = TreeNode(2)
        root.right.right = TreeNode(3)
        return root
```

```
        elif tree_type == 4:  # 单节点树
            return TreeNode(1)
        else:  # 普通二叉树
            root = TreeNode(1)
            root.left = TreeNode(2)
            root.right = TreeNode(3)
            root.left.left = TreeNode(4)
            root.right.right = TreeNode(5)
            root.right.right.right = TreeNode(6)
            return root

# 测试代码
if __name__ == "__main__":
    depth_calculator = BinaryTreeDepth()
    print("二叉树深度相关算法测试")
    print("====================\n")

    # 测试不同类型的树
    for tree_type in range(5):
        print(f"测试树类型 {tree_type}:")
        root = depth_calculator.generate_test_tree(tree_type)

        # 最大深度测试
        max_depth1 = depth_calculator.max_depth_recursive(root)
        max_depth2 = depth_calculator.max_depth_bfs(root)
        max_depth3 = depth_calculator.max_depth_dfs(root)
        print(f"最大深度 (递归 DFS): {max_depth1}")
        print(f"最大深度 (迭代 BFS): {max_depth2}")
        print(f"最大深度 (迭代 DFS): {max_depth3}")

        # 最小深度测试
        min_depth1 = depth_calculator.min_depth_recursive(root)
        min_depth2 = depth_calculator.min_depth_optimized(root)
        min_depth3 = depth_calculator.min_depth_bfs(root)
        min_depth4 = depth_calculator.min_depth_dfs(root)
        print(f"最小深度 (原递归): {min_depth1}")
        print(f"最小深度 (优化递归): {min_depth2}")
        print(f"最小深度 (迭代 BFS): {min_depth3}")
        print(f"最小深度 (迭代 DFS): {min_depth4}")

        # 平衡树测试
        is_balanced = depth_calculator.is_balanced(root)
        print(f"是否是平衡二叉树: {is_balanced}")
```

```
        # 直径测试
        diameter = depth_calculator.diameter_of_binary_tree(root)
        print(f"二叉树的直径: {diameter}")

        print()

    # 边界情况测试
    print("边界情况测试:")

    print(f"空树最大深度: {depth_calculator.max_depth_recursive(None)}")
    print(f"空树最小深度: {depth_calculator.min_depth_optimized(None)}")
    print(f"空树是否平衡: {depth_calculator.is_balanced(None)}")
    print(f"空树直径: {depth_calculator.diameter_of_binary_tree(None)}")
```

C++实现:

```
#include <iostream>
#include <vector>
#include <stack>
#include <queue>
#include <algorithm>
#include <climits>
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 BinaryTreeDepth {
public:
    /**
     * 二叉树深度相关算法实现类
     */

    // 计算二叉树的最大深度 - 递归DFS解法
    int maxDepthRecursive(TreeNode* root) {
```

```
    if (!root) {
        return 0;
    }
    return max(maxDepthRecursive(root->left), maxDepthRecursive(root->right)) + 1;
}

// 计算二叉树的最大深度 - 迭代BFS解法
int maxDepthBFS(TreeNode* root) {
    if (!root) {
        return 0;
    }

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

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

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

    return depth;
}

// 计算二叉树的最大深度 - 迭代DFS解法
int maxDepthDFS(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int maxDepth = 0;
    stack<pair<TreeNode*, int>> s;
```

```
    s.push({root, 1});

    while (!s.empty()) {
        auto [node, depth] = s.top();
        s.pop();

        maxDepth = max(maxDepth, depth);

        if (node->right) {
            s.push({node->right, depth + 1});
        }
        if (node->left) {
            s.push({node->left, depth + 1});
        }
    }

    return maxDepth;
}

// 计算二叉树的最小深度 - 递归DFS解法
int minDepthRecursive(TreeNode* root) {
    if (!root) {
        return 0;
    }

    if (!root->left && !root->right) {
        return 1;
    }

    int minDepth = INT_MAX;
    if (root->left) {
        minDepth = min(minDepth, minDepthRecursive(root->left));
    }
    if (root->right) {
        minDepth = min(minDepth, minDepthRecursive(root->right));
    }

    return minDepth + 1;
}

// 计算二叉树的最小深度 - 优化递归解法
int minDepthOptimized(TreeNode* root) {
    if (!root) {
```

```
        return 0;
    }

    // 如果左子树为空，则返回右子树的最小深度+1
    if (!root->left) {
        return minDepthOptimized(root->right) + 1;
    }

    // 如果右子树为空，则返回左子树的最小深度+1
    if (!root->right) {
        return minDepthOptimized(root->left) + 1;
    }

    // 如果左右子树都不为空，取较小值+1
    return min(minDepthOptimized(root->left), minDepthOptimized(root->right)) + 1;
}

// 计算二叉树的最小深度 - 迭代BFS解法（最优解）
int minDepthBFS(TreeNode* root) {
    if (!root) {
        return 0;
    }

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

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

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

            // 如果是叶子节点，直接返回当前深度
            if (!node->left && !node->right) {
                return depth;
            }

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

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

    return depth;
}

// 计算二叉树的最小深度 - 迭代DFS解法
int minDepthDFS(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int minDepth = INT_MAX;
    stack<pair<TreeNode*, int>> s;
    s.push({root, 1});

    while (!s.empty()) {
        auto [node, depth] = s.top();
        s.pop();

        // 如果是叶子节点，更新最小深度
        if (!node->left && !node->right) {
            minDepth = min(minDepth, depth);
        }

        if (node->right) {
            s.push({node->right, depth + 1});
        }
        if (node->left) {
            s.push({node->left, depth + 1});
        }
    }

    return minDepth;
}

// 判断二叉树是否是平衡二叉树
bool isBalanced(TreeNode* root) {
    return height(root) != -1;
}
```

```
// 辅助方法：计算树的高度，如果不是平衡树则返回-1
int height(TreeNode* root) {
    if (!root) {
        return 0;
    }

    int leftHeight = height(root->left);
    if (leftHeight == -1) {
        return -1; // 左子树不平衡
    }

    int rightHeight = height(root->right);
    if (rightHeight == -1) {
        return -1; // 右子树不平衡
    }

    // 检查当前节点是否平衡
    if (abs(leftHeight - rightHeight) > 1) {
        return -1; // 不平衡
    }

    return max(leftHeight, rightHeight) + 1;
}

// 计算二叉树的直径（最长路径长度）
int diameterOfBinaryTree(TreeNode* root) {
    int diameter = 0;
    calculateDiameter(root, diameter);
    return diameter;
}

// 辅助方法：计算树的高度并更新直径
int calculateDiameter(TreeNode* root, int& diameter) {
    if (!root) {
        return 0;
    }

    int leftHeight = calculateDiameter(root->left, diameter);
    int rightHeight = calculateDiameter(root->right, diameter);

    // 更新直径
    diameter = max(diameter, leftHeight + rightHeight);
```

```
    return max(leftHeight, rightHeight) + 1;
}

// 生成测试用例
TreeNode* generateTestTree(int treeType) {
    if (treeType == 1) {  // 完全二叉树
        TreeNode* root = new TreeNode(1);
        root->left = new TreeNode(2);
        root->right = new TreeNode(3);
        root->left->left = new TreeNode(4);
        root->left->right = new TreeNode(5);
        root->right->left = new TreeNode(6);
        root->right->right = new TreeNode(7);
        return root;
    } else if (treeType == 2) {  // 左倾斜树
        TreeNode* root = new TreeNode(1);
        root->left = new TreeNode(2);
        root->left->left = new TreeNode(3);
        return root;
    } else if (treeType == 3) {  // 右倾斜树
        TreeNode* root = new TreeNode(1);
        root->right = new TreeNode(2);
        root->right->right = new TreeNode(3);
        return root;
    } else if (treeType == 4) {  // 单节点树
        return new TreeNode(1);
    } else {  // 普通二叉树
        TreeNode* root = new TreeNode(1);
        root->left = new TreeNode(2);
        root->right = new TreeNode(3);
        root->left->left = new TreeNode(4);
        root->right->right = new TreeNode(5);
        root->right->right->right = new TreeNode(6);
        return root;
    }
}

// 释放树内存
void deleteTree(TreeNode* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
```

```
            delete root;
        }
    }
};

// 测试代码
int main() {
    BinaryTreeDepth depthCalculator;
    cout << "二叉树深度相关算法测试" << endl;
    cout << "====================\n" << endl;

    // 测试不同类型的树
    for (int treeType = 0; treeType < 5; ++treeType) {
        cout << "测试树类型 " << treeType << ":" << endl;
        TreeNode* root = depthCalculator.generateTestTree(treeType);

        // 最大深度测试
        int maxDepth1 = depthCalculator.maxDepthRecursive(root);
        int maxDepth2 = depthCalculator.maxDepthBFS(root);
        int maxDepth3 = depthCalculator.maxDepthDFS(root);
        cout << "最大深度 (递归 DFS): " << maxDepth1 << endl;
        cout << "最大深度 (迭代 BFS): " << maxDepth2 << endl;
        cout << "最大深度 (迭代 DFS): " << maxDepth3 << endl;

        // 最小深度测试
        int minDepth1 = depthCalculator.minDepthRecursive(root);
        int minDepth2 = depthCalculator.minDepthOptimized(root);
        int minDepth3 = depthCalculator.minDepthBFS(root);
        int minDepth4 = depthCalculator.minDepthDFS(root);
        cout << "最小深度 (原递归): " << minDepth1 << endl;
        cout << "最小深度 (优化递归): " << minDepth2 << endl;
        cout << "最小深度 (迭代 BFS): " << minDepth3 << endl;
        cout << "最小深度 (迭代 DFS): " << minDepth4 << endl;

        // 平衡树测试
        bool isBalanced = depthCalculator.isBalanced(root);
        cout << "是否是平衡二叉树: " << (isBalanced ? "true" : "false") << endl;

        // 直径测试
        int diameter = depthCalculator.diameterOfBinaryTree(root);
        cout << "二叉树的直径: " << diameter << endl;

        cout << endl;
```

```
        // 释放内存
        depthCalculator.deleteTree(root);
    }

    // 边界情况测试
    cout << "边界情况测试:" << endl;

    cout << "空树最大深度: " << depthCalculator.maxDepthRecursive(nullptr) << endl;
    cout << "空树最小深度: " << depthCalculator.minDepthOptimized(nullptr) << endl;
    cout << "空树是否平衡: " << (depthCalculator.isBalanced(nullptr) ? "true" : "false") << endl;
    cout << "空树直径: " << depthCalculator.diameterOfBinaryTree(nullptr) << endl;

    return 0;
}
*/

==========================================

文件: Code05_PreorderSerializeAndDeserialize.java
==========================================
package class036;

import java.util.*;

/**
 * 二叉树序列化与反序列化算法实现
 * 测试链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
 *
 * 核心算法思想:
 * 1. 序列化：将二叉树结构转换为字符串表示形式
 * 2. 反序列化：将字符串表示形式恢复为原始二叉树结构
 *
 * 实现方式:
 * - 先序遍历序列化（递归实现）
 * - BFS 层序遍历序列化（迭代实现）
 * - 后序遍历序列化（递归实现）
 *
 * 关键优化点:
 * - 使用字符串构建器提升性能，避免频繁字符串拼接
 * - 使用特殊标记（如"#"）表示空节点
 * - 使用分隔符（如","）分离节点值
 * - 对于 BFS 实现，使用队列高效管理节点访问顺序
```

```
 *
 * 边界情况处理：
 * - 空树处理：返回特殊标记
 * - 单节点树处理：直接返回节点值加空标记
 * - 数据类型转换异常处理
 *
 * 工程化考量：
 * - 代码可读性：方法职责单一，命名清晰
 * - 可扩展性：支持多种序列化方式
 * - 健壮性：包含异常处理机制
 * - 性能优化：使用 StringBuilder、避免递归栈溢出
 *
 * 相关题目：
 * 1. LeetCode 297. 二叉树的序列化与反序列化 - https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
 * 2. LeetCode 449. 序列化和反序列化二叉搜索树 - https://leetcode.cn/problems/serialize-and-deserialize-bst/
 * 3. LeetCode 536. 从字符串生成二叉树 - https://leetcode.cn/problems/construct-binary-tree-from-string/
 * 4. LeetCode 652. 寻找重复的子树 - https://leetcode.cn/problems/find-duplicate-subtrees/
 * 5. LeetCode 1008. 前序遍历构造二叉搜索树 - https://leetcode.cn/problems/construct-binary-search-tree-from-preorder-traversal/
 * 6. LeetCode 889. 根据前序和后序遍历构造二叉树 - https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-postorder-traversal/
 * 7. LeetCode 105. 从前序与中序遍历序列构造二叉树 - https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
 * 8. LeetCode 106. 从中序与后序遍历序列构造二叉树 - https://leetcode.cn/problems/construct-binary-tree-from-inorder-and-postorder-traversal/
 * 9. LeetCode 1028. 从先序遍历还原二叉树 - https://leetcode.cn/problems/recover-a-tree-from-preorder-traversal/
 * 10. LeetCode 1302. 层数最深叶子节点的和 - https://leetcode.cn/problems/deepest-leaves-sum/
 * 11. LeetCode 116. 填充每个节点的下一个右侧节点指针 - https://leetcode.cn/problems/populating-next-right-pointers-in-each-node/
 * 12. LeetCode 117. 填充每个节点的下一个右侧节点指针 II - https://leetcode.cn/problems/populating-next-right-pointers-in-each-node-ii/
 * 13. LeetCode 104. 二叉树的最大深度 - https://leetcode.cn/problems/maximum-depth-of-binary-tree/
 */
public class Code05_PreorderSerializeAndDeserialize {

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

```
    public TreeNode left;
    public TreeNode right;

    public TreeNode(int v) {
        val = v;
    }

    @Override
    public String toString() {
        return String.valueOf(val);
    }
}

/**
 * 重要说明：
 * 二叉树可以通过先序、后序或者按层遍历的方式序列化和反序列化
 * 但是，二叉树无法通过中序遍历的方式实现序列化和反序列化
 * 因为不同的两棵树，可能得到同样的中序序列，即便补了空位置也可能一样。
 * 比如如下两棵树
 *         __2
 *        /
 *       1
 *       和
 *       1__
 *          \
 *           2
 * 补足空位置的中序遍历结果都是{ null, 1, null, 2, null}
 */

// 提交这个类 - 先序遍历实现
public class CodecPreorder {

    /**
     * 将二叉树序列化为字符串
     * 实现思路：使用先序遍历（根-左-右），递归访问每个节点
     * 时间复杂度：O(n)，其中 n 是节点数
     * 空间复杂度：O(h)，h 是树的高度，最坏情况下为 O(n)
     * @param root 二叉树根节点
     * @return 序列化后的字符串
     */
    public String serialize(TreeNode root) {
        StringBuilder builder = new StringBuilder();
        preorderSerialize(root, builder);
```

```
    return builder.toString();
}

/**
 * 先序遍历序列化辅助方法
 * @param root 当前节点
 * @param builder 字符串构建器
 */
void preorderSerialize(TreeNode root, StringBuilder builder) {
    if (root == null) {
        builder.append("#,");
    } else {
        builder.append(root.val + ",");
        preorderSerialize(root.left, builder);
        preorderSerialize(root.right, builder);
    }
}

/**
 * 将字符串反序列化为二叉树
 * 实现思路：根据先序遍历的特点，递归重建二叉树
 * 时间复杂度：O(n)
 * 空间复杂度：O(h)，h 是树的高度
 * @param data 序列化的字符串
 * @return 重建后的二叉树根节点
 */
public TreeNode deserialize(String data) {
    String[] vals = data.split(",");
    cnt = 0;
    return preorderDeserialize(vals);
}

// 当前数组消费到哪了
public static int cnt;

/**
 * 先序遍历反序列化辅助方法
 * @param vals 节点值数组
 * @return 重建的节点
 */
TreeNode preorderDeserialize(String[] vals) {
    String cur = vals[cnt++];
    if (cur.equals("#")) {
```

```
            return null;
        } else {
            TreeNode head = new TreeNode(Integer.valueOf(cur));
            head.left = preorderDeserialize(vals);
            head.right = preorderDeserialize(vals);
            return head;
        }
    }
}

// BFS 层序遍历实现
public class CodecBFS {

    /**
     * 使用 BFS 层序遍历序列化二叉树
     * 实现思路：使用队列进行层次遍历，记录每个节点的值或空标记
     * 时间复杂度：O(n)
     * 空间复杂度：O(w)，w 是树的最大宽度
     * @param root 二叉树根节点
     * @return 序列化后的字符串
     */
    public String serialize(TreeNode root) {
        if (root == null) {
            return "#";
        }
        StringBuilder builder = new StringBuilder();
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (node == null) {
                builder.append("#,");
            } else {
                builder.append(node.val + ",");
                queue.offer(node.left);
                queue.offer(node.right);
            }
        }

        // 移除末尾的逗号
        if (builder.length() > 0) {
            builder.setLength(builder.length() - 1);
```

```
    }

    return builder.toString();
}

/**
 * 使用 BFS 层序遍历反序列化二叉树
 * 实现思路：使用队列重建树结构，按层次构建每个节点的左右子节点
 * 时间复杂度：O(n)
 * 空间复杂度：O(w)，w 是树的最大宽度
 * @param data 序列化的字符串
 * @return 重建后的二叉树根节点
 */
public TreeNode deserialize(String data) {
    if (data.equals("#")) {
        return null;
    }

    String[] vals = data.split(",");
    TreeNode root = new TreeNode(Integer.parseInt(vals[0]));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int index = 1;

    while (!queue.isEmpty() && index < vals.length) {
        TreeNode node = queue.poll();

        // 处理左子节点
        if (!vals[index].equals("#")) {
            node.left = new TreeNode(Integer.parseInt(vals[index]));
            queue.offer(node.left);
        }
        index++;

        // 处理右子节点
        if (index < vals.length && !vals[index].equals("#")) {
            node.right = new TreeNode(Integer.parseInt(vals[index]));
            queue.offer(node.right);
        }
        index++;
    }

    return root;
```

```
    }
}

// 后序遍历实现
public class CodecPostorder {

    /**
     * 使用后序遍历序列化二叉树
     * 实现思路：递归后序遍历（左-右-根），记录每个节点的值或空标记
     * 时间复杂度：O(n)
     * 空间复杂度：O(h)，h 是树的高度
     * @param root 二叉树根节点
     * @return 序列化后的字符串
     */
    public String serialize(TreeNode root) {
        StringBuilder builder = new StringBuilder();
        postorderSerialize(root, builder);
        // 移除末尾的逗号
        if (builder.length() > 0) {
            builder.setLength(builder.length() - 1);
        }
        return builder.toString();
    }

    private void postorderSerialize(TreeNode root, StringBuilder builder) {
        if (root == null) {
            builder.append("#,");
        } else {
            postorderSerialize(root.left, builder);
            postorderSerialize(root.right, builder);
            builder.append(root.val + ",");
        }
    }

    /**
     * 使用后序遍历反序列化二叉树
     * 实现思路：将数组反向处理，使用栈模拟递归过程，按照根-右-左的顺序重建
     * 时间复杂度：O(n)
     * 空间复杂度：O(h)，h 是树的高度
     * @param data 序列化的字符串
     * @return 重建后的二叉树根节点
     */
    public TreeNode deserialize(String data) {
```

```
        if (data.equals("#")) {
            return null;
        }

        String[] vals = data.split(",");
        Stack<String> stack = new Stack<>();
        for (String val : vals) {
            stack.push(val);
        }

        return postorderDeserialize(stack);
    }

    private TreeNode postorderDeserialize(Stack<String> stack) {
        String val = stack.pop();
        if (val.equals("#")) {
            return null;
        }

        TreeNode root = new TreeNode(Integer.parseInt(val));
        // 注意：后序遍历反序列化时，先处理右子树再处理左子树
        root.right = postorderDeserialize(stack);
        root.left = postorderDeserialize(stack);

        return root;
    }
}

/**
 * 打印二叉树结构的辅助方法
 * 使用层序遍历方式，打印树的结构
 * @param root 二叉树根节点
 */
public static void printTree(TreeNode root) {
    if (root == null) {
        System.out.println("[空树]");
        return;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    List<List<String>> levels = new ArrayList<>();
```

```
while (!queue.isEmpty()) {
    int levelSize = queue.size();
    List<String> level = new ArrayList<>();
    boolean hasNonNull = false;

    for (int i = 0; i < levelSize; i++) {
        TreeNode node = queue.poll();
        if (node != null) {
            level.add(String.valueOf(node.val));
            queue.offer(node.left);
            queue.offer(node.right);
            if (node.left != null || node.right != null) {
                hasNonNull = true;
            }
        } else {
            level.add("null");
            queue.offer(null);
            queue.offer(null);
        }
    }

    levels.add(level);
    // 如果当前层之后都是 null，停止遍历
    if (!hasNonNull) {
        break;
    }
}

// 打印树结构
int maxWidth = levels.get(levels.size() - 2).size() * 4 - 3;
for (int i = 0; i < levels.size() - 1; i++) { // 去掉最后一层全为 null 的层
    int levelWidth = levels.get(i).size();
    int space = (maxWidth - (levelWidth * 4 - 3)) / 2;

    StringBuilder line = new StringBuilder();
    for (int j = 0; j < space; j++) {
        line.append(" ");
    }

    for (int j = 0; j < levelWidth; j++) {
        String val = levels.get(i).get(j);
        line.append(val);
        if (j < levelWidth - 1) {
```

```
                    for (int k = 0; k < 4 - val.length(); k++) {
                        line.append(" ");
                    }
                }
            }

            System.out.println(line.toString());
        }
    }

    /**
     * 性能测试方法
     * 测试不同序列化方法在深层树情况下的性能
     */
    public static void performanceTest() {
        System.out.println("\n===== 性能测试 =====");

        // 创建深度为1000的左链式树
        TreeNode deepTree = createDeepTree(1000);

        Code05_PreorderSerializeAndDeserialize outer = new
Code05_PreorderSerializeAndDeserialize();
        CodecPreorder codecPreorder = outer.new CodecPreorder();
        CodecBFS codecBFS = outer.new CodecBFS();
        CodecPostorder codecPostorder = outer.new CodecPostorder();

        // 测试先序序列化性能
        long start = System.currentTimeMillis();
        String preorderSerialized = codecPreorder.serialize(deepTree);
        long end = System.currentTimeMillis();
        System.out.println("先序序列化耗时: " + (end - start) + "ms");

        // 测试BFS序列化性能
        start = System.currentTimeMillis();
        String bfsSerialized = codecBFS.serialize(deepTree);
        end = System.currentTimeMillis();
        System.out.println("BFS序列化耗时: " + (end - start) + "ms");

        // 测试后序序列化性能
        start = System.currentTimeMillis();
        String postorderSerialized = codecPostorder.serialize(deepTree);
        end = System.currentTimeMillis();
        System.out.println("后序序列化耗时: " + (end - start) + "ms");
```

```
        // 测试先序反序列化性能
        start = System.currentTimeMillis();
        TreeNode preorderDeserialized = codecPreorder.deserialize(preorderSerialized);
        end = System.currentTimeMillis();
        System.out.println("先序反序列化耗时: " + (end - start) + "ms");

        // 测试 BFS 反序列化性能
        start = System.currentTimeMillis();
        TreeNode bfsDeserialized = codecBFS.deserialize(bfsSerialized);
        end = System.currentTimeMillis();
        System.out.println("BFS 反序列化耗时: " + (end - start) + "ms");

        // 测试后序反序列化性能
        start = System.currentTimeMillis();
        TreeNode postorderDeserialized = codecPostorder.deserialize(postorderSerialized);
        end = System.currentTimeMillis();
        System.out.println("后序反序列化耗时: " + (end - start) + "ms");
    }

    /**
     * 创建指定深度的左链式树
     * @param depth 树的深度
     * @return 创建的树的根节点
     */
    private static TreeNode createDeepTree(int depth) {
        TreeNode root = new TreeNode(1);
        TreeNode current = root;
        for (int i = 2; i <= depth; i++) {
            current.left = new TreeNode(i);
            current = current.left;
        }
        return root;
    }

    /**
     * 创建完全二叉树
     * @param height 树的高度
     * @return 创建的完全二叉树的根节点
     */
    private static TreeNode createCompleteBinaryTree(int height) {
        if (height <= 0) {
            return null;
```

```
    }
    return createCompleteBinaryTreeHelper(1, height);
}

private static TreeNode createCompleteBinaryTreeHelper(int val, int height) {
    if (val > Math.pow(2, height) - 1) {
        return null;
    }
    TreeNode root = new TreeNode(val);
    root.left = createCompleteBinaryTreeHelper(2 * val, height);
    root.right = createCompleteBinaryTreeHelper(2 * val + 1, height);
    return root;
}

/**
 * 生成各种测试树
 * @param type 树类型：1-完全二叉树，2-左倾斜树，3-右倾斜树，4-单节点树，5-普通二叉树
 * @return 生成的测试树
 */
public static TreeNode generateTestTree(int type) {
    switch (type) {
        case 1: // 完全二叉树
            return createCompleteBinaryTree(4);
        case 2: // 左倾斜树
            TreeNode leftSkewed = new TreeNode(1);
            TreeNode current = leftSkewed;
            for (int i = 2; i <= 5; i++) {
                current.left = new TreeNode(i);
                current = current.left;
            }
            return leftSkewed;
        case 3: // 右倾斜树
            TreeNode rightSkewed = new TreeNode(1);
            current = rightSkewed;
            for (int i = 2; i <= 5; i++) {
                current.right = new TreeNode(i);
                current = current.right;
            }
            return rightSkewed;
        case 4: // 单节点树
            return new TreeNode(1);
        case 5: // 普通二叉树
            TreeNode root = new TreeNode(1);
```

```
                root.left = new TreeNode(2);
                root.right = new TreeNode(3);
                root.left.left = new TreeNode(4);
                root.left.right = new TreeNode(5);
                root.right.right = new TreeNode(6);
                root.left.left.left = new TreeNode(7);
                return root;
            default:
                return null;
        }
    }

    /**
     * 比较两个二叉树是否完全相同
     */
    public static boolean isSameTree(TreeNode p, TreeNode q) {
        if (p == null && q == null) {
            return true;
        }
        if (p == null || q == null) {
            return false;
        }
        return p.val == q.val && isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }

    public static void main(String[] args) {
        Code05_PreorderSerializeAndDeserialize outer = new
Code05_PreorderSerializeAndDeserialize();
        CodecPreorder codecPreorder = outer.new CodecPreorder();
        CodecBFS codecBFS = outer.new CodecBFS();
        CodecPostorder codecPostorder = outer.new CodecPostorder();

        // 1. 标准测试用例
        System.out.println("===== 标准测试用例 =====");
        for (int i = 1; i <= 5; i++) {
            System.out.println("\n 测试树类型 " + i + ":");
            TreeNode tree = generateTestTree(i);
            System.out.println("原始树结构:");
            printTree(tree);

            // 先序序列化测试
            String preorderSerialized = codecPreorder.serialize(tree);
            System.out.println("先序序列化结果: " + preorderSerialized);
```

```
            TreeNode preorderDeserialized = codecPreorder.deserialize(preorderSerialized);
            System.out.println("先序反序列化后树结构:");
            printTree(preorderDeserialized);
            System.out.println("先序序列化/反序列化正确性: " + isSameTree(tree,
preorderDeserialized));

            // BFS 序列化测试
            String bfsSerialized = codecBFS.serialize(tree);
            System.out.println("BFS 序列化结果: " + bfsSerialized);
            TreeNode bfsDeserialized = codecBFS.deserialize(bfsSerialized);
            System.out.println("BFS 反序列化后树结构:");
            printTree(bfsDeserialized);
            System.out.println("BFS 序列化/反序列化正确性: " + isSameTree(tree, bfsDeserialized));

            // 后序序列化测试
            String postorderSerialized = codecPostorder.serialize(tree);
            System.out.println("后序序列化结果: " + postorderSerialized);
            TreeNode postorderDeserialized = codecPostorder.deserialize(postorderSerialized);
            System.out.println("后序反序列化后树结构:");
            printTree(postorderDeserialized);
            System.out.println("后序序列化/反序列化正确性: " + isSameTree(tree,
postorderDeserialized));
        }

        // 2. 边界情况测试
        System.out.println("\n===== 边界情况测试 =====");
        // 空树测试
        TreeNode nullTree = null;
        System.out.println("空树测试:");
        String preorderNull = codecPreorder.serialize(nullTree);
        System.out.println("先序序列化空树: " + preorderNull);
        TreeNode deserializedNull = codecPreorder.deserialize(preorderNull);
        System.out.println("先序反序列化空树结果: " + (deserializedNull == null ? "null" :
deserializedNull.val));

        // 3. 深层树测试
        System.out.println("\n===== 深层树测试 =====");
        TreeNode deepTree = createDeepTree(10);
        System.out.println("深层树(深度 10):");
        String deepSerialized = codecPreorder.serialize(deepTree);
        TreeNode deepDeserialized = codecPreorder.deserialize(deepSerialized);
        System.out.println("深层树序列化/反序列化正确性: " + isSameTree(deepTree,
deepDeserialized));
```

```
    // 4. 性能测试
    performanceTest();

    // 5. 不同序列化方法结果比较
    System.out.println("\n===== 不同序列化方法结果比较 =====");
    TreeNode sampleTree = generateTestTree(5);
    String preorder = codecPreorder.serialize(sampleTree);
    String bfs = codecBFS.serialize(sampleTree);
    String postorder = codecPostorder.serialize(sampleTree);
    System.out.println("先序序列化: " + preorder);
    System.out.println("BFS 序列化: " + bfs);
    System.out.println("后序序列化: " + postorder);
}

/*
// Python 实现
# 二叉树序列化与反序列化的 Python 实现
# 包含先序、BFS 和后序三种实现方式

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

    def __str__(self):
        return str(self.val)

class CodecPreorder:
    def serialize(self, root):
        """将二叉树序列化为字符串（先序遍历）"""
        result = []
        self._preorder_serialize(root, result)
        return ",".join(result)

    def _preorder_serialize(self, root, result):
        if root is None:
            result.append("#")
        else:
            result.append(str(root.val))
            self._preorder_serialize(root.left, result)
            self._preorder_serialize(root.right, result)
```

```
    def deserialize(self, data):
        """将字符串反序列化为二叉树（先序遍历）"""
        if data == "":
            return None
        values = data.split(",")
        self.index = 0
        return self._preorder_deserialize(values)

    def _preorder_deserialize(self, values):
        if self.index >= len(values) or values[self.index] == "#":
            self.index += 1
            return None

        root = TreeNode(int(values[self.index]))
        self.index += 1
        root.left = self._preorder_deserialize(values)
        root.right = self._preorder_deserialize(values)
        return root

class CodecBFS:
    def serialize(self, root):
        """使用 BFS 层序遍历序列化二叉树"""
        if not root:
            return "#"

        result = []
        queue = [root]

        while queue:
            node = queue.pop(0)
            if node is None:
                result.append("#")
            else:
                result.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)

        return ",".join(result)

    def deserialize(self, data):
        """使用 BFS 层序遍历反序列化二叉树"""
        if data == "#":
```

```
            return None

        values = data.split(",")
        root = TreeNode(int(values[0]))
        queue = [root]
        index = 1

        while queue and index < len(values):
            node = queue.pop(0)

            # 处理左子节点
            if index < len(values) and values[index] != "#":
                node.left = TreeNode(int(values[index]))
                queue.append(node.left)
            index += 1

            # 处理右子节点
            if index < len(values) and values[index] != "#":
                node.right = TreeNode(int(values[index]))
                queue.append(node.right)
            index += 1

        return root

class CodecPostorder:
    def serialize(self, root):
        """使用后序遍历序列化二叉树"""
        result = []
        self._postorder_serialize(root, result)
        return ",".join(result)

    def _postorder_serialize(self, root, result):
        if root is None:
            result.append("#")
        else:
            self._postorder_serialize(root.left, result)
            self._postorder_serialize(root.right, result)
            result.append(str(root.val))

    def deserialize(self, data):
        """使用后序遍历反序列化二叉树"""
        if data == "":
            return None
```

```
        values = data.split(",")
        self.stack = values[::-1]  # 反转数组，用于模拟栈操作
        return self._postorder_deserialize()

    def _postorder_deserialize(self):
        if not self.stack:
            return None

        val = self.stack.pop(0)
        if val == "#":
            return None

        root = TreeNode(int(val))
        # 注意：后序遍历反序列化时，先处理右子树再处理左子树
        root.right = self._postorder_deserialize()
        root.left = self._postorder_deserialize()

        return root
*/

/*
// C++实现
// 二叉树序列化与反序列化的C++实现
// 包含先序、BFS和后序三种实现方式

#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <sstream>

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

class CodecPreorder {
public:
    // 先序遍历序列化
```

```
    std::string serialize(TreeNode* root) {
        std::string result;
        preorderSerialize(root, result);
        return result;
    }

    // 先序遍历反序列化
    TreeNode* deserialize(std::string data) {
        std::vector<std::string> values;
        split(data, values, ',');
        int index = 0;
        return preorderDeserialize(values, index);
    }

private:
    void preorderSerialize(TreeNode* root, std::string& result) {
        if (root == nullptr) {
            result += "#,";
        } else {
            result += std::to_string(root->val) + ",";
            preorderSerialize(root->left, result);
            preorderSerialize(root->right, result);
        }
    }

    TreeNode* preorderDeserialize(const std::vector<std::string>& values, int& index) {
        if (index >= values.size() || values[index] == "#") {
            index++;
            return nullptr;
        }

        TreeNode* root = new TreeNode(std::stoi(values[index]));
        index++;
        root->left = preorderDeserialize(values, index);
        root->right = preorderDeserialize(values, index);
        return root;
    }

    void split(const std::string& s, std::vector<std::string>& tokens, char delimiter) {
        std::string token;
        std::istringstream tokenStream(s);
        while (std::getline(tokenStream, token, delimiter)) {
            tokens.push_back(token);
```

```
        }
    }
};

class CodecBFS {
public:
    // BFS层序遍历序列化
    std::string serialize(TreeNode* root) {
        if (!root) return "#";

        std::string result;
        std::queue<TreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            TreeNode* node = q.front();
            q.pop();

            if (!node) {
                result += "#,";
            } else {
                result += std::to_string(node->val) + ",";
                q.push(node->left);
                q.push(node->right);
            }
        }

        // 移除最后一个逗号
        if (!result.empty()) {
            result.pop_back();
        }
        return result;
    }

    // BFS层序遍历反序列化
    TreeNode* deserialize(std::string data) {
        if (data == "#") return nullptr;

        std::vector<std::string> values;
        split(data, values, ',');

        TreeNode* root = new TreeNode(std::stoi(values[0]));
        std::queue<TreeNode*> q;
```

```
        q.push(root);
        int index = 1;

        while (!q.empty() && index < values.size()) {
            TreeNode* node = q.front();
            q.pop();

            // 处理左子节点
            if (index < values.size() && values[index] != "#") {
                node->left = new TreeNode(std::stoi(values[index]));
                q.push(node->left);
            }
            index++;

            // 处理右子节点
            if (index < values.size() && values[index] != "#") {
                node->right = new TreeNode(std::stoi(values[index]));
                q.push(node->right);
            }
            index++;
        }

        return root;
    }

private:
    void split(const std::string& s, std::vector<std::string>& tokens, char delimiter) {
        std::string token;
        std::istringstream tokenStream(s);
        while (std::getline(tokenStream, token, delimiter)) {
            tokens.push_back(token);
        }
    }
};

class CodecPostorder {
public:
    // 后序遍历序列化
    std::string serialize(TreeNode* root) {
        std::string result;
        postorderSerialize(root, result);
        // 移除最后一个逗号
        if (!result.empty()) {
```

```
            result.pop_back();
        }
        return result;
    }

    // 后序遍历反序列化
    TreeNode* deserialize(std::string data) {
        if (data.empty()) return nullptr;

        std::vector<std::string> values;
        split(data, values, ',');

        std::stack<std::string> stack;
        for (const auto& val : values) {
            stack.push(val);
        }

        return postorderDeserialize(stack);
    }

private:
    void postorderSerialize(TreeNode* root, std::string& result) {
        if (root == nullptr) {
            result += "#,";
        } else {
            postorderSerialize(root->left, result);
            postorderSerialize(root->right, result);
            result += std::to_string(root->val) + ",";
        }
    }

    TreeNode* postorderDeserialize(std::stack<std::string>& stack) {
        if (stack.empty()) return nullptr;

        std::string val = stack.top();
        stack.pop();

        if (val == "#") {
            return nullptr;
        }

        TreeNode* root = new TreeNode(std::stoi(val));
        // 注意：后序遍历反序列化时，先处理右子树再处理左子树
```

```
        root->right = postorderDeserialize(stack);
        root->left = postorderDeserialize(stack);

        return root;
    }

    void split(const std::string& s, std::vector<std::string>& tokens, char delimiter) {
        std::string token;
        std::istringstream tokenStream(s);
        while (std::getline(tokenStream, token, delimiter)) {
            tokens.push_back(token);
        }
    }
};

// 辅助函数：释放树的内存
void deleteTree(TreeNode* root) {
    if (root) {
        deleteTree(root->left);
        deleteTree(root->right);
        delete root;
    }
}

// 辅助函数：打印树结构（简化版）
void printTree(TreeNode* root) {
    if (!root) {
        std::cout << "[空树]" << std::endl;
        return;
    }

    std::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();

            if (node) {
                std::cout << node->val << " ";
                q.push(node->left);
```

```
                    q.push(node->right);
                } else {
                    std::cout << "# ";
                }
            }
            std::cout << std::endl;
        }
    }
    */
}
```

==========

文件: Code06_LevelorderSerializeAndDeserialize.java

==========

```
package class036;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;
import java.util.StringJoiner;

/**
 * 二叉树的层序序列化与反序列化
 * 测试链接 : https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
 *
 * 二叉树序列化是将二叉树结构转换为可存储或传输的字符串格式的过程，
 * 反序列化则是将该字符串恢复为原二叉树结构的过程。
 *
 * 层序序列化的核心思想：
 * 1. 按层遍历二叉树，将每个节点的值转换为字符串
 * 2. 对于空节点，使用特殊标记（如"#"）表示
 * 3. 用分隔符（如","）连接所有节点值
 *
 * 时间复杂度：O(N)，其中 N 是二叉树中的节点数
 * 空间复杂度：O(N)，需要一个队列存储节点，以及一个字符串存储序列化结果
 *
 * 相关题目：
 * 1. LeetCode 297. 二叉树的序列化与反序列化 (本文件)
 * 2. LintCode 7. 二叉树序列化与反序列化
 * 3. HackerRank Tree: Serialize and Deserialize
 * 4. CodeChef Serialize and Deserialize a Binary Tree
 * 5. UVA 10562 - Undraw the Trees
```

```
 * 6. Codeforces 862B - Mahmoud and Ehab and the bipartiteness
 * 7. AtCoder AHC001 - Let's Play Tag
 */
public class Code06_LevelorderSerializeAndDeserialize {

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

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

        public TreeNode() {}

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

    /**
     * 方法1：序列化，使用层序遍历的方式
     * 时间复杂度：O(N)
     * 空间复杂度：O(N)
     * @param root 二叉树的根节点
     * @return 序列化后的字符串
     */
    public static String serialize(TreeNode root) {
        StringBuilder builder = new StringBuilder();
        if (root == null) {
            builder.append("#");
        } else {
            // 层序遍历，用队列实现
            Queue<TreeNode> queue = new LinkedList<>();
            queue.add(root);
            // 先放入头节点的值
            builder.append(root.val);
            while (!queue.isEmpty()) {
                TreeNode node = queue.poll();
```

```
            // 左孩子
            if (node.left != null) {
                builder.append("," + node.left.val);
                queue.add(node.left);
            } else {
                builder.append(",#");
            }
            // 右孩子
            if (node.right != null) {
                builder.append("," + node.right.val);
                queue.add(node.right);
            } else {
                builder.append(",#");
            }
        }
    }
    return builder.toString();
}

/**
 * 方法 2：序列化优化版本，使用 StringJoiner 提高可读性
 * 时间复杂度：O(N)
 * 空间复杂度：O(N)
 * 优点：代码更清晰，使用 StringJoiner 处理分隔符
 */
public static String serializeOptimized(TreeNode root) {
    if (root == null) {
        return "#";
    }

    StringJoiner joiner = new StringJoiner(",");
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

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

        if (node == null) {
            joiner.add("#");
        } else {
            joiner.add(String.valueOf(node.val));
            // 无论子节点是否为空，都入队，因为后续需要序列化空节点
            queue.offer(node.left);
```

```
            queue.offer(node.right);
        }
    }

    // 移除末尾的连续#以节省空间（可选优化）
    String result = joiner.toString();
    int lastNonHash = result.lastIndexOf('#');
    while (lastNonHash >= 0 && result.charAt(lastNonHash) == '#') {
        lastNonHash--;
    }
    return result.substring(0, lastNonHash + 1);
}

/**
 * 反序列化，根据层序遍历的字符串，重建二叉树
 * 时间复杂度：O(N)
 * 空间复杂度：O(N)
 * @param data 序列化的字符串
 * @return 重建的二叉树根节点
 */
public static TreeNode deserialize(String data) {
    if (data == null || data.equals("#")) {
        return null;
    }
    String[] values = data.split(",");
    // 头节点
    TreeNode root = new TreeNode(Integer.parseInt(values[0]));
    // 队列
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    // 从第二个位置开始遍历，因为0位置是头节点
    int index = 1;
    while (!queue.isEmpty() && index < values.length) {
        TreeNode node = queue.poll();
        // 左孩子
        if (index < values.length && !values[index].equals("#")) {
            node.left = new TreeNode(Integer.parseInt(values[index]));
            queue.add(node.left);
        }
        index++;
        // 右孩子
        if (index < values.length && !values[index].equals("#")) {
            node.right = new TreeNode(Integer.parseInt(values[index]));
```

```
            queue.add(node.right);
        }
        index++;
    }
    return root;
}

/**
 * 反序列化优化版本，更健壮的边界处理
 * 时间复杂度：O(N)
 * 空间复杂度：O(N)
 */
public static TreeNode deserializeOptimized(String data) {
    if (data == null || data.trim().isEmpty() || data.equals("#")) {
        return null;
    }

    String[] values = data.split(",");
    if (values.length == 0 || values[0].equals("#")) {
        return null;
    }

    // 头节点
    TreeNode root = new TreeNode(Integer.parseInt(values[0]));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    int index = 1;
    while (!queue.isEmpty() && index < values.length) {
        TreeNode current = queue.poll();

        // 处理左子节点
        if (index < values.length) {
            String leftVal = values[index++];
            if (!leftVal.equals("#")) {
                current.left = new TreeNode(Integer.parseInt(leftVal));
                queue.offer(current.left);
            }
        }

        // 处理右子节点
        if (index < values.length) {
            String rightVal = values[index++];
```

```
            if (!rightVal.equals("#")) {
                current.right = new TreeNode(Integer.parseInt(rightVal));
                queue.offer(current.right);
            }
        }
    }

    return root;
}

/**
 * 用于测试的函数，打印层序遍历
 */
public static String levelOrder(TreeNode root) {
    if (root == null) {
        return "#";
    }
    StringBuilder builder = new StringBuilder();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    builder.append(root.val);
    while (!queue.isEmpty()) {
        TreeNode node = queue.poll();
        if (node.left != null) {
            builder.append("," + node.left.val);
            queue.add(node.left);
        } else {
            builder.append(",#");
        }
        if (node.right != null) {
            builder.append("," + node.right.val);
            queue.add(node.right);
        } else {
            builder.append(",#");
        }
    }
    return builder.toString();
}

/**
 * 验证序列化和反序列化是否正确
 */
public static boolean isValidSerialization(TreeNode original) {
```

```
        String serialized = serialize(original);
        TreeNode deserialized = deserialize(serialized);
        String reserialized = serialize(deserialized);
        return serialized.equals(reserialized);
    }

    /**
     * 生成用于测试的二叉树
     */
    public static TreeNode generateTestTree(int type) {
        switch (type) {
            case 1: // 完全二叉树
                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);
                root1.right.left = new TreeNode(6);
                root1.right.right = new TreeNode(7);
                return root1;
            case 2: // 不平衡树
                TreeNode root2 = new TreeNode(1);
                root2.left = new TreeNode(2);
                root2.left.left = new TreeNode(3);
                root2.left.left.left = new TreeNode(4);
                return root2;
            case 3: // 只有右子树
                TreeNode root3 = new TreeNode(1);
                root3.right = new TreeNode(2);
                root3.right.right = new TreeNode(3);
                root3.right.right.right = new TreeNode(4);
                return root3;
            default: // 普通树
                TreeNode root = new TreeNode(1);
                root.left = new TreeNode(2);
                root.right = new TreeNode(3);
                root.right.left = new TreeNode(4);
                root.right.right = new TreeNode(5);
                return root;
        }
    }

    // 提交这个类
```

```
public class Codec {
    public String serialize(TreeNode root) {
        return Code06_LevelorderSerializeAndDeserialize.serialize(root);
    }

    public TreeNode deserialize(String data) {
        return Code06_LevelorderSerializeAndDeserialize.deserialize(data);
    }
}

public static void main(String[] args) {
    // 测试多种树结构
    for (int i = 0; i <= 3; i++) {
        System.out.println("\n测试树类型 " + i + ":");
        TreeNode root = generateTestTree(i);

        // 序列化测试
        String serialized = serialize(root);
        System.out.println("常规序列化结果: " + serialized);

        // 优化序列化测试
        String serializedOpt = serializeOptimized(root);
        System.out.println("优化序列化结果: " + serializedOpt);

        // 反序列化测试
        TreeNode deserialized = deserialize(serialized);
        String check = levelOrder(deserialized);
        System.out.println("反序列化后层序遍历: " + check);

        // 验证序列化和反序列化的正确性
        boolean isValid = isValidSerialization(root);
        System.out.println("序列化和反序列化正确性: " + isValid);
    }

    // 边界情况测试
    System.out.println("\n边界情况测试:");
    // 空节点
    System.out.println("空节点序列化: " + serialize(null));
    System.out.println("空字符串反序列化: " + levelOrder(deserialize("#")));

    // 单节点
    TreeNode singleNode = new TreeNode(42);
    System.out.println("单节点序列化: " + serialize(singleNode));
```

```
        System.out.println("单节点反序列化: " + levelOrder(deserialize(serialize(singleNode))));

        // 测试优化版本的反序列化
        String optimized = serializeOptimized(generateTestTree(0));
        TreeNode optimizedDeserialized = deserializeOptimized(optimized);
        System.out.println("优化版本反序列化: " + levelOrder(optimizedDeserialized));
    }
}
```

==========

文件: Code07_PreorderInorderBuildBinaryTree.java

==========

```
package class036;

import java.util.HashMap;

// 利用先序与中序遍历序列构造二叉树
// 测试链接 : https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
public class Code07_PreorderInorderBuildBinaryTree {

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

        public TreeNode(int v) {
            val = v;
        }
    }

    // 提交如下的方法
    public static TreeNode buildTree(int[] pre, int[] in) {
        if (pre == null || in == null || pre.length != in.length) {
            return null;
        }
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < in.length; i++) {
            map.put(in[i], i);
        }
        return f(pre, 0, pre.length - 1, in, 0, in.length - 1, map);
```

```
    }

    public static TreeNode f(int[] pre, int l1, int r1, int[] in, int l2, int r2, HashMap<Integer,
Integer> map) {
        if (l1 > r1) {
            return null;
        }
        TreeNode head = new TreeNode(pre[l1]);
        if (l1 == r1) {
            return head;
        }
        int k = map.get(pre[l1]);
        // pre : l1(........)[.......r1]
        // in  : (l2......)k[........r2]
        // (...)是左树对应，[...]是右树的对应
        head.left = f(pre, l1 + 1, l1 + k - l2, in, l2, k - 1, map);
        head.right = f(pre, l1 + k - l2 + 1, r1, in, k + 1, r2, map);
        return head;
    }

}
```

==========

文件: Code08_CompletenessOfBinaryTree.java

==========

```
package class036;

// 验证完全二叉树
// 测试链接 : https://leetcode.cn/problems/check-completeness-of-a-binary-tree/
public class Code08_CompletenessOfBinaryTree {

    // 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
    // 如果测试数据量变大了就修改这个值
    public static int MAXN = 101;
```

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

    public static int l, r;

    public static boolean isCompleteTree(TreeNode h) {
        if (h == null) {
            return true;
        }
        l = r = 0;
        queue[r++] = h;
        // 是否遇到过左右两个孩子不双全的节点
        boolean leaf = false;
        while (l < r) {
            h = queue[l++];
            if ((h.left == null && h.right != null) || (leaf && (h.left != null || h.right !=
null))) {
                return false;
            }
            if (h.left != null) {
                queue[r++] = h.left;
            }
            if (h.right != null) {
                queue[r++] = h.right;
            }
            if (h.left == null || h.right == null) {
                leaf = true;
            }
        }
        return true;
    }

}
```

==========

文件: Code09_CountCompleteTreeNodes.java

==========

```
package class036;

// 求完全二叉树的节点个数
// 测试链接 : https://leetcode.cn/problems/count-complete-tree-nodes/
public class Code09_CountCompleteTreeNodes {
```

```
    // 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 countNodes(TreeNode head) {
        if (head == null) {
            return 0;
        }
        return f(head, 1, mostLeft(head, 1));
    }

    // cur : 当前来到的节点
    // level :  当前来到的节点在第几层
    // h : 整棵树的高度，不是 cur 这棵子树的高度
    // 求 : cur 这棵子树上有多少节点
    public static int f(TreeNode cur, int level, int h) {
        if (level == h) {
            return 1;
        }
        if (mostLeft(cur.right, level + 1) == h) {
            // cur 右树上的最左节点，扎到了最深层
            return (1 << (h - level)) + f(cur.right, level + 1, h);
        } else {
            // cur 右树上的最左节点，没扎到最深层
            return (1 << (h - level - 1)) + f(cur.left, level + 1, h);
        }
    }

    // 当前节点是 cur，并且它在 level 层
    // 返回从 cur 开始不停往左，能扎到几层
    public static int mostLeft(TreeNode cur, int level) {
        while (cur != null) {
            level++;
            cur = cur.left;
        }
        return level - 1;
    }

}
```

```
==========================================

文件: Codeforces1335B_ConstructTheString.java
==========================================
package class036;

import java.util.*;

/**
 * Codeforces 1335B. Construct the String
 * 题目链接: https://codeforces.com/problemset/problem/1335/B
 * 题目描述: 构造一个长度为 n 的字符串，使得任意长度为 a 的子串中恰好包含 b 个不同的字符。
 *
 * 核心算法思想:
 * 1. 循环构造: 使用长度为 b 的字符循环来构造字符串
 * 2. 滑动窗口: 确保每个长度为 a 的窗口包含恰好 b 个不同字符
 * 3. 模式重复: 通过重复特定模式来满足条件
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，需要构造长度为 n 的字符串
 *
 * 空间复杂度分析:
 * - 方法 1(循环构造): O(N)，存储结果字符串
 * - 方法 2(优化构造): O(N)，存储结果字符串
 *
 * 相关题目:
 * 1. LeetCode 1100. 长度为 K 的无重复字符子串 - 类似滑动窗口
 * 2. LeetCode 340. 至多包含 K 个不同字符的最长子串 - 滑动窗口变种
 * 3. Codeforces 1328B. K-th Beautiful String - 字符串构造
 *
 * 工程化考量:
 * 1. 字符选择: 使用小写字母表保证字符充足
 * 2. 边界处理: 处理 b>a 或 b>26 的特殊情况
 * 3. 性能优化: 使用 StringBuilder 提升字符串构造效率
 */
public class Codeforces1335B_ConstructTheString {

    /**
     * 方法 1: 循环构造法 - 基础实现
     * 思路: 使用长度为 b 的字符循环来构造字符串
     * 时间复杂度: O(N) - 构造长度为 n 的字符串
     * 空间复杂度: O(N) - 存储结果字符串
```

```
 *
 * 核心思想:
 * 1. 使用前 b 个小写字母作为字符集
 * 2. 循环重复这 b 个字符来构造整个字符串
 * 3. 确保每个长度为 a 的窗口包含完整的 b 个不同字符
 */
public static String constructString1(int n, int a, int b) {
    if (b > a || b > 26) {
        return ""; // 无效输入
    }

    StringBuilder sb = new StringBuilder();

    // 生成 b 个不同的字符
    char[] chars = new char[b];
    for (int i = 0; i < b; i++) {
        chars[i] = (char) ('a' + i);
    }

    // 循环构造字符串
    for (int i = 0; i < n; i++) {
        sb.append(chars[i % b]);
    }

    return sb.toString();
}

/**
 * 方法 2: 优化构造法 - 处理 a>b 的情况
 * 思路: 当 a>b 时，需要在循环模式中添加重复字符
 * 时间复杂度: O(N) - 构造长度为 n 的字符串
 * 空间复杂度: O(N) - 存储结果字符串
 *
 * 关键优化:
 * 1. 当 a>b 时，使用前 b 个字符加上重复字符来构造
 * 2. 确保窗口内字符多样性满足要求
 */
public static String constructString2(int n, int a, int b) {
    if (b > a || b > 26) {
        return "";
    }

    StringBuilder sb = new StringBuilder();
```

```
    char[] pattern = new char[a];

    // 构造长度为 a 的模式
    for (int i = 0; i < a; i++) {
        if (i < b) {
            pattern[i] = (char) ('a' + i);
        } else {
            pattern[i] = pattern[i % b]; // 重复前 b 个字符
        }
    }

    // 重复模式构造整个字符串
    for (int i = 0; i < n; i++) {
        sb.append(pattern[i % a]);
    }

    return sb.toString();
}

/**
 * 方法 3: 滑动窗口验证法 - 构造并验证
 * 思路: 先构造字符串，然后验证是否满足条件
 * 时间复杂度: O(N) - 构造和验证
 * 空间复杂度: O(N) - 存储结果字符串和频率数组
 */
public static String constructString3(int n, int a, int b) {
    if (b > a || b > 26) {
        return "";
    }

    // 使用方法 1 构造字符串
    String result = constructString1(n, a, b);

    // 验证是否满足条件
    if (validateString(result, n, a, b)) {
        return result;
    } else {
        // 如果不满足，使用方法 2
        return constructString2(n, a, b);
    }
}

/**
```

```
     * 验证字符串是否满足条件
     */
    private static boolean validateString(String s, int n, int a, int b) {
        if (s.length() != n) {
            return false;
        }

        // 检查每个长度为 a 的子串
        for (int i = 0; i <= n - a; i++) {
            String substring = s.substring(i, i + a);
            if (countDistinctChars(substring) != b) {
                return false;
            }
        }

        return true;
    }

    /**
     * 计算字符串中不同字符的数量
     */
    private static int countDistinctChars(String s) {
        boolean[] visited = new boolean[26];
        int count = 0;

        for (char c : s.toCharArray()) {
            int index = c - 'a';
            if (!visited[index]) {
                visited[index] = true;
                count++;
            }
        }

        return count;
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== Codeforces 1335B 测试 ==========");

        // 测试用例 1: n=5, a=5, b=2
```

```
        System.out.println("测试用例 1: n=5, a=5, b=2");
        System.out.println("方法 1 结果: " + constructString1(5, 5, 2));
        System.out.println("方法 2 结果: " + constructString2(5, 5, 2));

        // 测试用例 2: n=8, a=3, b=2
        System.out.println("\n 测试用例 2: n=8, a=3, b=2");
        System.out.println("方法 1 结果: " + constructString1(8, 3, 2));
        System.out.println("方法 2 结果: " + constructString2(8, 3, 2));

        // 测试用例 3: n=10, a=4, b=3
        System.out.println("\n 测试用例 3: n=10, a=4, b=3");
        System.out.println("方法 1 结果: " + constructString1(10, 4, 3));
        System.out.println("方法 2 结果: " + constructString2(10, 4, 3));

        // 边界测试
        System.out.println("\n 边界测试: n=26, a=26, b=26");
        System.out.println("方法 1 结果: " + constructString1(26, 26, 26));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（循环构造）: 简单高效，适用于大多数情况");
        System.out.println("2. 方法 2（优化构造）: 处理 a>b 的情况更准确");
        System.out.println("3. 方法 3（验证构造）: 保证结果正确性，但性能稍差");
    }
}

/*
Python 实现:

def constructString(n: int, a: int, b: int) -> str:
    if b > a or b > 26:
        return ""

    # 生成 b 个不同的字符
    chars = [chr(ord('a') + i) for i in range(b)]

    # 循环构造字符串
    result = []
    for i in range(n):
        result.append(chars[i % b])

    return ''.join(result)
```

C++实现:

```
#include <iostream>
#include <string>
using namespace std;

string constructString(int n, int a, int b) {
    if (b > a || b > 26) {
        return "";
    }

    string result;
    for (int i = 0; i < n; i++) {
        result += 'a' + (i % b);
    }

    return result;
}
*/
```

==========

文件: HackerRank_TreeLevelOrderTraversal.cpp

==========

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

// HackerRank Tree: Level Order Traversal
// 题目链接: https://www.hackerrank.com/challenges/tree-level-order-traversal/problem
// 题目大意: 给你一个二叉树的根节点，按照层序遍历的方式打印所有节点的值，从左到右，一层一层地打印。

// 二叉树节点定义
struct Node {
    int data;
    Node *left;
    Node *right;

    Node(int val) {
        data = val;
        left = NULL;
        right = NULL;
```

```
    }
};

/**
 * 层序遍历实现
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 从根节点开始，将节点加入队列
 * 3. 当队列不为空时，取出队首节点并打印其值
 * 4. 将该节点的左右子节点（如果存在）加入队列
 * 5. 重复步骤 3-4 直到队列为空
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
void levelOrder(Node* root) {
    if (root == NULL) {
        return;
    }

    // 使用队列存储待访问的节点
    queue<Node*> q;
    q.push(root);

    // 当队列不为空时继续遍历
    while (!q.empty()) {
        // 取出队首节点
        Node* current = q.front();
        q.pop();

        // 打印当前节点的值
        cout << current->data << " ";

        // 将左右子节点加入队列（如果存在）
        if (current->left != NULL) {
            q.push(current->left);
        }
        if (current->right != NULL) {
            q.push(current->right);
        }
    }
}

// 以下代码是 HackerRank 提供的测试框架，无需修改
```

```
int main() {
    // 注意：这是简化版本，实际 HackerRank 会有完整的测试框架
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);

    levelOrder(root);

    return 0;
}
```

==========

文件: HackerRank_TreeLevelOrderTraversal.java

==========

```
package class036;

import java.util.*;

// HackerRank Tree: Level Order Traversal
// 题目链接: https://www.hackerrank.com/challenges/tree-level-order-traversal/problem
// 题目大意: 给你一个二叉树的根节点，按照层序遍历的方式打印所有节点的值，从左到右，一层一层地打印。

class Node {
    Node left;
    Node right;
    int data;

    Node(int data) {
        this.data = data;
        left = null;
        right = null;
    }
}

class HackerRank_TreeLevelOrderTraversal {

    /**
     * 层序遍历实现
     * 思路:
```

```
     * 1. 使用队列进行层序遍历
     * 2. 从根节点开始，将节点加入队列
     * 3. 当队列不为空时，取出队首节点并打印其值
     * 4. 将该节点的左右子节点（如果存在）加入队列
     * 5. 重复步骤 3-4 直到队列为空
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    public static void levelOrder(Node root) {
        if (root == null) {
            return;
        }

        // 使用队列存储待访问的节点
        Queue<Node> queue = new LinkedList<>();
        queue.offer(root);

        // 当队列不为空时继续遍历
        while (!queue.isEmpty()) {
            // 取出队首节点
            Node current = queue.poll();

            // 打印当前节点的值
            System.out.print(current.data + " ");

            // 将左右子节点加入队列（如果存在）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }
    }

    // 以下代码是 HackerRank 提供的测试框架，无需修改
    public static Node insert(Node root, int data) {
        if(root == null) {
            return new Node(data);
        } else {
            Node cur;
            if(data <= root.data) {
                cur = insert(root.left, data);
```

```
                root.left = cur;
            } else {
                cur = insert(root.right, data);
                root.right = cur;
            }
            return root;
        }
    }

    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int t = scan.nextInt();
        Node root = null;
        while(t-- > 0) {
            int data = scan.nextInt();
            root = insert(root, data);
        }
        scan.close();
        levelOrder(root);
    }
}
```

==========

文件: HackerRank_TreeLevelOrderTraversal.py

==========

```
from collections import deque
from typing import Optional

# HackerRank Tree: Level Order Traversal
# 题目链接: https://www.hackerrank.com/challenges/tree-level-order-traversal/problem
# 题目大意: 给你一个二叉树的根节点，按照层序遍历的方式打印所有节点的值，从左到右，一层一层地打印。

class Node:
    def __init__(self, info):
        self.info = info
        self.left: Optional['Node'] = None
        self.right: Optional['Node'] = None
        self.level = None

    def __str__(self):
        return str(self.info)
```

```
class BinarySearchTree:
    def __init__(self):
        self.root: Optional[Node] = None

    def create(self, val):
        if self.root == None:
            self.root = Node(val)
        else:
            current = self.root

            while True:
                if val < current.info:
                    if current.left:
                        current = current.left
                    else:
                        current.left = Node(val)
                        break
                elif val > current.info:
                    if current.right:
                        current = current.right
                    else:
                        current.right = Node(val)
                        break
                else:
                    break

def levelOrder(root: Optional[Node]) -> None:
    """
    层序遍历实现
    思路:
    1. 使用队列进行层序遍历
    2. 从根节点开始，将节点加入队列
    3. 当队列不为空时，取出队首节点并打印其值
    4. 将该节点的左右子节点（如果存在）加入队列
    5. 重复步骤 3-4 直到队列为空
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
    """
    if not root:
        return

    # 使用双端队列存储待访问的节点
```

```
    queue = deque([root])

    # 当队列不为空时继续遍历
    while queue:
        # 取出队首节点
        current = queue.popleft()

        # 打印当前节点的值
        print(current.info, end=" ")

        # 将左右子节点加入队列（如果存在）
        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

# 以下代码是 HackerRank 提供的测试框架，无需修改
if __name__ == "__main__":
    tree = BinarySearchTree()
    t = int(input())

    arr = list(map(int, input().split()))

    for i in range(t):
        tree.create(arr[i])

    levelOrder(tree.root)
```

```
==========================================
```

文件: LeetCode100_SameTree.java

```
==========================================
package class036;

import java.util.*;

// LeetCode 100. 相同的树
// 题目链接: https://leetcode.cn/problems/same-tree/
// 题目大意: 给你两棵二叉树的根节点 p 和 q ，编写一个函数来检验这两棵树是否相同。
// 如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

public class LeetCode100_SameTree {
```

```
// 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: 递归实现判断两棵树是否相同
 * 思路:
 * 1. 如果两个节点都为空，返回 true
 * 2. 如果其中一个节点为空，另一个不为空，返回 false
 * 3. 如果两个节点的值不相等，返回 false
 * 4. 递归判断左子树和右子树是否相同
 * 5. 返回左右子树都相同的判断结果
 * 时间复杂度: O(min(m,n)) - m 和 n 分别是两棵树的节点数
 * 空间复杂度: O(min(h1,h2)) - h1 和 h2 分别是两棵树的高度，递归调用栈的深度
 */
public static boolean isSameTree1(TreeNode p, TreeNode q) {
    // 如果两个节点都为空，返回 true
    if (p == null && q == null) {
        return true;
    }

    // 如果其中一个节点为空，另一个不为空，返回 false
    if (p == null || q == null) {
        return false;
    }

    // 如果两个节点的值不相等，返回 false
    if (p.val != q.val) {
        return false;
    }

    // 递归判断左子树和右子树是否相同
    return isSameTree1(p.left, q.left) && isSameTree1(p.right, q.right);
```

```
}

/**
 * 方法 2: 迭代实现判断两棵树是否相同
 * 思路:
 * 1. 使用队列存储待比较的节点对
 * 2. 每次从队列中取出一对节点进行比较
 * 3. 如果节点对都为空，继续下一对
 * 4. 如果其中一个为空或值不相等，返回 false
 * 5. 将左右子节点对加入队列继续比较
 * 时间复杂度: O(min(m,n)) - m 和 n 分别是两棵树的节点数
 * 空间复杂度: O(min(w1,w2)) - w1 和 w2 分别是两棵树的最大宽度
 */
public static boolean isSameTree2(TreeNode p, TreeNode q) {
    // 使用队列存储待比较的节点对
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(p);
    queue.offer(q);

    while (!queue.isEmpty()) {
        // 取出一对节点
        TreeNode node1 = queue.poll();
        TreeNode node2 = queue.poll();

        // 如果两个节点都为空，继续下一对
        if (node1 == null && node2 == null) {
            continue;
        }

        // 如果其中一个节点为空或值不相等，返回 false
        if (node1 == null || node2 == null || node1.val != node2.val) {
            return false;
        }

        // 将左右子节点对加入队列继续比较
        queue.offer(node1.left);
        queue.offer(node2.left);
        queue.offer(node1.right);
        queue.offer(node2.right);
    }

    return true;
}
```

```
// 测试方法
public static void main(String[] args) {
    // 构建测试二叉树 1:
    //     1
    //    / \
    //   2   3
    TreeNode p1 = new TreeNode(1);
    p1.left = new TreeNode(2);
    p1.right = new TreeNode(3);

    // 构建测试二叉树 2:
    //     1
    //    / \
    //   2   3
    TreeNode q1 = new TreeNode(1);
    q1.left = new TreeNode(2);
    q1.right = new TreeNode(3);

    System.out.println("测试用例 1 - 相同的树:");
    System.out.println("递归方法: " + isSameTree1(p1, q1));
    System.out.println("迭代方法: " + isSameTree2(p1, q1));

    // 构建测试二叉树 3:
    //     1
    //    /
    //   2
    TreeNode p2 = new TreeNode(1);
    p2.left = new TreeNode(2);

    // 构建测试二叉树 4:
    //     1
    //      \
    //       2
    TreeNode q2 = new TreeNode(1);
    q2.right = new TreeNode(2);

    System.out.println("\n 测试用例 2 - 不同的树:");
    System.out.println("递归方法: " + isSameTree1(p2, q2));
    System.out.println("迭代方法: " + isSameTree2(p2, q2));

    // 测试空树
    TreeNode empty1 = null;
```

```
        TreeNode empty2 = null;
        System.out.println("\n测试用例 3 - 空树:");
        System.out.println("递归方法: " + isSameTree1(empty1, empty2));
        System.out.println("迭代方法: " + isSameTree2(empty1, empty2));
    }
}
```

==========

文件: LeetCode100_SameTree.py

==========

```
from typing import Optional
from collections import deque

# LeetCode 100. 相同的树
# 题目链接: https://leetcode.cn/problems/same-tree/
# 题目大意: 给你两棵二叉树的根节点 p 和 q ，编写一个函数来检验这两棵树是否相同。
# 如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

# 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 isSameTree1(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        """
        方法 1: 递归实现判断两棵树是否相同
        思路:
        1. 如果两个节点都为空，返回 true
        2. 如果其中一个节点为空，另一个不为空，返回 false
        3. 如果两个节点的值不相等，返回 false
        4. 递归判断左子树和右子树是否相同
        5. 返回左右子树都相同的判断结果
        时间复杂度: O(min(m,n)) - m 和 n 分别是两棵树的节点数
        空间复杂度: O(min(h1,h2)) - h1 和 h2 分别是两棵树的高度，递归调用栈的深度
        """
        # 如果两个节点都为空，返回 True
        if not p and not q:
            return True
```

```
    # 如果其中一个节点为空，另一个不为空，返回 False
    if not p or not q:
        return False

    # 如果两个节点的值不相等，返回 False
    if p.val != q.val:
        return False

    # 递归判断左子树和右子树是否相同
    return self.isSameTree1(p.left, q.left) and self.isSameTree1(p.right, q.right)

def isSameTree2(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
    """
    方法 2: 迭代实现判断两棵树是否相同
    思路:
    1. 使用队列存储待比较的节点对
    2. 每次从队列中取出一对节点进行比较
    3. 如果节点对都为空，继续下一对
    4. 如果其中一个为空或值不相等，返回 false
    5. 将左右子节点对加入队列继续比较
    时间复杂度: O(min(m,n)) - m 和 n 分别是两棵树的节点数
    空间复杂度: O(min(w1,w2)) - w1 和 w2 分别是两棵树的最大宽度
    """
    # 使用队列存储待比较的节点对
    queue = deque([p, q])

    while queue:
        # 取出一对节点
        node1 = queue.popleft()
        node2 = queue.popleft()

        # 如果两个节点都为空，继续下一对
        if not node1 and not node2:
            continue

        # 如果其中一个节点为空或值不相等，返回 False
        if not node1 or not node2 or node1.val != node2.val:
            return False

        # 将左右子节点对加入队列继续比较
        queue.append(node1.left)
        queue.append(node2.left)
        queue.append(node1.right)
```

```
            queue.append(node2.right)

        return True

# 测试代码
if __name__ == "__main__":
    # 构建测试二叉树 1:
    #     1
    #    / \
    #   2   3
    p1 = TreeNode(1)
    p1.left = TreeNode(2)
    p1.right = TreeNode(3)

    # 构建测试二叉树 2:
    #     1
    #    / \
    #   2   3
    q1 = TreeNode(1)
    q1.left = TreeNode(2)
    q1.right = TreeNode(3)

    solution = Solution()
    print("测试用例 1 - 相同的树:")
    print("递归方法:", solution.isSameTree1(p1, q1))
    print("迭代方法:", solution.isSameTree2(p1, q1))

    # 构建测试二叉树 3:
    #     1
    #    /
    #   2
    p2 = TreeNode(1)
    p2.left = TreeNode(2)

    # 构建测试二叉树 4:
    #     1
    #      \
    #       2
    q2 = TreeNode(1)
    q2.right = TreeNode(2)

    print("\n测试用例 2 - 不同的树:")
    print("递归方法:", solution.isSameTree1(p2, q2))
```

```
    print("迭代方法:", solution.isSameTree2(p2, q2))

    # 测试空树
    empty1 = None
    empty2 = None
    print("\n测试用例 3 - 空树:")
    print("递归方法:", solution.isSameTree1(empty1, empty2))
    print("迭代方法:", solution.isSameTree2(empty1, empty2))

==========================================

文件: LeetCode101_SymmetricTree.java
==========================================
package class036;

import java.util.*;

// LeetCode 101. 对称二叉树
// 题目链接: https://leetcode.cn/problems/symmetric-tree/
// 题目大意: 给你一个二叉树的根节点 root ，检查它是否轴对称。

public class LeetCode101_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: 递归实现判断二叉树是否对称
     * 思路:
     * 1. 创建辅助函数比较两个子树是否互为镜像
     * 2. 如果两个节点都为空，返回 true
     * 3. 如果其中一个节点为空，另一个不为空，返回 false
```

```
     * 4. 如果两个节点的值不相等，返回 false
     * 5. 递归比较左子树的左子树与右子树的右子树，以及左子树的右子树与右子树的左子树
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    public static boolean isSymmetric1(TreeNode root) {
        if (root == null) {
            return true;
        }
        return isMirror(root.left, root.right);
    }

    /**
     * 辅助函数：判断两个子树是否互为镜像
     * @param left 左子树根节点
     * @param right 右子树根节点
     * @return 是否互为镜像
     */
    private static boolean isMirror(TreeNode left, TreeNode right) {
        // 如果两个节点都为空，返回 true
        if (left == null && right == null) {
            return true;
        }

        // 如果其中一个节点为空，另一个不为空，返回 false
        if (left == null || right == null) {
            return false;
        }

        // 如果两个节点的值不相等，返回 false
        if (left.val != right.val) {
            return false;
        }

        // 递归比较左子树的左子树与右子树的右子树，以及左子树的右子树与右子树的左子树
        return isMirror(left.left, right.right) && isMirror(left.right, right.left);
    }

    /**
     * 方法 2: 迭代实现判断二叉树是否对称
     * 思路:
     * 1. 使用队列存储待比较的节点对
     * 2. 每次从队列中取出一对节点进行比较
```

```
     * 3. 如果节点对都为空，继续下一对
     * 4. 如果其中一个为空或值不相等，返回 false
     * 5. 将左子树的左子节点与右子树的右子节点、左子树的右子节点与右子树的左子节点加入队列
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    public static boolean isSymmetric2(TreeNode root) {
        if (root == null) {
            return true;
        }

        // 使用队列存储待比较的节点对
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root.left);
        queue.offer(root.right);

        while (!queue.isEmpty()) {
            // 取出一对节点
            TreeNode left = queue.poll();
            TreeNode right = queue.poll();

            // 如果两个节点都为空，继续下一对
            if (left == null && right == null) {
                continue;
            }

            // 如果其中一个节点为空或值不相等，返回 false
            if (left == null || right == null || left.val != right.val) {
                return false;
            }

            // 将左子树的左子节点与右子树的右子节点、左子树的右子节点与右子树的左子节点加入队列
            queue.offer(left.left);
            queue.offer(right.right);
            queue.offer(left.right);
            queue.offer(right.left);
        }

        return true;
    }

    // 测试方法
    public static void main(String[] args) {
```

```
        // 构建测试对称二叉树:
        //     1
        //    / \
        //   2   2
        //  / \ / \
        // 3  4 4  3
        TreeNode symmetricRoot = new TreeNode(1);
        symmetricRoot.left = new TreeNode(2);
        symmetricRoot.right = new TreeNode(2);
        symmetricRoot.left.left = new TreeNode(3);
        symmetricRoot.left.right = new TreeNode(4);
        symmetricRoot.right.left = new TreeNode(4);
        symmetricRoot.right.right = new TreeNode(3);

        System.out.println("测试用例 1 - 对称二叉树:");
        System.out.println("递归方法: " + isSymmetric1(symmetricRoot));
        System.out.println("迭代方法: " + isSymmetric2(symmetricRoot));

        // 构建测试非对称二叉树:
        //     1
        //    / \
        //   2   2
        //    \   \
        //     3   3
        TreeNode asymmetricRoot = new TreeNode(1);
        asymmetricRoot.left = new TreeNode(2);
        asymmetricRoot.right = new TreeNode(2);
        asymmetricRoot.left.right = new TreeNode(3);
        asymmetricRoot.right.right = new TreeNode(3);

        System.out.println("\n测试用例 2 - 非对称二叉树:");
        System.out.println("递归方法: " + isSymmetric1(asymmetricRoot));
        System.out.println("迭代方法: " + isSymmetric2(asymmetricRoot));

        // 测试空树
        TreeNode emptyRoot = null;
        System.out.println("\n测试用例 3 - 空树:");
        System.out.println("递归方法: " + isSymmetric1(emptyRoot));
        System.out.println("迭代方法: " + isSymmetric2(emptyRoot));
    }
}
```

==========

文件: LeetCode101_SymmetricTree.py

```
==========================================
from typing import Optional
from collections import deque

# LeetCode 101. 对称二叉树
# 题目链接: https://leetcode.cn/problems/symmetric-tree/
# 题目大意: 给你一个二叉树的根节点 root ，检查它是否轴对称。

# 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 isSymmetric1(self, root: Optional[TreeNode]) -> bool:
        """
        方法 1: 递归实现判断二叉树是否对称
        思路:
        1. 创建辅助函数比较两个子树是否互为镜像
        2. 如果两个节点都为空，返回 true
        3. 如果其中一个节点为空，另一个不为空，返回 false
        4. 如果两个节点的值不相等，返回 false
        5. 递归比较左子树的左子树与右子树的右子树，以及左子树的右子树与右子树的左子树
        时间复杂度: O(n) - n 是节点数量，每个节点访问一次
        空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
        """
        if not root:
            return True
        return self.isMirror(root.left, root.right)

    def isMirror(self, left: Optional[TreeNode], right: Optional[TreeNode]) -> bool:
        """
        辅助函数：判断两个子树是否互为镜像
        :param left: 左子树根节点
        :param right: 右子树根节点
        :return: 是否互为镜像
        """
        # 如果两个节点都为空，返回 True
        if not left and not right:
```

```
        return True

    # 如果其中一个节点为空，另一个不为空，返回 False
    if not left or not right:
        return False

    # 如果两个节点的值不相等，返回 False
    if left.val != right.val:
        return False

    # 递归比较左子树的左子树与右子树的右子树，以及左子树的右子树与右子树的左子树
    return self.isMirror(left.left, right.right) and self.isMirror(left.right, right.left)

def isSymmetric2(self, root: Optional[TreeNode]) -> bool:
    """
    方法 2: 迭代实现判断二叉树是否对称
    思路:
    1. 使用队列存储待比较的节点对
    2. 每次从队列中取出一对节点进行比较
    3. 如果节点对都为空，继续下一对
    4. 如果其中一个为空或值不相等，返回 false
    5. 将左子树的左子节点与右子树的右子节点、左子树的右子节点与右子树的左子节点加入队列
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
    """
    if not root:
        return True

    # 使用队列存储待比较的节点对
    queue = deque([root.left, root.right])

    while queue:
        # 取出一对节点
        left = queue.popleft()
        right = queue.popleft()

        # 如果两个节点都为空，继续下一对
        if not left and not right:
            continue

        # 如果其中一个节点为空或值不相等，返回 False
        if not left or not right or left.val != right.val:
            return False
```

```
            # 将左子树的左子节点与右子树的右子节点、左子树的右子节点与右子树的左子节点加入队列
            queue.append(left.left)
            queue.append(right.right)
            queue.append(left.right)
            queue.append(right.left)

        return True

# 测试代码
if __name__ == "__main__":
    # 构建测试对称二叉树:
    #     1
    #    / \
    #   2   2
    #  / \ / \
    # 3  4 4  3
    symmetric_root = TreeNode(1)
    symmetric_root.left = TreeNode(2)
    symmetric_root.right = TreeNode(2)
    symmetric_root.left.left = TreeNode(3)
    symmetric_root.left.right = TreeNode(4)
    symmetric_root.right.left = TreeNode(4)
    symmetric_root.right.right = TreeNode(3)

    solution = Solution()
    print("测试用例 1 - 对称二叉树:")
    print("递归方法:", solution.isSymmetric1(symmetric_root))
    print("迭代方法:", solution.isSymmetric2(symmetric_root))

    # 构建测试非对称二叉树:
    #     1
    #    / \
    #   2   2
    #    \   \
    #     3   3
    asymmetric_root = TreeNode(1)
    asymmetric_root.left = TreeNode(2)
    asymmetric_root.right = TreeNode(2)
    asymmetric_root.left.right = TreeNode(3)
    asymmetric_root.right.right = TreeNode(3)

    print("\n测试用例 2 - 非对称二叉树:")
```

```
    print("递归方法:", solution.isSymmetric1(asymmetric_root))
    print("迭代方法:", solution.isSymmetric2(asymmetric_root))

    # 测试空树
    empty_root = None
    print("\n测试用例 3 - 空树:")
    print("递归方法:", solution.isSymmetric1(empty_root))
    print("迭代方法:", solution.isSymmetric2(empty_root))

==========

文件: LeetCode103_BinaryTreeZigzagLevelOrderTraversal.cpp
==========
// LeetCode 103. 二叉树的锯齿形层序遍历
// 题目链接: https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
// 题目大意: 给你二叉树的根节点 root ，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）

#include <vector>
#include <queue>
#include <deque>
#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: 使用双端队列实现锯齿形层序遍历
     * 思路:
     * 1. 使用一个布尔变量记录当前层的遍历方向（从左到右或从右到左）
     * 2. 对于每一层，根据遍历方向决定是从队列头部取节点还是从尾部取节点
     * 3. 添加子节点时也根据方向决定是添加到头部还是尾部
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
```

```
 * 空间复杂度: O(n) - 存储队列和结果
 */
vector<vector<int>> zigzagLevelOrder1(TreeNode* root) {
    vector<vector<int>> ans;
    if (root) {
        // 使用双端队列存储节点
        deque<TreeNode*> dq;
        dq.push_back(root);
        // true 表示从左到右，false 表示从右到左
        bool leftToRight = true;

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

            if (leftToRight) {
                // 从左到右遍历：从头部取节点，子节点添加到尾部
                for (int i = 0; i < size; i++) {
                    TreeNode* node = dq.front();
                    dq.pop_front();
                    level.push_back(node->val);
                    // 先添加左子节点，再添加右子节点
                    if (node->left) dq.push_back(node->left);
                    if (node->right) dq.push_back(node->right);
                }
            } else {
                // 从右到左遍历：从尾部取节点，子节点添加到头部
                for (int i = 0; i < size; i++) {
                    TreeNode* node = dq.back();
                    dq.pop_back();
                    level.push_back(node->val);
                    // 先添加右子节点，再添加左子节点
                    if (node->right) dq.push_front(node->right);
                    if (node->left) dq.push_front(node->left);
                }
            }

            ans.push_back(level);
            // 切换方向
            leftToRight = !leftToRight;
        }
    }
    return ans;
```

```
}

/**
 * 方法 2: 使用普通队列实现锯齿形层序遍历
 * 思路:
 * 1. 使用普通队列进行层序遍历
 * 2. 使用一个布尔变量记录当前层是否需要反转
 * 3. 对于需要反转的层，在添加到结果之前进行反转
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储队列和结果
 */
vector<vector<int>> zigzagLevelOrder2(TreeNode* root) {
    vector<vector<int>> ans;
    if (root) {
        queue<TreeNode*> q;
        q.push(root);
        // true 表示从左到右，false 表示从右到左
        bool leftToRight = true;

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

            // 正常的层序遍历
            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);
            }

            // 如果当前层需要从右到左，则反转列表
            if (!leftToRight) {
                reverse(level.begin(), level.end());
            }

            ans.push_back(level);
            // 切换方向
            leftToRight = !leftToRight;
        }
    }
    return ans;
```

```
    }

    /**
     * 方法 3: 使用递归实现锯齿形层序遍历
     * 思路:
     * 1. 使用递归进行深度优先遍历
     * 2. 根据层数的奇偶性决定节点值的添加方向
     * 3. 奇数层从右到左，偶数层从左到右
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    vector<vector<int>> zigzagLevelOrder3(TreeNode* root) {
        vector<vector<int>> ans;
        dfs(root, 0, ans);
        return ans;
    }

private:
    /**
     * 递归辅助函数
     * @param node 当前节点
     * @param level 当前层数
     * @param ans 结果列表
     */
    void dfs(TreeNode* node, int level, vector<vector<int>>& ans) {
        if (!node) {
            return;
        }

        // 如果当前层还没有对应的列表，创建一个新的
        if (ans.size() <= level) {
            ans.push_back({});
        }

        // 根据层数的奇偶性决定添加方向
        if (level % 2 == 0) {
            // 偶数层：从左到右，添加到列表末尾
            ans[level].push_back(node->val);
        } else {
            // 奇数层：从右到左，添加到列表开头
            ans[level].insert(ans[level].begin(), node->val);
        }
```

```
        // 递归处理左右子树
        dfs(node->left, level + 1, ans);
        dfs(node->right, level + 1, ans);
    }
};

// 辅助函数：打印二维向量
void printVector(const vector<vector<int>>& vec) {
    for (const auto& v : vec) {
        cout << "[";
        for (size_t i = 0; i < v.size(); ++i) {
            cout << v[i];
            if (i < v.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
}

// 测试代码
int main() {
    // 测试用例 1: [3,9,20,null,null,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);

    Solution solution;
    cout << "方法 1 结果:" << endl;
    printVector(solution.zigzagLevelOrder1(root1));

    cout << "方法 2 结果:" << endl;
    printVector(solution.zigzagLevelOrder2(root1));

    cout << "方法 3 结果:" << endl;
    printVector(solution.zigzagLevelOrder3(root1));

    // 测试用例 2: [1]
    TreeNode* root2 = new TreeNode(1);
    cout << "单节点树结果:" << endl;
    printVector(solution.zigzagLevelOrder1(root2));

    // 释放内存
```

```
    delete root1->right->right;
    delete root1->right->left;
    delete root1->right;
    delete root1->left;
    delete root1;
    delete root2;

    return 0;
}
```

==========

文件: LeetCode103_BinaryTreeZigzagLevelOrderTraversal.java

==========

```
package class036;

import java.util.*;

// LeetCode 103. 二叉树的锯齿形层序遍历
// 题目链接: https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
// 题目大意: 给你二叉树的根节点 root ，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)

public class LeetCode103_BinaryTreeZigzagLevelOrderTraversal {

    // 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: 使用双端队列实现锯齿形层序遍历
     * 思路:
     * 1. 使用一个布尔变量记录当前层的遍历方向（从左到右或从右到左）
```

```
     * 2. 对于每一层，根据遍历方向决定是从队列头部取节点还是从尾部取节点
     * 3. 添加子节点时也根据方向决定是添加到头部还是尾部
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(n) - 存储队列和结果
     */
    public static List<List<Integer>> zigzagLevelOrder1(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        if (root != null) {
            // 使用双端队列存储节点
            Deque<TreeNode> deque = new LinkedList<>();
            deque.offerFirst(root);
            // true 表示从左到右，false 表示从右到左
            boolean leftToRight = true;

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

                if (leftToRight) {
                    // 从左到右遍历：从头部取节点，子节点添加到尾部
                    for (int i = 0; i < size; i++) {
                        TreeNode node = deque.pollFirst();
                        level.add(node.val);
                        // 先添加左子节点，再添加右子节点
                        if (node.left != null) deque.offerLast(node.left);
                        if (node.right != null) deque.offerLast(node.right);
                    }
                } else {
                    // 从右到左遍历：从尾部取节点，子节点添加到头部
                    for (int i = 0; i < size; i++) {
                        TreeNode node = deque.pollLast();
                        level.add(node.val);
                        // 先添加右子节点，再添加左子节点
                        if (node.right != null) deque.offerFirst(node.right);
                        if (node.left != null) deque.offerFirst(node.left);
                    }
                }

                ans.add(level);
                // 切换方向
                leftToRight = !leftToRight;
            }
        }
```

```
    return ans;
}

/**
 * 方法 2: 使用普通队列实现锯齿形层序遍历
 * 思路:
 * 1. 使用普通队列进行层序遍历
 * 2. 使用一个布尔变量记录当前层是否需要反转
 * 3. 对于需要反转的层，在添加到结果之前进行反转
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储队列和结果
 */
public static List<List<Integer>> zigzagLevelOrder2(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        // true 表示从左到右，false 表示从右到左
        boolean leftToRight = true;

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

            // 正常的层序遍历
            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);
            }

            // 如果当前层需要从右到左，则反转列表
            if (!leftToRight) {
                Collections.reverse(level);
            }

            ans.add(level);
            // 切换方向
            leftToRight = !leftToRight;
        }
    }
    return ans;
```

```
}

/**
 * 方法 3: 使用递归实现锯齿形层序遍历
 * 思路:
 * 1. 使用递归进行深度优先遍历
 * 2. 根据层数的奇偶性决定节点值的添加方向
 * 3. 奇数层从右到左，偶数层从左到右
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
 */
public static List<List<Integer>> zigzagLevelOrder3(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    dfs(root, 0, ans);
    return ans;
}

/**
 * 递归辅助函数
 * @param node 当前节点
 * @param level 当前层数
 * @param ans 结果列表
 */
private static void dfs(TreeNode node, int level, List<List<Integer>> ans) {
    if (node == null) {
        return;
    }

    // 如果当前层还没有对应的列表，创建一个新的
    if (ans.size() <= level) {
        ans.add(new ArrayList<>());
    }

    // 根据层数的奇偶性决定添加方向
    if (level % 2 == 0) {
        // 偶数层：从左到右，添加到列表末尾
        ans.get(level).add(node.val);
    } else {
        // 奇数层：从右到左，添加到列表开头
        ans.get(level).add(0, node.val);
    }

    // 递归处理左右子树
```

```
        dfs(node.left, level + 1, ans);
        dfs(node.right, level + 1, ans);
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1: [3,9,20,null,null,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);

        System.out.println("方法 1 结果:");
        System.out.println(zigzagLevelOrder1(root1));

        System.out.println("方法 2 结果:");
        System.out.println(zigzagLevelOrder2(root1));

        System.out.println("方法 3 结果:");
        System.out.println(zigzagLevelOrder3(root1));

        // 测试用例 2: [1]
        TreeNode root2 = new TreeNode(1);
        System.out.println("单节点树结果:");
        System.out.println(zigzagLevelOrder1(root2));

        // 测试用例 3: []
        TreeNode root3 = null;
        System.out.println("空树结果:");
        System.out.println(zigzagLevelOrder1(root3));
    }
}

================================================

文件: LeetCode103_BinaryTreeZigzagLevelOrderTraversal.py
================================================
from typing import List, Optional
from collections import deque

# LeetCode 103. 二叉树的锯齿形层序遍历
# 题目链接: https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
```

题目大意: 给你二叉树的根节点 root ，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）

```
# 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 zigzagLevelOrder1(self, root: Optional[TreeNode]) -> List[List[int]]:
        """
        方法 1: 使用双端队列实现锯齿形层序遍历
        思路:
        1. 使用一个布尔变量记录当前层的遍历方向（从左到右或从右到左）
        2. 对于每一层，根据遍历方向决定是从队列头部取节点还是从尾部取节点
        3. 添加子节点时也根据方向决定是添加到头部还是尾部
        时间复杂度: O(n) - n 是节点数量，每个节点访问一次
        空间复杂度: O(n) - 存储队列和结果
        """
        ans = []
        if root:
            # 使用双端队列存储节点
            dq = deque([root])
            # True 表示从左到右，False 表示从右到左
            left_to_right = True

            while dq:
                size = len(dq)
                level = []

                if left_to_right:
                    # 从左到右遍历：从头部取节点，子节点添加到尾部
                    for _ in range(size):
                        node = dq.popleft()
                        level.append(node.val)
                        # 先添加左子节点，再添加右子节点
                        if node.left:
                            dq.append(node.left)
                        if node.right:
                            dq.append(node.right)
                else:
```

```
                # 从右到左遍历：从尾部取节点，子节点添加到头部
                for _ in range(size):
                    node = dq.pop()
                    level.append(node.val)
                    # 先添加右子节点，再添加左子节点
                    if node.right:
                        dq.appendleft(node.right)
                    if node.left:
                        dq.appendleft(node.left)

            ans.append(level)
            # 切换方向
            left_to_right = not left_to_right

    return ans


def zigzagLevelOrder2(self, root: Optional[TreeNode]) -> List[List[int]]:
    """
    方法 2: 使用普通队列实现锯齿形层序遍历
    思路:
    1. 使用普通队列进行层序遍历
    2. 使用一个布尔变量记录当前层是否需要反转
    3. 对于需要反转的层，在添加到结果之前进行反转
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(n) - 存储队列和结果
    """
    ans = []
    if root:
        queue = deque([root])
        # True 表示从左到右，False 表示从右到左
        left_to_right = True

        while queue:
            size = len(queue)
            level = []

            # 正常的层序遍历
            for _ in range(size):
                node = queue.popleft()
                level.append(node.val)
                if node.left:
                    queue.append(node.left)
                if node.right:
```

```
                    queue.append(node.right)

            # 如果当前层需要从右到左，则反转列表
            if not left_to_right:
                level.reverse()

            ans.append(level)
            # 切换方向
            left_to_right = not left_to_right

    return ans

def zigzagLevelOrder3(self, root: Optional[TreeNode]) -> List[List[int]]:
    """
    方法 3: 使用递归实现锯齿形层序遍历
    思路:
    1. 使用递归进行深度优先遍历
    2. 根据层数的奇偶性决定节点值的添加方向
    3. 奇数层从右到左，偶数层从左到右
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
    """
    ans = []

    def dfs(node: Optional[TreeNode], level: int) -> None:
        if not node:
            return

        # 如果当前层还没有对应的列表，创建一个新的
        if len(ans) <= level:
            ans.append([])

        # 根据层数的奇偶性决定添加方向
        if level % 2 == 0:
            # 偶数层：从左到右，添加到列表末尾
            ans[level].append(node.val)
        else:
            # 奇数层：从右到左，添加到列表开头
            ans[level].insert(0, node.val)

        # 递归处理左右子树
        dfs(node.left, level + 1)
        dfs(node.right, level + 1)
```

```
        dfs(root, 0)
        return ans

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: [3,9,20,null,null,15,7]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)

    solution = Solution()
    print("方法 1 结果:")
    print(solution.zigzagLevelOrder1(root1))

    print("方法 2 结果:")
    print(solution.zigzagLevelOrder2(root1))

    print("方法 3 结果:")
    print(solution.zigzagLevelOrder3(root1))

    # 测试用例 2: [1]
    root2 = TreeNode(1)
    print("单节点树结果:")
    print(solution.zigzagLevelOrder1(root2))

    # 测试用例 3: []
    root3 = None
    print("空树结果:")
    print(solution.zigzagLevelOrder1(root3))
```

```
================================================
```

文件: LeetCode104_MaximumDepthOfBinaryTree.java

```
================================================
package class036;

import java.util.*;

// LeetCode 104. 二叉树的最大深度
// 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
```

// 题目大意：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。

```
public class LeetCode104_MaximumDepthOfBinaryTree {

    // 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：递归实现计算二叉树的最大深度
     * 思路：
     * 1. 如果当前节点为空，深度为 0
     * 2. 递归计算左子树的最大深度
     * 3. 递归计算右子树的最大深度
     * 4. 返回左右子树最大深度的最大值加 1
     * 时间复杂度：O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度：O(h) - h 是树的高度，递归调用栈的深度
     */
    public static int maxDepth1(TreeNode root) {
        // 递归终止条件
        if (root == null) {
            return 0;
        }

        // 递归计算左子树和右子树的最大深度
        int leftDepth = maxDepth1(root.left);
        int rightDepth = maxDepth1(root.right);

        // 返回左右子树最大深度的最大值加 1
        return Math.max(leftDepth, rightDepth) + 1;
    }
```

```
/**
 * 方法 2: 迭代实现计算二叉树的最大深度
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 记录层数，每处理完一层层数加 1
 * 3. 返回最终的层数
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static int maxDepth2(TreeNode root) {
    if (root == null) {
        return 0;
    }

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

    int depth = 0;

    while (!queue.isEmpty()) {
        // 记录当前层的节点数量
        int size = queue.size();

        // 处理当前层的所有节点
        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();

            // 将左右子节点加入队列（如果存在）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        // 每处理完一层，深度加 1
        depth++;
    }

    return depth;
}
```

```
    // 测试方法
    public static void main(String[] args) {
        // 构建测试二叉树:
        //     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);

        System.out.println("递归方法计算的最大深度: " + maxDepth1(root));
        System.out.println("迭代方法计算的最大深度: " + maxDepth2(root));

        // 测试空树
        TreeNode emptyRoot = null;
        System.out.println("空树的最大深度: " + maxDepth1(emptyRoot));

        // 测试单节点树
        TreeNode singleRoot = new TreeNode(1);
        System.out.println("单节点树的最大深度: " + maxDepth1(singleRoot));
    }
}
```

```
==========================================
```

文件: LeetCode104_MaximumDepthOfBinaryTree.py

```
==========================================
from typing import Optional
from collections import deque

# LeetCode 104. 二叉树的最大深度
# 题目链接: https://leetcode.cn/problems/maximum-depth-of-binary-tree/
# 题目大意: 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。

# 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 maxDepth1(self, root: Optional[TreeNode]) -> int:
        """
        方法 1: 递归实现计算二叉树的最大深度
        思路:
        1. 如果当前节点为空，深度为 0
        2. 递归计算左子树的最大深度
        3. 递归计算右子树的最大深度
        4. 返回左右子树最大深度的最大值加 1
        时间复杂度: O(n) - n 是节点数量，每个节点访问一次
        空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
        """
        # 递归终止条件
        if not root:
            return 0

        # 递归计算左子树和右子树的最大深度
        left_depth = self.maxDepth1(root.left)
        right_depth = self.maxDepth1(root.right)

        # 返回左右子树最大深度的最大值加 1
        return max(left_depth, right_depth) + 1

    def maxDepth2(self, root: Optional[TreeNode]) -> int:
        """
        方法 2: 迭代实现计算二叉树的最大深度
        思路:
        1. 使用队列进行层序遍历
        2. 记录层数，每处理完一层层数加 1
        3. 返回最终的层数
        时间复杂度: O(n) - n 是节点数量，每个节点访问一次
        空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
        """
        if not root:
            return 0

        # 使用队列进行层序遍历
        queue = deque([root])
        depth = 0
```

```
        while queue:
            # 记录当前层的节点数量
            size = len(queue)

            # 处理当前层的所有节点
            for _ in range(size):
                current = queue.popleft()

                # 将左右子节点加入队列（如果存在）
                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            # 每处理完一层，深度加 1
            depth += 1

        return depth

# 测试代码
if __name__ == "__main__":
    # 构建测试二叉树:
    #     3
    #    / \
    #   9  20
    #     /  \
    #    15   7
    root = TreeNode(3)
    root.left = TreeNode(9)
    root.right = TreeNode(20)
    root.right.left = TreeNode(15)
    root.right.right = TreeNode(7)

    solution = Solution()
    print("递归方法计算的最大深度:", solution.maxDepth1(root))
    print("迭代方法计算的最大深度:", solution.maxDepth2(root))

    # 测试空树
    empty_root = None
    print("空树的最大深度:", solution.maxDepth1(empty_root))

    # 测试单节点树
```

```
    single_root = TreeNode(1)
    print("单节点树的最大深度:", solution.maxDepth1(single_root))

==========================================

文件: LeetCode105_ConstructBinaryTreeFromPreorderAndInorderTraversal.cpp
==========================================
// LeetCode 105. 从前序与中序遍历序列构造二叉树
// 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
// 题目大意: 给定两个整数数组 preorder 和 inorder ，其中 preorder 是二叉树的先序遍历，inorder 是同一棵树的中序遍历，
// 请构造二叉树并返回其根节点。

// 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: 递归构建二叉树
     * 思路:
     * 1. 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，将中序遍历分为左子树和右子树
     * 3. 递归构建左子树和右子树
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(n) - 存储哈希表和递归调用栈
     */
    TreeNode* buildTree1(int* preorder, int preorderSize, int* inorder, int inorderSize) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要使用哈希表来快速查找根节点位置
        // 并递归构建左右子树
        return nullptr;
    }

    /**
```

```
     * 方法 2: 使用迭代器优化的递归方法
     * 思路: 使用一个全局索引跟踪前序遍历的当前位置
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(n) - 存储哈希表和递归调用栈
     */
    TreeNode* buildTree2(int* preorder, int preorderSize, int* inorder, int inorderSize) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要使用递归方式构建树
        return nullptr;
    }
};
```

==========

文件: LeetCode105_ConstructBinaryTreeFromPreorderAndInorderTraversal.java

==========

```
package class036;

import java.util.*;

// LeetCode 105. 从前序与中序遍历序列构造二叉树
// 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
// 题目大意: 给定两个整数数组 preorder 和 inorder ，其中 preorder 是二叉树的先序遍历，inorder 是同一棵树的中序遍历，
// 请构造二叉树并返回其根节点。

public class LeetCode105_ConstructBinaryTreeFromPreorderAndInorderTraversal {

    // 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: 递归构建二叉树
 * 思路:
 * 1. 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，将中序遍历分为左子树和右子树
 * 3. 递归构建左子树和右子树
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储哈希表和递归调用栈
 */
public static TreeNode buildTree1(int[] preorder, int[] inorder) {
    // 创建中序遍历值到索引的映射，用于快速查找根节点位置
    Map<Integer, Integer> inorderMap = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        inorderMap.put(inorder[i], i);
    }

    return buildTreeHelper(preorder, 0, preorder.length - 1,
                           inorder, 0, inorder.length - 1, inorderMap);
}

/**
 * 递归构建二叉树的辅助函数
 * @param preorder 前序遍历数组
 * @param preStart 前序遍历的起始索引
 * @param preEnd 前序遍历的结束索引
 * @param inorder 中序遍历数组
 * @param inStart 中序遍历的起始索引
 * @param inEnd 中序遍历的结束索引
 * @param inorderMap 中序遍历值到索引的映射
 * @return 构建的子树根节点
 */
private static TreeNode buildTreeHelper(int[] preorder, int preStart, int preEnd,
                                        int[] inorder, int inStart, int inEnd,
                                        Map<Integer, Integer> inorderMap) {
    // 递归终止条件
    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 rootIndex = inorderMap.get(rootVal);

    // 计算左子树的节点数量
    int leftSubtreeSize = rootIndex - inStart;

    // 递归构建左子树
    root.left = buildTreeHelper(preorder, preStart + 1, preStart + leftSubtreeSize,
                                inorder, inStart, rootIndex - 1, inorderMap);

    // 递归构建右子树
    root.right = buildTreeHelper(preorder, preStart + leftSubtreeSize + 1, preEnd,
                                 inorder, rootIndex + 1, inEnd, inorderMap);

    return root;
}

/**
 * 方法 2: 使用迭代器优化的递归方法
 * 思路: 使用一个全局索引跟踪前序遍历的当前位置
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储哈希表和递归调用栈
 */
private static int preorderIndex;

public static TreeNode buildTree2(int[] preorder, int[] inorder) {
    // 重置索引
    preorderIndex = 0;

    // 创建中序遍历值到索引的映射，用于快速查找根节点位置
    Map<Integer, Integer> inorderMap = new HashMap<>();
    for (int i = 0; i < inorder.length; i++) {
        inorderMap.put(inorder[i], i);
    }

    return buildTreeHelper2(preorder, inorder, 0, inorder.length - 1, inorderMap);
}

/**
 * 使用迭代器优化的递归构建二叉树的辅助函数
 * @param preorder 前序遍历数组
 * @param inorder 中序遍历数组
 * @param inStart 中序遍历的起始索引
```

```
     * @param inEnd 中序遍历的结束索引
     * @param inorderMap 中序遍历值到索引的映射
     * @return 构建的子树根节点
     */
    private static TreeNode buildTreeHelper2(int[] preorder, int[] inorder,
                                             int inStart, int inEnd,
                                             Map<Integer, Integer> inorderMap) {
        // 递归终止条件
        if (inStart > inEnd) {
            return null;
        }

        // 前序遍历的当前元素是根节点
        int rootVal = preorder[preorderIndex++];
        TreeNode root = new TreeNode(rootVal);

        // 在中序遍历中找到根节点的位置
        int rootIndex = inorderMap.get(rootVal);

        // 递归构建左子树
        root.left = buildTreeHelper2(preorder, inorder, inStart, rootIndex - 1, inorderMap);

        // 递归构建右子树
        root.right = buildTreeHelper2(preorder, inorder, rootIndex + 1, inEnd, inorderMap);

        return root;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
        int[] preorder1 = {3, 9, 20, 15, 7};
        int[] inorder1 = {9, 3, 15, 20, 7};
        TreeNode root1 = buildTree1(preorder1, inorder1);
        System.out.println("方法 1 构建的树根节点值: " + root1.val);

        TreeNode root2 = buildTree2(preorder1, inorder1);
        System.out.println("方法 2 构建的树根节点值: " + root2.val);
    }
}
```

==========

文件: LeetCode105_ConstructBinaryTreeFromPreorderAndInorderTraversal.py

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

# LeetCode 105. 从前序与中序遍历序列构造二叉树
# 题目链接: https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/
# 题目大意: 给定两个整数数组 preorder 和 inorder ，其中 preorder 是二叉树的先序遍历，inorder 是同一棵树的中序遍历，
# 请构造二叉树并返回其根节点。

# 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 buildTree1(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        """
        方法 1: 递归构建二叉树
        思路:
        1. 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，将中序遍历分为左子树和右子树
        3. 递归构建左子树和右子树
        时间复杂度: O(n) - n 是节点数量，每个节点访问一次
        空间复杂度: O(n) - 存储哈希表和递归调用栈
        """
        # 创建中序遍历值到索引的映射，用于快速查找根节点位置
        inorder_map = {val: i for i, val in enumerate(inorder)}

        def build_tree_helper(pre_start: int, pre_end: int,
                              in_start: int, in_end: int) -> Optional[TreeNode]:
            # 递归终止条件
            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 = inorder_map[root_val]

        # 计算左子树的节点数量
        left_subtree_size = root_index - in_start

        # 递归构建左子树
        root.left = build_tree_helper(pre_start + 1, pre_start + left_subtree_size,
                                      in_start, root_index - 1)

        # 递归构建右子树
        root.right = build_tree_helper(pre_start + left_subtree_size + 1, pre_end,
                                       root_index + 1, in_end)

        return root

    return build_tree_helper(0, len(preorder) - 1, 0, len(inorder) - 1)

def buildTree2(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
    """
    方法 2: 使用迭代器优化的递归方法
    思路: 使用一个全局索引跟踪前序遍历的当前位置
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(n) - 存储哈希表和递归调用栈
    """
    # 创建中序遍历值到索引的映射，用于快速查找根节点位置
    inorder_map = {val: i for i, val in enumerate(inorder)}

    # 使用迭代器跟踪前序遍历的当前位置
    self.preorder_index = 0

    def build_tree_helper2(in_start: int, in_end: int) -> Optional[TreeNode]:
        # 递归终止条件
        if in_start > in_end:
            return None

        # 前序遍历的当前元素是根节点
        root_val = preorder[self.preorder_index]
        self.preorder_index += 1
        root = TreeNode(root_val)

        # 在中序遍历中找到根节点的位置
        root_index = inorder_map[root_val]
```

```
            # 递归构建左子树
            root.left = build_tree_helper2(in_start, root_index - 1)

            # 递归构建右子树
            root.right = build_tree_helper2(root_index + 1, in_end)

            return root

        return build_tree_helper2(0, len(inorder) - 1)

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
    preorder1 = [3, 9, 20, 15, 7]
    inorder1 = [9, 3, 15, 20, 7]

    solution = Solution()
    root1 = solution.buildTree1(preorder1, inorder1)
    print("方法 1 构建的树根节点值:", root1.val if root1 else None)

    root2 = solution.buildTree2(preorder1, inorder1)
    print("方法 2 构建的树根节点值:", root2.val if root2 else None)

==========================================

文件: LeetCode107_BinaryTreeLevelOrderTraversalII.cpp
==========================================
// LeetCode 107. 二叉树的层序遍历 II
// 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
// 题目大意: 给你二叉树的根节点 root ，返回其节点值 自底向上的层序遍历 。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

#include <iostream>
#include <vector>
#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: 先正常层序遍历，再反转结果
     * 思路:
     * 1. 使用队列进行正常的层序遍历，从上到下收集每层节点值
     * 2. 遍历完成后，将结果列表反转，得到自底向上的遍历结果
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次，反转操作是 O(L)，L 为层数
     * 空间复杂度: O(n) - 存储队列和结果
     */
    vector<vector<int>> levelOrderBottom1(TreeNode* root) {
        vector<vector<int>> ans;
        if (root) {
            queue<TreeNode*> q;
            q.push(root);

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

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

                    // 将子节点加入队列，供下一层处理
                    if (cur->left) q.push(cur->left);
                    if (cur->right) q.push(cur->right);
                }

                // 将当前层的结果添加到最终答案中
                ans.push_back(level);
            }

            // 反转结果，得到自底向上的遍历
            reverse(ans.begin(), ans.end());
```

```
    }
    return ans;
}

/**
 * 方法2: 使用栈存储中间结果
 * 思路:
 * 1. 使用队列进行正常的层序遍历
 * 2. 使用栈存储每层的结果
 * 3. 遍历完成后，从栈中弹出结果，得到自底向上的遍历结果
 * 时间复杂度: O(n) - n是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储队列、栈和结果
 */
vector<vector<int>> levelOrderBottom2(TreeNode* root) {
    vector<vector<int>> ans;
    if (root) {
        queue<TreeNode*> q;
        stack<vector<int>> stk;
        q.push(root);

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

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

                // 将子节点加入队列，供下一层处理
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }

            // 将当前层的结果压入栈中
            stk.push(level);
        }

        // 从栈中弹出结果，得到自底向上的遍历
        while (!stk.empty()) {
            ans.push_back(stk.top());
            stk.pop();
```

```
        }
    }
    return ans;
}

/**
 * 方法 3: 在遍历过程中直接在列表开头插入
 * 思路:
 * 1. 使用队列进行正常的层序遍历
 * 2. 每层遍历完成后，将结果插入到结果列表的开头
 * 3. 这样最终结果就是自底向上的遍历
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储队列和结果
 * 注意: 在列表开头插入元素的时间复杂度是 O(L)，L 为当前列表长度，总体时间复杂度仍为 O(n)
 */
vector<vector<int>> levelOrderBottom3(TreeNode* root) {
    vector<vector<int>> ans;
    if (root) {
        queue<TreeNode*> q;
        q.push(root);

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

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

                // 将子节点加入队列，供下一层处理
                if (cur->left) q.push(cur->left);
                if (cur->right) q.push(cur->right);
            }

            // 将当前层的结果插入到结果列表的开头
            ans.insert(ans.begin(), level);
        }
    }
    return ans;
}
};
```

```
// 辅助函数：打印二维向量
void printVector(const vector<vector<int>>& vec) {
    for (const auto& v : vec) {
        cout << "[";
        for (size_t i = 0; i < v.size(); ++i) {
            cout << v[i];
            if (i < v.size() - 1) cout << ", ";
        }
        cout << "]" << endl;
    }
}

// 测试代码
int main() {
    // 测试用例 1: [3,9,20,null,null,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);

    Solution solution;
    cout << "方法 1 结果:" << endl;
    printVector(solution.levelOrderBottom1(root1));

    cout << "方法 2 结果:" << endl;
    printVector(solution.levelOrderBottom2(root1));

    cout << "方法 3 结果:" << endl;
    printVector(solution.levelOrderBottom3(root1));

    // 测试用例 2: [1]
    TreeNode* root2 = new TreeNode(1);
    cout << "单节点树结果:" << endl;
    printVector(solution.levelOrderBottom1(root2));

    // 释放内存
    delete root1->right->right;
    delete root1->right->left;
    delete root1->right;
    delete root1->left;
    delete root1;
```

```
    delete root2;

    return 0;
}
```

==========

文件: LeetCode107_BinaryTreeLevelOrderTraversalII.java

==========

```
package class036;

import java.util.*;

// LeetCode 107. 二叉树的层序遍历 II
// 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
// 题目大意: 给你二叉树的根节点 root ，返回其节点值 自底向上的层序遍历 。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

public class LeetCode107_BinaryTreeLevelOrderTraversalII {

    // 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: 先正常层序遍历，再反转结果
     * 思路:
     * 1. 使用队列进行正常的层序遍历，从上到下收集每层节点值
     * 2. 遍历完成后，将结果列表反转，得到自底向上的遍历结果
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次，反转操作是 O(L)，L 为层数
     * 空间复杂度: O(n) - 存储队列和结果
     */
    public static List<List<Integer>> levelOrderBottom1(TreeNode root) {
```

```
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

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

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

                // 将子节点加入队列，供下一层处理
                if (cur.left != null) {
                    queue.offer(cur.left);
                }
                if (cur.right != null) {
                    queue.offer(cur.right);
                }
            }

            // 将当前层的结果添加到最终答案中
            ans.add(level);
        }

        // 反转结果，得到自底向上的遍历
        Collections.reverse(ans);
    }
    return ans;
}

/**
 * 方法 2: 使用栈存储中间结果
 * 思路:
 * 1. 使用队列进行正常的层序遍历
 * 2. 使用栈存储每层的结果
 * 3. 遍历完成后，从栈中弹出结果，得到自底向上的遍历结果
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(n) - 存储队列、栈和结果
 */
public static List<List<Integer>> levelOrderBottom2(TreeNode root) {
```

```
    List<List<Integer>> ans = new ArrayList<>();
    if (root != null) {
        Queue<TreeNode> queue = new LinkedList<>();
        Stack<List<Integer>> stack = new Stack<>();
        queue.offer(root);

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

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

                // 将子节点加入队列，供下一层处理
                if (cur.left != null) {
                    queue.offer(cur.left);
                }
                if (cur.right != null) {
                    queue.offer(cur.right);
                }
            }

            // 将当前层的结果压入栈中
            stack.push(level);
        }

        // 从栈中弹出结果，得到自底向上的遍历
        while (!stack.isEmpty()) {
            ans.add(stack.pop());
        }
    }
    return ans;
}

/**
 * 方法 3: 在遍历过程中直接在列表开头插入
 * 思路:
 * 1. 使用队列进行正常的层序遍历
 * 2. 每层遍历完成后，将结果插入到结果列表的开头
 * 3. 这样最终结果就是自底向上的遍历
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
```

```
     * 空间复杂度: O(n) - 存储队列和结果
     * 注意: 在列表开头插入元素的时间复杂度是 O(L)，L 为当前列表长度，总体时间复杂度仍为 O(n)
     */
    public static List<List<Integer>> levelOrderBottom3(TreeNode root) {
        List<List<Integer>> ans = new ArrayList<>();
        if (root != null) {
            Queue<TreeNode> queue = new LinkedList<>();
            queue.offer(root);

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

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

                    // 将子节点加入队列，供下一层处理
                    if (cur.left != null) {
                        queue.offer(cur.left);
                    }
                    if (cur.right != null) {
                        queue.offer(cur.right);
                    }
                }

                // 将当前层的结果插入到结果列表的开头
                ans.add(0, level);
            }
        }
        return ans;
    }

    // 测试方法
    public static void main(String[] args) {
        // 测试用例 1: [3,9,20,null,null,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);
```

```
        System.out.println("方法 1 结果:");
        System.out.println(levelOrderBottom1(root1));

        System.out.println("方法 2 结果:");
        System.out.println(levelOrderBottom2(root1));

        System.out.println("方法 3 结果:");
        System.out.println(levelOrderBottom3(root1));

        // 测试用例 2: [1]
        TreeNode root2 = new TreeNode(1);
        System.out.println("单节点树结果:");
        System.out.println(levelOrderBottom1(root2));

        // 测试用例 3: []
        TreeNode root3 = null;
        System.out.println("空树结果:");
        System.out.println(levelOrderBottom1(root3));
    }
}
```

==========

文件: LeetCode107_BinaryTreeLevelOrderTraversalII.py

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

# LeetCode 107. 二叉树的层序遍历 II
# 题目链接: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
# 题目大意: 给你二叉树的根节点 root ，返回其节点值 自底向上的层序遍历 。（即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历）

# 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 levelOrderBottom1(self, root: Optional[TreeNode]) -> List[List[int]]:
        """
```

```
    方法 1: 先正常层序遍历，再反转结果
    思路:
    1. 使用队列进行正常的层序遍历，从上到下收集每层节点值
    2. 遍历完成后，将结果列表反转，得到自底向上的遍历结果
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次，反转操作是 O(L)，L 为层数
    空间复杂度: O(n) - 存储队列和结果
    """
    ans = []
    if root:
        queue = deque([root])

        while queue:
            size = len(queue)
            level = []

            # 处理当前层的所有节点
            for _ in range(size):
                cur = queue.popleft()
                level.append(cur.val)

                # 将子节点加入队列，供下一层处理
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)

            # 将当前层的结果添加到最终答案中
            ans.append(level)

        # 反转结果，得到自底向上的遍历
        ans.reverse()

    return ans

def levelOrderBottom2(self, root: Optional[TreeNode]) -> List[List[int]]:
    """
    方法 2: 使用栈存储中间结果
    思路:
    1. 使用队列进行正常的层序遍历
    2. 使用栈存储每层的结果
    3. 遍历完成后，从栈中弹出结果，得到自底向上的遍历结果
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(n) - 存储队列、栈和结果
```

```
    """
    ans = []
    if root:
        queue = deque([root])
        stack = []

        while queue:
            size = len(queue)
            level = []

            # 处理当前层的所有节点
            for _ in range(size):
                cur = queue.popleft()
                level.append(cur.val)

                # 将子节点加入队列，供下一层处理
                if cur.left:
                    queue.append(cur.left)
                if cur.right:
                    queue.append(cur.right)

            # 将当前层的结果压入栈中
            stack.append(level)

        # 从栈中弹出结果，得到自底向上的遍历
        while stack:
            ans.append(stack.pop())

    return ans

def levelOrderBottom3(self, root: Optional[TreeNode]) -> List[List[int]]:
    """
    方法 3: 在遍历过程中直接在列表开头插入
    思路:
    1. 使用队列进行正常的层序遍历
    2. 每层遍历完成后，将结果插入到结果列表的开头
    3. 这样最终结果就是自底向上的遍历
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(n) - 存储队列和结果
    注意: 在列表开头插入元素的时间复杂度是 O(L)，L 为当前列表长度，总体时间复杂度仍为 O(n)
    """
    ans = []
    if root:
```

```
            queue = deque([root])

            while queue:
                size = len(queue)
                level = []

                # 处理当前层的所有节点
                for _ in range(size):
                    cur = queue.popleft()
                    level.append(cur.val)

                    # 将子节点加入队列，供下一层处理
                    if cur.left:
                        queue.append(cur.left)
                    if cur.right:
                        queue.append(cur.right)

                # 将当前层的结果插入到结果列表的开头
                ans.insert(0, level)

        return ans

# 测试代码
if __name__ == "__main__":
    # 测试用例 1: [3,9,20,null,null,15,7]
    root1 = TreeNode(3)
    root1.left = TreeNode(9)
    root1.right = TreeNode(20)
    root1.right.left = TreeNode(15)
    root1.right.right = TreeNode(7)

    solution = Solution()
    print("方法 1 结果:")
    print(solution.levelOrderBottom1(root1))

    print("方法 2 结果:")
    print(solution.levelOrderBottom2(root1))

    print("方法 3 结果:")
    print(solution.levelOrderBottom3(root1))

    # 测试用例 2: [1]
    root2 = TreeNode(1)
```

```
    print("单节点树结果:")
    print(solution.levelOrderBottom1(root2))

    # 测试用例 3: []
    root3 = None
    print("空树结果:")
    print(solution.levelOrderBottom1(root3))

# ==========================================

文件: LeetCode110_BalancedBinaryTree.java
==========================================
package class036;

import java.util.*;

/**
 * LeetCode 110. 平衡二叉树
 * 题目链接: https://leetcode.cn/problems/balanced-binary-tree/
 * 题目描述: 给定一个二叉树，判断它是否是高度平衡的二叉树。
 * 高度平衡二叉树定义: 一个二叉树每个节点的左右两个子树的高度差的绝对值不超过 1。
 *
 * 核心算法思想:
 * 1. 自顶向下递归: 计算每个节点的高度并检查平衡性
 * 2. 自底向上递归: 在计算高度的同时检查平衡性，避免重复计算
 * 3. 优化递归: 使用返回值同时传递高度和平衡信息
 *
 * 时间复杂度分析:
 * - 方法 1(自顶向下): O(NlogN) - 最坏情况 O(N²)
 * - 方法 2(自底向上): O(N) - 每个节点访问一次
 * - 方法 3(优化递归): O(N) - 每个节点访问一次
 *
 * 空间复杂度分析:
 * - 所有方法: O(H) - H 为树的高度，递归调用栈深度
 *
 * 相关题目:
 * 1. LeetCode 104. 二叉树的最大深度 - 基础高度计算
 * 2. LeetCode 111. 二叉树的最小深度 - 深度计算变种
 * 3. 剑指 Offer 55 - II. 平衡二叉树 - 相同题目
 *
 * 工程化考量:
 * 1. 提前终止: 发现不平衡立即返回，避免不必要的计算
 * 2. 返回值设计: 使用特殊值或对象传递多个信息
```

```
 * 3. 边界处理: 空树、单节点树等特殊情况
 */
public class LeetCode110_BalancedBinaryTree {

    // 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: 自顶向下递归 - 基础实现
     * 思路: 对于每个节点，计算左右子树高度并检查平衡性
     * 时间复杂度: O(NlogN) - 平衡树情况，最坏 O(N²)
     * 空间复杂度: O(H) - H 为树的高度
     *
     * 缺点: 存在重复计算高度的问题
     */
    public static boolean isBalanced1(TreeNode root) {
        if (root == null) {
            return true;
        }

        // 检查当前节点是否平衡
        int leftHeight = getHeight(root.left);
        int rightHeight = getHeight(root.right);

        if (Math.abs(leftHeight - rightHeight) > 1) {
            return false;
        }

        // 递归检查左右子树
        return isBalanced1(root.left) && isBalanced1(root.right);
    }
```

```
/**
 * 计算二叉树的高度
 */
private static int getHeight(TreeNode node) {
    if (node == null) {
        return 0;
    }
    return Math.max(getHeight(node.left), getHeight(node.right)) + 1;
}

/**
 * 方法 2: 自底向上递归 - 最优解法
 * 思路: 在计算高度的同时检查平衡性，避免重复计算
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(H) - H 为树的高度
 *
 * 核心思想:
 * 1. 使用-1 表示不平衡状态
 * 2. 如果子树不平衡，立即返回-1
 * 3. 否则返回正常的高度值
 */
public static boolean isBalanced2(TreeNode root) {
    return checkHeight(root) != -1;
}

private static int checkHeight(TreeNode node) {
    if (node == null) {
        return 0;
    }

    // 检查左子树
    int leftHeight = checkHeight(node.left);
    if (leftHeight == -1) {
        return -1; // 左子树不平衡
    }

    // 检查右子树
    int rightHeight = checkHeight(node.right);
    if (rightHeight == -1) {
        return -1; // 右子树不平衡
    }

    // 检查当前节点是否平衡
```

```
    if (Math.abs(leftHeight - rightHeight) > 1) {
        return -1; // 当前节点不平衡
    }

    // 返回当前节点的高度
    return Math.max(leftHeight, rightHeight) + 1;
}

/**
 * 方法 3: 使用返回值对象 - 更清晰的实现
 * 思路: 创建返回值对象同时包含高度和平衡信息
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(H) - H 为树的高度
 */
public static boolean isBalanced3(TreeNode root) {
    Result result = checkBalanced(root);
    return result.isBalanced;
}

/**
 * 返回值对象，包含高度和平衡信息
 */
private static class Result {
    boolean isBalanced;
    int height;

    Result(boolean isBalanced, int height) {
        this.isBalanced = isBalanced;
        this.height = height;
    }
}

private static Result checkBalanced(TreeNode node) {
    if (node == null) {
        return new Result(true, 0);
    }

    // 检查左子树
    Result leftResult = checkBalanced(node.left);
    if (!leftResult.isBalanced) {
        return new Result(false, 0);
    }
```

```
    // 检查右子树
    Result rightResult = checkBalanced(node.right);
    if (!rightResult.isBalanced) {
        return new Result(false, 0);
    }

    // 检查当前节点是否平衡
    if (Math.abs(leftResult.height - rightResult.height) > 1) {
        return new Result(false, 0);
    }

    // 返回当前节点的结果
    int height = Math.max(leftResult.height, rightResult.height) + 1;
    return new Result(true, height);
}

/**
 * 辅助方法: 根据数组构建测试树
 */
public static TreeNode buildTree(Integer[] arr) {
    if (arr == null || arr.length == 0 || arr[0] == null) {
        return null;
    }

    TreeNode root = new TreeNode(arr[0]);
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < arr.length) {
        TreeNode current = queue.poll();

        if (i < arr.length && arr[i] != null) {
            current.left = new TreeNode(arr[i]);
            queue.offer(current.left);
        }
        i++;

        if (i < arr.length && arr[i] != null) {
            current.right = new TreeNode(arr[i]);
            queue.offer(current.right);
        }
        i++;
```

```
        }

        return root;
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== LeetCode 110 测试 ==========");

        // 测试用例 1: 平衡二叉树 [3,9,20,null,null,15,7]
        Integer[] arr1 = {3, 9, 20, null, null, 15, 7};
        TreeNode root1 = buildTree(arr1);

        System.out.println("测试用例 1 - 平衡二叉树:");
        System.out.println("方法 1 结果: " + isBalanced1(root1));
        System.out.println("方法 2 结果: " + isBalanced2(root1));
        System.out.println("方法 3 结果: " + isBalanced3(root1));

        // 测试用例 2: 不平衡二叉树 [1,2,2,3,3,null,null,4,4]
        Integer[] arr2 = {1, 2, 2, 3, 3, null, null, 4, 4};
        TreeNode root2 = buildTree(arr2);

        System.out.println("\n测试用例 2 - 不平衡二叉树:");
        System.out.println("方法 1 结果: " + isBalanced1(root2));
        System.out.println("方法 2 结果: " + isBalanced2(root2));
        System.out.println("方法 3 结果: " + isBalanced3(root2));

        // 测试用例 3: 单节点树
        TreeNode root3 = new TreeNode(1);
        System.out.println("\n测试用例 3 - 单节点树:");
        System.out.println("方法 1 结果: " + isBalanced1(root3));

        // 测试用例 4: 空树
        TreeNode root4 = null;
        System.out.println("\n测试用例 4 - 空树:");
        System.out.println("方法 1 结果: " + isBalanced1(root4));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（自顶向下）: 逻辑简单但效率低，存在重复计算");
        System.out.println("2. 方法 2（自底向上）: 最优解法 O(N)，推荐使用");
```

```
        System.out.println("3. 方法 3（返回值对象）: 代码更清晰，性能与方法 2 相同");
    }
}

/*
Python 实现:

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

class Solution:
    # 方法 2: 自底向上递归（最优解）
    def isBalanced(self, root: TreeNode) -> bool:
        def check_height(node):
            if not node:
                return 0

            left_height = check_height(node.left)
            if left_height == -1:
                return -1

            right_height = check_height(node.right)
            if right_height == -1:
                return -1

            if abs(left_height - right_height) > 1:
                return -1

            return max(left_height, right_height) + 1

        return check_height(root) != -1

C++实现:

#include <iostream>
#include <algorithm>
#include <cmath>
using namespace std;

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 isBalanced(TreeNode* root) {
        return checkHeight(root) != -1;
    }

private:
    int checkHeight(TreeNode* node) {
        if (!node) return 0;

        int leftHeight = checkHeight(node->left);
        if (leftHeight == -1) return -1;

        int rightHeight = checkHeight(node->right);
        if (rightHeight == -1) return -1;

        if (abs(leftHeight - rightHeight) > 1) return -1;

        return max(leftHeight, rightHeight) + 1;
    }
};
*/
```

==========

文件: LeetCode116_PopulatingNextRightPointersInEachNode.java

==========

```
package class036;

import java.util.*;

/**
 * LeetCode 116. 填充每个节点的下一个右侧节点指针
 * 题目链接: https://leetcode.cn/problems/populating-next-right-pointers-in-each-node/
 * 题目描述: 给定一个完美二叉树，其所有叶子节点都在同一层，每个父节点都有两个子节点。
```

```
 * 填充它的每个 next 指针，让这个指针指向其下一个右侧节点。如果找不到下一个右侧节点，则将 next 指针设置为 null。
 * 初始状态下，所有 next 指针都被设置为 null。
 *
 * 完美二叉树定义：所有叶子节点都在同一层，每个非叶子节点都有两个子节点
 *
 * 核心算法思想：
 * 1. 层序遍历(BFS)：使用队列进行层序遍历，在每层中连接相邻节点的 next 指针
 * 2. 利用已建立的 next 指针：利用上一层的 next 指针来连接当前层的节点，避免使用队列
 *
 * 时间复杂度分析：
 * - 方法 1(层序遍历)：O(N)，每个节点访问一次
 * - 方法 2(利用 next 指针)：O(N)，每个节点访问一次
 *
 * 空间复杂度分析：
 * - 方法 1(层序遍历)：O(W)，W 为树的最大宽度，最坏情况为 O(N/2)≈O(N)
 * - 方法 2(利用 next 指针)：O(1)，只使用常数级别的额外空间
 *
 * 相关题目：
 * 1. LeetCode 117. 填充每个节点的下一个右侧节点指针 II - 非完美二叉树版本
 * 2. LeetCode 199. 二叉树的右视图 - 类似的分层处理思想
 * 3. LeetCode 102. 二叉树的层序遍历 - 基础层序遍历
 *
 * 工程化考量：
 * 1. 线程安全：多线程环境下需要同步机制
 * 2. 内存管理：对于大树需要注意内存使用
 * 3. 异常处理：处理空指针和边界情况
 */
public class LeetCode116_PopulatingNextRightPointersInEachNode {

    // 完美二叉树节点定义
    static class Node {
        public int val;
        public Node left;
        public Node right;
        public Node next;

        public Node() {}

        public Node(int _val) {
            val = _val;
        }
```

```
    public Node(int _val, Node _left, Node _right, Node _next) {
        val = _val;
        left = _left;
        right = _right;
        next = _next;
    }

    @Override
    public String toString() {
        return "Node{" + val + "}";
    }
}

/**
 * 方法 1: 层序遍历法 - 使用队列进行 BFS 遍历
 * 思路: 使用队列进行层序遍历，在每层遍历时连接相邻节点的 next 指针
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度，最坏情况为 O(N/2)≈O(N)
 *
 * 优点:
 * - 逻辑清晰，易于理解
 * - 适用于各种二叉树结构
 * 缺点:
 * - 需要额外的队列空间
 * - 对于完美二叉树有更优的空间复杂度解法
 *
 * 关键步骤:
 * 1. 使用队列存储当前层的所有节点
 * 2. 遍历当前层，连接每个节点到下一个节点
 * 3. 将下一层的节点加入队列
 */
public static Node connect1(Node root) {
    if (root == null) {
        return null;
    }

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

    while (!queue.isEmpty()) {
        int size = queue.size();
        Node prev = null; // 前一个节点
```

```
            for (int i = 0; i < size; i++) {
                Node current = queue.poll();

                // 连接前一个节点到当前节点
                if (prev != null) {
                    prev.next = current;
                }
                prev = current;

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

            // 每层最后一个节点的 next 为 null（默认值）
        }

        return root;
    }

    /**
     * 方法 2: 利用已建立的 next 指针 - 最优解法
     * 思路: 利用完美二叉树的特性，通过上一层的 next 指针来连接当前层的节点
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(1) - 只使用常数级别的额外空间
     *
     * 核心思想:
     * 1. 使用两个指针: leftmost 指向每层的最左节点，current 用于遍历当前层
     * 2. 对于每个节点，连接其左子节点到右子节点
     * 3. 连接相邻子树的节点（通过 next 指针）
     *
     * 优点:
     * - 空间复杂度最优
     * - 充分利用完美二叉树的特性
     * 缺点:
     * - 只适用于完美二叉树
     *
     * 关键步骤:
     * 1. leftmost 指针始终指向当前层的最左节点
```

```
     * 2. current 指针遍历当前层，连接下一层的节点
     * 3. 当 leftmost 没有左子节点时，说明到达叶子层，遍历结束
     */
    public static Node connect2(Node root) {
        if (root == null) {
            return null;
        }

        // leftmost 指针始终指向当前层的最左节点
        Node leftmost = root;

        // 当 leftmost 还有下一层时继续循环
        while (leftmost.left != null) {
            // current 指针用于遍历当前层
            Node current = leftmost;

            while (current != null) {
                // 连接当前节点的左子节点到右子节点
                current.left.next = current.right;

                // 如果当前节点有下一个右侧节点，连接当前节点的右子节点到下一个节点的左子节点
                if (current.next != null) {
                    current.right.next = current.next.left;
                }

                // 移动到当前层的下一个节点
                current = current.next;
            }

            // 移动到下一层的最左节点
            leftmost = leftmost.left;
        }

        return root;
    }

    /**
     * 方法 3: 递归解法 - 深度优先遍历
     * 思路: 使用递归连接相邻节点，先处理同一父节点的子节点，再处理跨父节点的子节点
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(logN) - 递归调用栈的深度，对于完美二叉树为树的高度
     *
     * 优点:
```

```
     * - 代码简洁，递归思想清晰
     * - 不需要显式使用队列
     * 缺点:
     * - 递归调用栈可能较深
     * - 对于大树可能栈溢出
     *
     * 关键递归关系:
     * 1. 连接左子节点到右子节点
     * 2. 连接右子节点到下一个左子节点（如果存在）
     * 3. 递归处理左右子树
     */
    public static Node connect3(Node root) {
        if (root == null) {
            return null;
        }

        connectNodes(root.left, root.right);
        return root;
    }

    private static void connectNodes(Node node1, Node node2) {
        if (node1 == null || node2 == null) {
            return;
        }

        // 连接两个节点
        node1.next = node2;

        // 递归连接同一父节点的子节点
        connectNodes(node1.left, node1.right);
        connectNodes(node2.left, node2.right);

        // 递归连接跨父节点的子节点
        connectNodes(node1.right, node2.left);
    }

    /**
     * 辅助方法: 根据数组构建完美二叉树
     * 数组格式: 层序遍历，完美二叉树
     * 例如: [1,2,3,4,5,6,7] 构建 3 层的完美二叉树
     */
    public static Node buildPerfectBinaryTree(Integer[] arr) {
        if (arr == null || arr.length == 0 || arr[0] == null) {
```

```
        return null;
    }

    Node root = new Node(arr[0]);
    Queue<Node> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < arr.length) {
        Node current = queue.poll();

        // 构建左子节点
        if (i < arr.length && arr[i] != null) {
            current.left = new Node(arr[i]);
            queue.offer(current.left);
        }
        i++;

        // 构建右子节点
        if (i < arr.length && arr[i] != null) {
            current.right = new Node(arr[i]);
            queue.offer(current.right);
        }
        i++;
    }

    return root;
}

/**
 * 辅助方法: 打印树的层序遍历结果（包含 next 指针信息）
 * 用于验证连接结果
 */
public static void printTreeWithNext(Node root) {
    if (root == null) {
        System.out.println("Empty Tree");
        return;
    }

    Node levelStart = root;

    while (levelStart != null) {
        Node current = levelStart;
```

```
        StringBuilder level = new StringBuilder();
        boolean first = true;

        while (current != null) {
            if (!first) {
                level.append(" -> ");
            }
            level.append(current.val);
            first = false;
            current = current.next;
        }

        System.out.println(level.toString());
        levelStart = levelStart.left; // 完美二叉树，最左节点一定有左子节点
    }
}

/**
 * 测试方法: 包含多种测试用例
 * 覆盖边界情况和典型场景
 */
public static void main(String[] args) {
    System.out.println("========== LeetCode 116 测试 ==========");

    // 测试用例 1: 3 层完美二叉树 [1,2,3,4,5,6,7]
    System.out.println("\n 测试用例 1: 3 层完美二叉树");
    Integer[] arr1 = {1, 2, 3, 4, 5, 6, 7};
    Node root1 = buildPerfectBinaryTree(arr1);

    System.out.println("方法 1 结果:");
    printTreeWithNext(connect1(buildPerfectBinaryTree(arr1)));

    System.out.println("\n 方法 2 结果:");
    printTreeWithNext(connect2(buildPerfectBinaryTree(arr1)));

    System.out.println("\n 方法 3 结果:");
    printTreeWithNext(connect3(buildPerfectBinaryTree(arr1)));

    // 测试用例 2: 单节点树
    System.out.println("\n 测试用例 2: 单节点树");
    Integer[] arr2 = {1};
    Node root2 = buildPerfectBinaryTree(arr2);
```

```
        System.out.println("方法 2 结果:");
        printTreeWithNext(connect2(root2));

        // 测试用例 3: 空树
        System.out.println("\n 测试用例 3: 空树");
        Node root3 = null;
        System.out.println("方法 2 结果: " + connect2(root3));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，但空间复杂度较高");
        System.out.println("2. 方法 2（利用 next 指针）: 空间复杂度最优，只适用于完美二叉树");
        System.out.println("3. 方法 3（递归）: 代码简洁，但递归深度可能较大");
        System.out.println("推荐: 对于完美二叉树使用方法 2，对于一般二叉树使用 LeetCode 117 的解法
");
    }
}

/*
Python 实现:

class Node:
    def __init__(self, val=0, left=None, right=None, next=None):
        self.val = val
        self.left = left
        self.right = right
        self.next = next

class Solution:
    # 方法 1: 层序遍历法
    def connect1(self, root: 'Node') -> 'Node':
        if not root:
            return None

        from collections import deque
        queue = deque([root])

        while queue:
            size = len(queue)
            prev = None

            for i in range(size):
                current = queue.popleft()
```

```
            if prev:
                prev.next = current
            prev = current

            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)

    return root

# 方法 2: 利用 next 指针（最优解）
def connect2(self, root: 'Node') -> 'Node':
    if not root:
        return None

    leftmost = root

    while leftmost.left:
        current = leftmost

        while current:
            # 连接同一父节点的子节点
            current.left.next = current.right

            # 连接相邻父节点的子节点
            if current.next:
                current.right.next = current.next.left

            current = current.next

        leftmost = leftmost.left

    return root

# 方法 3: 递归解法
def connect3(self, root: 'Node') -> 'Node':
    if not root:
        return None

    self._connect_nodes(root.left, root.right)
    return root
```

```
    def _connect_nodes(self, node1: 'Node', node2: 'Node'):
        if not node1 or not node2:
            return

        node1.next = node2

        self._connect_nodes(node1.left, node1.right)
        self._connect_nodes(node2.left, node2.right)
        self._connect_nodes(node1.right, node2.left)

# 测试代码
if __name__ == "__main__":
    # 构建测试树
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    root.right.left = Node(6)
    root.right.right = Node(7)

    solution = Solution()

    print("方法 1 结果:")
    result1 = solution.connect1(root)
    # 打印结果...

    print("方法 2 结果:")
    result2 = solution.connect2(root)
    # 打印结果...
```

C++实现:

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

class Node {
public:
    int val;
    Node* left;
```

```
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}
    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
    Node(int _val, Node* _left, Node* _right, Node* _next)
        : val(_val), left(_left), right(_right), next(_next) {}
};

class Solution {
public:
    // 方法 1: 层序遍历法
    Node* connect1(Node* root) {
        if (!root) return nullptr;

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

        while (!q.empty()) {
            int size = q.size();
            Node* prev = nullptr;

            for (int i = 0; i < size; ++i) {
                Node* current = q.front();
                q.pop();

                if (prev) {
                    prev->next = current;
                }
                prev = current;

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

        return root;
    }

    // 方法 2: 利用 next 指针（最优解）
    Node* connect2(Node* root) {
        if (!root) return nullptr;
```

```
        Node* leftmost = root;

        while (leftmost->left) {
            Node* current = leftmost;

            while (current) {
                // 连接同一父节点的子节点
                current->left->next = current->right;

                // 连接相邻父节点的子节点
                if (current->next) {
                    current->right->next = current->next->left;
                }

                current = current->next;
            }

            leftmost = leftmost->left;
        }

        return root;
    }

    // 方法 3: 递归解法
    Node* connect3(Node* root) {
        if (!root) return nullptr;

        connectNodes(root->left, root->right);
        return root;
    }

private:
    void connectNodes(Node* node1, Node* node2) {
        if (!node1 || !node2) return;

        node1->next = node2;

        connectNodes(node1->left, node1->right);
        connectNodes(node2->left, node2->right);
        connectNodes(node1->right, node2->left);
    }
};
```

```
// 测试代码
int main() {
    // 构建测试树
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    root->right->left = new Node(6);
    root->right->right = new Node(7);

    Solution solution;

    cout << "方法 2 结果:" << endl;
    Node* result = solution.connect2(root);
    // 打印结果...

    // 释放内存
    delete root->right->right;
    delete root->right->left;
    delete root->right;
    delete root->left;
    delete root;

    return 0;
}
*/
```

==========

文件: LeetCode117_PopulatingNextRightPointersInEachNodeII.java

==========

```
package class036;

import java.util.*;

/**
 * LeetCode 117. 填充每个节点的下一个右侧节点指针 II
 * 题目链接: https://leetcode.cn/problems/populating-next-right-pointers-in-each-node-ii/
 * 题目描述: 给定一个二叉树，填充它的每个 next 指针，让这个指针指向其下一个右侧节点。
 * 如果找不到下一个右侧节点，则将 next 指针设置为 null。
 * 初始状态下，所有 next 指针都被设置为 null。
 *
```

```
 * 与 LeetCode 116 的区别：本题的二叉树不一定是完美二叉树，可能缺少某些节点
 *
 * 核心算法思想：
 * 1. 层序遍历(BFS)：使用队列进行层序遍历，在每层中连接相邻节点的 next 指针
 * 2. 利用虚拟头节点：使用虚拟头节点来简化每层的连接逻辑
 * 3. 空间优化遍历：利用已建立的 next 指针进行层序遍历，避免使用队列
 *
 * 时间复杂度分析：
 * - 所有方法：O(N)，每个节点访问一次
 *
 * 空间复杂度分析：
 * - 方法 1(层序遍历)：O(W)，W 为树的最大宽度
 * - 方法 2(虚拟头节点)：O(1)，只使用常数级别的额外空间
 * - 方法 3(空间优化)：O(1)，只使用常数级别的额外空间
 *
 * 相关题目：
 * 1. LeetCode 116. 填充每个节点的下一个右侧节点指针 - 完美二叉树版本
 * 2. LeetCode 199. 二叉树的右视图 - 类似的分层处理思想
 * 3. LeetCode 102. 二叉树的层序遍历 - 基础层序遍历
 *
 * 工程化考量：
 * 1. 通用性：适用于任意二叉树结构
 * 2. 鲁棒性：处理各种边界情况（空树、单节点、斜树等）
 * 3. 性能优化：对于大数据量需要考虑空间效率
 */
public class LeetCode117_PopulatingNextRightPointersInEachNodeII {

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

        public Node() {}

        public Node(int _val) {
            val = _val;
        }

        public Node(int _val, Node _left, Node _right, Node _next) {
            val = _val;
            left = _left;
```

```
        right = _right;
        next = _next;
    }

    @Override
    public String toString() {
        return "Node{" + val + "}";
    }
}

/**
 * 方法 1: 层序遍历法 - 使用队列进行 BFS 遍历
 * 思路: 使用队列进行层序遍历，在每层遍历时连接相邻节点的 next 指针
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度，最坏情况为 O(N)
 *
 * 优点:
 * - 逻辑清晰，易于理解和实现
 * - 适用于各种二叉树结构
 * 缺点:
 * - 需要额外的队列空间
 * - 空间复杂度不是最优
 *
 * 关键步骤:
 * 1. 使用队列存储当前层的所有节点
 * 2. 遍历当前层，连接每个节点到下一个节点
 * 3. 将下一层的节点加入队列
 */
public static Node connect1(Node root) {
    if (root == null) {
        return null;
    }

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

    while (!queue.isEmpty()) {
        int size = queue.size();
        Node prev = null; // 前一个节点

        for (int i = 0; i < size; i++) {
            Node current = queue.poll();
```

```
            // 连接前一个节点到当前节点
            if (prev != null) {
                prev.next = current;
            }
            prev = current;

            // 将子节点加入队列（注意：可能缺少某些子节点）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        // 每层最后一个节点的 next 为 null（默认值）
    }

    return root;
}

/**
 * 方法 2: 使用虚拟头节点 - 最优解法（空间复杂度 O(1)）
 * 思路: 使用虚拟头节点来简化每层的连接逻辑，利用已建立的 next 指针遍历下一层
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
 *
 * 核心思想:
 * 1. 使用虚拟头节点 dummy 来记录下一层的起始位置
 * 2. 使用 current 指针遍历当前层，同时构建下一层的连接
 * 3. 使用 tail 指针来跟踪下一层的最后一个节点
 *
 * 优点:
 * - 空间复杂度最优
 * - 适用于任意二叉树结构
 * 缺点:
 * - 逻辑相对复杂，需要仔细处理指针连接
 *
 * 关键步骤:
 * 1. 外层循环: 从根节点开始，逐层向下处理
 * 2. 内层循环: 遍历当前层的所有节点，连接下一层的节点
 * 3. 虚拟头节点: 简化下一层起始位置的处理
 */
```

```
public static Node connect2(Node root) {
    if (root == null) {
        return null;
    }

    Node current = root;

    while (current != null) {
        // 虚拟头节点，用于记录下一层的起始位置
        Node dummy = new Node(0);
        // tail 指针用于跟踪下一层的最后一个节点
        Node tail = dummy;

        // 遍历当前层的所有节点
        while (current != null) {
            // 处理左子节点
            if (current.left != null) {
                tail.next = current.left;
                tail = tail.next;
            }

            // 处理右子节点
            if (current.right != null) {
                tail.next = current.right;
                tail = tail.next;
            }

            // 移动到当前层的下一个节点
            current = current.next;
        }

        // 移动到下一层的第一个节点（通过虚拟头节点的 next）
        current = dummy.next;
    }

    return root;
}

/**
 * 方法 3: 空间优化遍历 - 另一种 O(1)空间复杂度的解法
 * 思路: 使用两个指针分别跟踪当前层和下一层，避免使用虚拟头节点
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(1) - 只使用常数级别的额外空间
```

```
     *
     * 核心思想:
     * 1. head 指针指向下一层的起始节点
     * 2. prev 指针跟踪下一层的最后一个节点
     * 3. current 指针遍历当前层
     *
     * 优点:
     * - 空间复杂度最优
     * - 不使用虚拟节点，更直观
     * 缺点:
     * - 需要处理头节点的特殊情况
     *
     * 关键步骤:
     * 1. 找到下一层的第一个非空节点作为 head
     * 2. 使用 prev 连接下一层的节点
     * 3. 当前层遍历完成后，移动到下一层
     */
    public static Node connect3(Node root) {
        if (root == null) {
            return null;
        }

        Node current = root;

        while (current != null) {
            Node head = null; // 下一层的头节点
            Node prev = null; // 下一层的前一个节点

            // 遍历当前层，构建下一层的连接
            while (current != null) {
                // 处理左子节点
                if (current.left != null) {
                    if (head == null) {
                        head = current.left; // 设置下一层的头节点
                    }
                    if (prev != null) {
                        prev.next = current.left;
                    }
                    prev = current.left;
                }

                // 处理右子节点
                if (current.right != null) {
```

```
                if (head == null) {
                    head = current.right; // 设置下一层的头节点
                }
                if (prev != null) {
                    prev.next = current.right;
                }
                prev = current.right;
            }

            // 移动到当前层的下一个节点
            current = current.next;
        }

        // 移动到下一层
        current = head;
    }

    return root;
}

/**
 * 辅助方法: 根据数组构建任意二叉树（不一定是完美二叉树）
 * 数组格式: 层序遍历，null 表示空节点
 * 例如: [1,2,3,4,null,null,5] 构建非完美二叉树
 */
public static Node buildBinaryTree(Integer[] arr) {
    if (arr == null || arr.length == 0 || arr[0] == null) {
        return null;
    }

    Node root = new Node(arr[0]);
    Queue<Node> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < arr.length) {
        Node current = queue.poll();

        // 构建左子节点
        if (i < arr.length && arr[i] != null) {
            current.left = new Node(arr[i]);
            queue.offer(current.left);
        }
```

```
            i++;

            // 构建右子节点
            if (i < arr.length && arr[i] != null) {
                current.right = new Node(arr[i]);
                queue.offer(current.right);
            }
            i++;
        }

        return root;
    }

    /**
     * 辅助方法: 打印树的层序遍历结果（包含 next 指针信息）
     * 用于验证连接结果，支持非完美二叉树
     */
    public static void printTreeWithNext(Node root) {
        if (root == null) {
            System.out.println("Empty Tree");
            return;
        }

        // 找到每层的最左节点
        Node levelStart = root;

        while (levelStart != null) {
            Node current = levelStart;
            StringBuilder level = new StringBuilder();
            boolean first = true;

            // 找到下一层的最左节点
            Node nextLevelStart = null;
            Node nextLevelPrev = null;

            while (current != null) {
                if (!first) {
                    level.append(" -> ");
                }
                level.append(current.val);
                first = false;

                // 同时构建下一层的最左节点
```

```
            if (nextLevelStart == null) {
                if (current.left != null) {
                    nextLevelStart = current.left;
                    nextLevelPrev = nextLevelStart;
                } else if (current.right != null) {
                    nextLevelStart = current.right;
                    nextLevelPrev = nextLevelStart;
                }
            } else {
                if (current.left != null) {
                    nextLevelPrev.next = current.left;
                    nextLevelPrev = nextLevelPrev.next;
                }
                if (current.right != null) {
                    nextLevelPrev.next = current.right;
                    nextLevelPrev = nextLevelPrev.next;
                }
            }

            current = current.next;
        }

        System.out.println(level.toString());
        levelStart = nextLevelStart;
    }
}

/**
 * 测试方法: 包含多种测试用例
 * 覆盖各种边界情况和典型场景
 */
public static void main(String[] args) {
    System.out.println("========== LeetCode 117 测试 ==========");

    // 测试用例 1: 非完美二叉树 [1,2,3,4,null,null,5]
    System.out.println("\n 测试用例 1: 非完美二叉树");
    Integer[] arr1 = {1, 2, 3, 4, null, null, 5};
    Node root1 = buildBinaryTree(arr1);

    System.out.println("方法 1 结果:");
    printTreeWithNext(connect1(buildBinaryTree(arr1)));

    System.out.println("\n 方法 2 结果:");
```

```
        printTreeWithNext(connect2(buildBinaryTree(arr1)));

        System.out.println("\n 方法 3 结果:");
        printTreeWithNext(connect3(buildBinaryTree(arr1)));

        // 测试用例 2: 单节点树
        System.out.println("\n 测试用例 2: 单节点树");
        Integer[] arr2 = {1};
        Node root2 = buildBinaryTree(arr2);

        System.out.println("方法 2 结果:");
        printTreeWithNext(connect2(root2));

        // 测试用例 3: 斜树（只有左子树）
        System.out.println("\n 测试用例 3: 斜树");
        Integer[] arr3 = {1, 2, null, 3, null, 4, null};
        Node root3 = buildBinaryTree(arr3);

        System.out.println("方法 2 结果:");
        printTreeWithNext(connect2(root3));

        // 测试用例 4: 空树
        System.out.println("\n 测试用例 4: 空树");
        Node root4 = null;
        System.out.println("方法 2 结果: " + connect2(root4));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，逻辑清晰，但空间复杂度较高");
        System.out.println("2. 方法 2（虚拟头节点）: 空间复杂度最优，推荐使用");
        System.out.println("3. 方法 3（空间优化）: 空间复杂度最优，逻辑相对复杂");
        System.out.println("推荐: 对于一般二叉树使用方法 2，平衡代码可读性和空间效率");
    }
}

/*
Python 实现:

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

```
        self.next = next

class Solution:
    # 方法 1: 层序遍历法
    def connect1(self, root: 'Node') -> 'Node':
        if not root:
            return None

        from collections import deque
        queue = deque([root])

        while queue:
            size = len(queue)
            prev = None

            for i in range(size):
                current = queue.popleft()

                if prev:
                    prev.next = current
                prev = current

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

        return root

    # 方法 2: 使用虚拟头节点（最优解）
    def connect2(self, root: 'Node') -> 'Node':
        if not root:
            return None

        current = root

        while current:
            dummy = Node(0)  # 虚拟头节点
            tail = dummy

            # 遍历当前层
            while current:
                if current.left:
```

```
                    tail.next = current.left
                    tail = tail.next
                if current.right:
                    tail.next = current.right
                    tail = tail.next

                current = current.next

            # 移动到下一层
            current = dummy.next

        return root

# 测试代码
if __name__ == "__main__":
    # 构建测试树
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.right.right = Node(5)

    solution = Solution()
    result = solution.connect2(root)
```

C++实现:

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

class Node {
public:
    int val;
    Node* left;
    Node* right;
    Node* next;

    Node() : val(0), left(NULL), right(NULL), next(NULL) {}
    Node(int _val) : val(_val), left(NULL), right(NULL), next(NULL) {}
};

class Solution {
```

```
public:
    // 方法 2: 使用虚拟头节点（最优解）
    Node* connect(Node* root) {
        if (!root) return nullptr;

        Node* current = root;

        while (current) {
            Node* dummy = new Node(0);  // 虚拟头节点
            Node* tail = dummy;

            // 遍历当前层
            while (current) {
                if (current->left) {
                    tail->next = current->left;
                    tail = tail->next;
                }
                if (current->right) {
                    tail->next = current->right;
                    tail = tail->next;
                }

                current = current->next;
            }

            // 移动到下一层
            current = dummy->next;
            delete dummy;  // 释放虚拟头节点
        }

        return root;
    }
};

// 测试代码
int main() {
    // 构建测试树
    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->right->right = new Node(5);
```

```
    Solution solution;
    Node* result = solution.connect(root);

    // 释放内存...
    return 0;
}
*/
```

==========

文件: LeetCode199_BinaryTreeRightSideView.java

==========

```
package class036;

import java.util.*;

/**
 * LeetCode 199. 二叉树的右视图
 * 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。
 *
 * 核心算法思想:
 * 1. 层序遍历(BFS): 使用队列进行层序遍历，记录每层的最右节点
 * 2. 深度优先遍历(DFS): 使用递归进行深度优先遍历，优先访问右子树
 * 3. 反向层序遍历: 从右到左进行层序遍历，记录每层的第一个节点
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，其中 N 是二叉树中的节点数，每个节点只被访问一次
 *
 * 空间复杂度分析:
 * - 方法 1(层序遍历): O(W)，W 为树的最大宽度
 * - 方法 2(DFS 递归): O(H)，H 为树的高度，递归调用栈的深度
 * - 方法 3(反向层序): O(W)，W 为树的最大宽度
 *
 * 相关题目:
 * 1. LeetCode 102. 二叉树的层序遍历 - 基础层序遍历
 * 2. LeetCode 116/117. 填充下一个右侧节点指针 - 类似的分层处理
 * 3. LeetCode 637. 二叉树的层平均值 - 分层统计
 * 4. LeetCode 515. 在每个树行中找最大值 - 分层找极值
 * 5. 剑指 Offer 32 - III. 从上到下打印二叉树 III - 锯齿形层序遍历
 *
 * 工程化考量:
```

```
 * 1. 鲁棒性: 处理空树、单节点树、斜树等各种边界情况
 * 2. 性能优化: 对于大数据量选择空间复杂度更低的方法
 * 3. 可读性: 代码结构清晰，注释完整
 */
public class LeetCode199_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: 层序遍历法 - 记录每层的最右节点
     * 思路: 使用队列进行层序遍历，在每层遍历时记录最后一个节点（最右节点）
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(W) - W 为树的最大宽度，最坏情况为 O(N/2)≈O(N)
     *
     * 优点:
     * - 逻辑清晰，易于理解和实现
     * - 适用于各种二叉树结构
     * 缺点:
     * - 需要额外的队列空间
     *
     * 关键步骤:
     * 1. 使用队列存储当前层的所有节点
     * 2. 记录每层的节点数量
     * 3. 遍历当前层，记录最后一个节点
     */
    public static List<Integer> rightSideView1(TreeNode root) {
        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();
        TreeNode rightmost = null;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            rightmost = current; // 记录当前节点

            // 先左后右加入队列（保证同一层从左到右）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        // 当前层的最右节点加入结果
        if (rightmost != null) {
            result.add(rightmost.val);
        }
    }

    return result;
}

/**
 * 方法 2: 深度优先遍历(DFS) - 优先访问右子树
 * 思路: 使用递归进行 DFS，优先访问右子树，每层第一个访问到的节点就是右视图节点
 * 时间复杂度: O(N) - 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
 *
 * 核心思想:
 * 1. 优先访问右子树，再访问左子树
 * 2. 当深度等于结果列表大小时，说明是当前层第一个访问的节点（最右节点）
 * 3. 将该节点加入结果列表
 *
 * 优点:
 * - 空间复杂度较低（树的高度通常远小于节点数）
 * - 递归代码简洁
```

```
     * 缺点:
     * - 递归深度可能较大（斜树）
     * - 对于大树可能栈溢出
     */
    public static List<Integer> rightSideView2(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        dfs(root, 0, result);
        return result;
    }

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

        // 如果当前深度等于结果列表大小，说明是当前层第一个访问的节点
        if (depth == result.size()) {
            result.add(node.val);
        }

        // 优先访问右子树（保证先看到右边的节点）
        dfs(node.right, depth + 1, result);
        // 再访问左子树
        dfs(node.left, depth + 1, result);
    }

    /**
     * 方法 3: 反向层序遍历 - 从右到左遍历，记录每层第一个节点
     * 思路: 层序遍历时从右到左处理节点，每层的第一个节点就是右视图节点
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(W) - W 为树的最大宽度
     *
     * 优点:
     * - 逻辑直观，易于理解
     * - 避免递归的栈溢出风险
     * 缺点:
     * - 需要额外的队列空间
     *
     * 关键优化:
     * - 从右到左处理节点，可以立即确定右视图节点
     */
    public static List<Integer> rightSideView3(TreeNode root) {
        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();
        // 每层的第一个节点（从右开始）就是右视图节点
        result.add(queue.peek().val);

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();

            // 先右后左加入队列（保证下一层从右开始）
            if (current.right != null) {
                queue.offer(current.right);
            }
            if (current.left != null) {
                queue.offer(current.left);
            }
        }
    }

    return result;
}

/**
 * 方法4: 优化的BFS - 只记录必要的节点信息
 * 思路: 使用数组实现队列，减少对象创建开销
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W为树的最大宽度
 *
 * 工程化优化:
 * 1. 使用数组代替LinkedList减少内存分配
 * 2. 预分配队列大小，避免动态扩容
 * 3. 减少对象创建，提升性能
 */
public static List<Integer> rightSideView4(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) {
        return result;
```

```
    }

    // 预分配队列大小
    TreeNode[] queue = new TreeNode[1000];
    int l = 0, r = 0;
    queue[r++] = root;

    while (l < r) {
        int size = r - l;
        TreeNode rightmost = null;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue[l++];
            rightmost = current;

            if (current.left != null) {
                queue[r++] = current.left;
            }
            if (current.right != null) {
                queue[r++] = current.right;
            }
        }

        if (rightmost != null) {
            result.add(rightmost.val);
        }
    }

    return result;
}

/**
 * 辅助方法: 根据数组构建测试树
 * 数组格式: 层序遍历，null 表示空节点
 */
public static TreeNode buildTree(Integer[] arr) {
    if (arr == null || arr.length == 0 || arr[0] == null) {
        return null;
    }

    TreeNode root = new TreeNode(arr[0]);
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
```

```
        int i = 1;

        while (!queue.isEmpty() && i < arr.length) {
            TreeNode current = queue.poll();

            if (i < arr.length && arr[i] != null) {
                current.left = new TreeNode(arr[i]);
                queue.offer(current.left);
            }
            i++;

            if (i < arr.length && arr[i] != null) {
                current.right = new TreeNode(arr[i]);
                queue.offer(current.right);
            }
            i++;
        }

        return root;
    }

    /**
     * 测试方法: 包含多种测试用例
     * 覆盖各种边界情况和典型场景
     */
    public static void main(String[] args) {
        System.out.println("========== LeetCode 199 测试 ==========");

        // 测试用例 1: 标准二叉树 [1,2,3,null,5,null,4]
        System.out.println("\n 测试用例 1: 标准二叉树");
        Integer[] arr1 = {1, 2, 3, null, 5, null, 4};
        TreeNode root1 = buildTree(arr1);

        System.out.println("方法 1 结果: " + rightSideView1(root1));
        System.out.println("方法 2 结果: " + rightSideView2(root1));
        System.out.println("方法 3 结果: " + rightSideView3(root1));
        System.out.println("方法 4 结果: " + rightSideView4(root1));

        // 测试用例 2: 只有左子树 [1,2,null,3,null,4]
        System.out.println("\n 测试用例 2: 斜树（左）");
        Integer[] arr2 = {1, 2, null, 3, null, 4, null};
        TreeNode root2 = buildTree(arr2);
        System.out.println("方法 1 结果: " + rightSideView1(root2));
```

```
        // 测试用例 3: 只有右子树 [1,null,2,null,3,null,4]
        System.out.println("\n 测试用例 3: 斜树（右）");
        Integer[] arr3 = {1, null, 2, null, 3, null, 4};
        TreeNode root3 = buildTree(arr3);
        System.out.println("方法 1 结果: " + rightSideView1(root3));

        // 测试用例 4: 单节点树
        System.out.println("\n 测试用例 4: 单节点树");
        TreeNode root4 = new TreeNode(1);
        System.out.println("方法 1 结果: " + rightSideView1(root4));

        // 测试用例 5: 空树
        System.out.println("\n 测试用例 5: 空树");
        TreeNode root5 = null;
        System.out.println("方法 1 结果: " + rightSideView1(root5));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，逻辑清晰");
        System.out.println("2. 方法 2（DFS 递归）: 空间复杂度优，适合平衡树");
        System.out.println("3. 方法 3（反向层序）: 逻辑直观，避免递归风险");
        System.out.println("4. 方法 4（优化 BFS）: 性能最优，适合大数据量");
        System.out.println("推荐: 根据具体场景选择合适的方法");
    }
}

/*
Python 实现:

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: 层序遍历法
    def rightSideView1(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        from collections import deque
```

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

        while queue:
            size = len(queue)
            rightmost = None

            for i in range(size):
                current = queue.popleft()
                rightmost = current

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            if rightmost:
                result.append(rightmost.val)

        return result

    # 方法2: DFS递归
    def rightSideView2(self, root: TreeNode) -> List[int]:
        result = []

        def dfs(node, depth):
            if not node:
                return

            if depth == len(result):
                result.append(node.val)

            dfs(node.right, depth + 1)
            dfs(node.left, depth + 1)

        dfs(root, 0)
        return result

# 测试代码
if __name__ == "__main__":
    # 构建测试树
    root = TreeNode(1)
    root.left = TreeNode(2)
```

```
    root.right = TreeNode(3)
    root.left.right = TreeNode(5)
    root.right.right = TreeNode(4)

    solution = Solution()
    print(solution.rightSideView1(root))
```

C++实现:

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

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: 层序遍历法
    vector<int> rightSideView(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

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

        while (!q.empty()) {
            int size = q.size();
            TreeNode* rightmost = nullptr;

            for (int i = 0; i < size; ++i) {
                TreeNode* current = q.front();
                q.pop();
                rightmost = current;

                if (current->left) q.push(current->left);
```

```
                if (current->right) q.push(current->right);
            }

            if (rightmost) {
                result.push_back(rightmost->val);
            }
        }

        return result;
    }
};

// 测试代码
int main() {
    TreeNode* root = new TreeNode(1);
    root->left = new TreeNode(2);
    root->right = new TreeNode(3);
    root->left->right = new TreeNode(5);
    root->right->right = new TreeNode(4);

    Solution solution;
    vector<int> result = solution.rightSideView(root);

    for (int val : result) {
        cout << val << " ";
    }

    // 释放内存...
    return 0;
}
*/

==========================================

文件: LeetCode222_CountCompleteTreeNodes.cpp
==========================================
// LeetCode 222. 完全二叉树的节点个数
// 题目链接: https://leetcode.cn/problems/count-complete-tree-nodes/
// 题目大意: 给你一棵完全二叉树的根节点 root ，求出该树的节点个数。
// 完全二叉树的定义：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，
// 并且最下面一层的节点都集中在该层最左边的若干位置。

// 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: 普通递归方法
     * 思路: 直接递归计算左子树和右子树的节点数，然后加 1
     * 时间复杂度: O(n) - n 是节点数量
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    int countNodes1(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }

        return 1 + countNodes1(root->left) + countNodes1(root->right);
    }

    /**
     * 方法 2: 利用完全二叉树性质优化的方法
     * 思路:
     * 1. 计算树的高度
     * 2. 利用完全二叉树的性质，如果左右子树高度相等，说明左子树是满二叉树
     * 3. 如果左右子树高度不等，说明右子树是满二叉树
     * 4. 满二叉树的节点数可以直接计算，不需要遍历
     * 时间复杂度: O(log²n) - 每次递归减少一半节点，每次计算高度需要 O(logn)
     * 空间复杂度: O(logn) - 递归调用栈的深度
     */
    int countNodes2(TreeNode* root) {
        if (root == nullptr) {
            return 0;
        }

        // 计算左子树的高度
        int leftHeight = getHeight(root->left);
        // 计算右子树的高度
```

```
        int rightHeight = getHeight(root->right);

        if (leftHeight == rightHeight) {
            // 左右子树高度相等，说明左子树是满二叉树
            // 左子树节点数 = 2^leftHeight - 1
            // 总节点数 = 左子树节点数 + 根节点 + 右子树节点数
            return (1 << leftHeight) + countNodes2(root->right);
        } else {
            // 左右子树高度不等，说明右子树是满二叉树
            // 右子树节点数 = 2^rightHeight - 1
            // 总节点数 = 右子树节点数 + 根节点 + 左子树节点数
            return (1 << rightHeight) + countNodes2(root->left);
        }
    }

private:
    /**
     * 计算树的高度
     * @param root 树的根节点
     * @return 树的高度
     */
    int getHeight(TreeNode* root) {
        int height = 0;
        while (root != nullptr) {
            height++;
            root = root->left;
        }
        return height;
    }

public:
    /**
     * 方法3: 二分查找法
     * 思路:
     * 1. 计算完全二叉树的高度
     * 2. 最后一层的节点数在1到2^h之间
     * 3. 使用二分查找确定最后一层有多少个节点
     * 时间复杂度: O(log²n) - 二分查找需要O(logn)，每次检查需要O(logn)
     * 空间复杂度: O(1) - 只使用常数额外空间
     */
    int countNodes3(TreeNode* root) {
        if (root == nullptr) {
            return 0;
```

```
        }

        // 计算树的高度
        int height = getHeight(root);

        // 如果只有根节点
        if (height == 1) {
            return 1;
        }

        // 计算除最后一层外的节点数
        int upperCount = (1 << (height - 1)) - 1;

        // 二分查找最后一层的节点数
        int left = 1, right = 1 << (height - 1);
        int lastLevelCount = 0;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nodeExists(root, mid, height)) {
                lastLevelCount = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return upperCount + lastLevelCount;
    }

private:
    /**
     * 检查最后一层第 index 个节点是否存在
     * @param root 树的根节点
     * @param index 节点在最后一层的索引(从 1 开始)
     * @param height 树的高度
     * @return 节点是否存在
     */
    bool nodeExists(TreeNode* root, int index, int height) {
        int left = 1, right = 1 << (height - 1);

        for (int i = 0; i < height - 1; i++) {
            int mid = left + (right - left) / 2;
```

```
            if (index <= mid) {
                root = root->left;
                right = mid;
            } else {
                root = root->right;
                left = mid + 1;
            }
        }

        return root != nullptr;
    }
};
```

==========

文件: LeetCode222_CountCompleteTreeNodes.java

==========

```
package class036;

// LeetCode 222. 完全二叉树的节点个数
// 题目链接: https://leetcode.cn/problems/count-complete-tree-nodes/
// 题目大意: 给你一棵完全二叉树的根节点 root ，求出该树的节点个数。
// 完全二叉树的定义：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，
// 并且最下面一层的节点都集中在该层最左边的若干位置。

public class LeetCode222_CountCompleteTreeNodes {

    // 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: 普通递归方法
```

```
     * 思路: 直接递归计算左子树和右子树的节点数，然后加 1
     * 时间复杂度: O(n) - n 是节点数量
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    public static int countNodes1(TreeNode root) {
        if (root == null) {
            return 0;
        }

        return 1 + countNodes1(root.left) + countNodes1(root.right);
    }

    /**
     * 方法 2: 利用完全二叉树性质优化的方法
     * 思路:
     * 1. 计算树的高度
     * 2. 利用完全二叉树的性质，如果左右子树高度相等，说明左子树是满二叉树
     * 3. 如果左右子树高度不等，说明右子树是满二叉树
     * 4. 满二叉树的节点数可以直接计算，不需要遍历
     * 时间复杂度: O(log²n) - 每次递归减少一半节点，每次计算高度需要 O(logn)
     * 空间复杂度: O(logn) - 递归调用栈的深度
     */
    public static int countNodes2(TreeNode root) {
        if (root == null) {
            return 0;
        }

        // 计算左子树的高度
        int leftHeight = getHeight(root.left);
        // 计算右子树的高度
        int rightHeight = getHeight(root.right);

        if (leftHeight == rightHeight) {
            // 左右子树高度相等，说明左子树是满二叉树
            // 左子树节点数 = 2^leftHeight - 1
            // 总节点数 = 左子树节点数 + 根节点 + 右子树节点数
            return (1 << leftHeight) + countNodes2(root.right);
        } else {
            // 左右子树高度不等，说明右子树是满二叉树
            // 右子树节点数 = 2^rightHeight - 1
            // 总节点数 = 右子树节点数 + 根节点 + 左子树节点数
            return (1 << rightHeight) + countNodes2(root.left);
        }
```

```
}

/**
 * 计算树的高度
 * @param root 树的根节点
 * @return 树的高度
 */
private static int getHeight(TreeNode root) {
    int height = 0;
    while (root != null) {
        height++;
        root = root.left;
    }
    return height;
}

/**
 * 方法 3: 二分查找法
 * 思路:
 * 1. 计算完全二叉树的高度
 * 2. 最后一层的节点数在 1 到 2^h 之间
 * 3. 使用二分查找确定最后一层有多少个节点
 * 时间复杂度: O(log²n) - 二分查找需要 O(logn)，每次检查需要 O(logn)
 * 空间复杂度: O(1) - 只使用常数额外空间
 */
public static int countNodes3(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 计算树的高度
    int height = getHeight(root);

    // 如果只有根节点
    if (height == 1) {
        return 1;
    }

    // 计算除最后一层外的节点数
    int upperCount = (1 << (height - 1)) - 1;

    // 二分查找最后一层的节点数
    int left = 1, right = 1 << (height - 1);
```

```
    int lastLevelCount = 0;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nodeExists(root, mid, height)) {
            lastLevelCount = mid;
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }

    return upperCount + lastLevelCount;
}

/**
 * 检查最后一层第 index 个节点是否存在
 * @param root 树的根节点
 * @param index 节点在最后一层的索引(从 1 开始)
 * @param height 树的高度
 * @return 节点是否存在
 */
private static boolean nodeExists(TreeNode root, int index, int height) {
    int left = 1, right = 1 << (height - 1);

    for (int i = 0; i < height - 1; i++) {
        int mid = left + (right - left) / 2;
        if (index <= mid) {
            root = root.left;
            right = mid;
        } else {
            root = root.right;
            left = mid + 1;
        }
    }

    return root != null;
}

// 测试方法
public static void main(String[] args) {
    // 构建测试完全二叉树:
    //       1
```

```
        //      / \
        //     2   3
        //    / \  /
        //   4   5 6
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(4);
        root.left.right = new TreeNode(5);
        root.right.left = new TreeNode(6);

        System.out.println("方法 1 结果: " + countNodes1(root));
        System.out.println("方法 2 结果: " + countNodes2(root));
        System.out.println("方法 3 结果: " + countNodes3(root));
    }
}
```

==========

文件: LeetCode222_CountCompleteTreeNodes.py

==========

```
from typing import Optional

# LeetCode 222. 完全二叉树的节点个数
# 题目链接: https://leetcode.cn/problems/count-complete-tree-nodes/
# 题目大意: 给你一棵完全二叉树的根节点 root ，求出该树的节点个数。
# 完全二叉树的定义：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，
# 并且最下面一层的节点都集中在该层最左边的若干位置。

# 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 countNodes1(self, root: Optional[TreeNode]) -> int:
        """
        方法 1: 普通递归方法
        思路: 直接递归计算左子树和右子树的节点数，然后加 1
        时间复杂度: O(n) - n 是节点数量
        空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
```

```
    """
    if not root:
        return 0

    return 1 + self.countNodes1(root.left) + self.countNodes1(root.right)

def countNodes2(self, root: Optional[TreeNode]) -> int:
    """
    方法 2: 利用完全二叉树性质优化的方法
    思路:
    1. 计算树的高度
    2. 利用完全二叉树的性质，如果左右子树高度相等，说明左子树是满二叉树
    3. 如果左右子树高度不等，说明右子树是满二叉树
    4. 满二叉树的节点数可以直接计算，不需要遍历
    时间复杂度: O(log²n) - 每次递归减少一半节点，每次计算高度需要 O(logn)
    空间复杂度: O(logn) - 递归调用栈的深度
    """
    if not root:
        return 0

    # 计算左子树的高度
    left_height = self.getHeight(root.left)
    # 计算右子树的高度
    right_height = self.getHeight(root.right)

    if left_height == right_height:
        # 左右子树高度相等，说明左子树是满二叉树
        # 左子树节点数 = 2^left_height - 1
        # 总节点数 = 左子树节点数 + 根节点 + 右子树节点数
        return (1 << left_height) + self.countNodes2(root.right)
    else:
        # 左右子树高度不等，说明右子树是满二叉树
        # 右子树节点数 = 2^right_height - 1
        # 总节点数 = 右子树节点数 + 根节点 + 左子树节点数
        return (1 << right_height) + self.countNodes2(root.left)

def getHeight(self, root: Optional[TreeNode]) -> int:
    """
    计算树的高度
    Args:
        root: 树的根节点
    Returns:
        树的高度
```

```
    """
    height = 0
    while root:
        height += 1
        root = root.left
    return height

def countNodes3(self, root: Optional[TreeNode]) -> int:
    """
    方法 3: 二分查找法
    思路:
    1. 计算完全二叉树的高度
    2. 最后一层的节点数在 1 到 2^h 之间
    3. 使用二分查找确定最后一层有多少个节点
    时间复杂度: O(log²n) - 二分查找需要 O(logn)，每次检查需要 O(logn)
    空间复杂度: O(1) - 只使用常数额外空间
    """
    if not root:
        return 0

    # 计算树的高度
    height = self.getHeight(root)

    # 如果只有根节点
    if height == 1:
        return 1

    # 计算除最后一层外的节点数
    upper_count = (1 << (height - 1)) - 1

    # 二分查找最后一层的节点数
    left, right = 1, 1 << (height - 1)
    last_level_count = 0

    while left <= right:
        mid = left + (right - left) // 2
        if self.nodeExists(root, mid, height):
            last_level_count = mid
            left = mid + 1
        else:
            right = mid - 1

    return upper_count + last_level_count
```

```
    def nodeExists(self, root: Optional[TreeNode], index: int, height: int) -> bool:
        """
        检查最后一层第 index 个节点是否存在
        Args:
            root: 树的根节点
            index: 节点在最后一层的索引(从 1 开始)
            height: 树的高度
        Returns:
            节点是否存在
        """
        left, right = 1, 1 << (height - 1)

        for _ in range(height - 1):
            mid = left + (right - left) // 2
            if index <= mid:
                if root:
                    root = root.left
                right = mid
            else:
                if root:
                    root = root.right
                left = mid + 1

        return root is not None

# 测试代码
if __name__ == "__main__":
    # 构建测试完全二叉树:
    #       1
    #      / \
    #     2   3
    #    / \  /
    #   4   5 6
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)
    root.right.left = TreeNode(6)

    solution = Solution()
    print("方法 1 结果:", solution.countNodes1(root))
```

```
    print("方法 2 结果:", solution.countNodes2(root))
    print("方法 3 结果:", solution.countNodes3(root))

==========================================

文件: LeetCode226_InvertBinaryTree.cpp
==========================================
// LeetCode 226. 翻转二叉树
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目大意: 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。

#include <iostream>
#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: 递归实现翻转二叉树
     * 思路:
     * 1. 如果当前节点为空，直接返回
     * 2. 递归翻转左子树
     * 3. 递归翻转右子树
     * 4. 交换左右子树
     * 5. 返回根节点
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    TreeNode* invertTree1(TreeNode* root) {
        // 递归终止条件
        if (root == nullptr) {
            return nullptr;
        }
```

```
    // 递归翻转左子树和右子树
    TreeNode* left = invertTree1(root->left);
    TreeNode* right = invertTree1(root->right);

    // 交换左右子树
    root->left = right;
    root->right = left;

    return root;
}

/**
 * 方法 2: 迭代实现翻转二叉树
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 对于每个节点，交换其左右子树
 * 3. 将左右子节点加入队列继续处理
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
TreeNode* invertTree2(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }

    // 使用队列进行层序遍历
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        // 取出队首节点
        TreeNode* current = q.front();
        q.pop();

        // 交换当前节点的左右子树
        TreeNode* temp = current->left;
        current->left = current->right;
        current->right = temp;

        // 将左右子节点加入队列（如果存在）
        if (current->left != nullptr) {
            q.push(current->left);
```

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

        return root;
    }
};

// 辅助函数：打印二叉树（前序遍历）
void printTree(TreeNode* root) {
    if (root != nullptr) {
        cout << root->val << " ";
        printTree(root->left);
        printTree(root->right);
    }
}

// 测试代码
int main() {
    // 构建测试二叉树:
    //     4
    //    / \
    //   2   7
    //  / \ / \
    // 1  3 6  9
    TreeNode* root = new TreeNode(4);
    root->left = new TreeNode(2);
    root->right = new TreeNode(7);
    root->left->left = new TreeNode(1);
    root->left->right = new TreeNode(3);
    root->right->left = new TreeNode(6);
    root->right->right = new TreeNode(9);

    cout << "翻转前: ";
    printTree(root);
    cout << endl;

    // 翻转二叉树
    Solution solution;
    TreeNode* invertedRoot = solution.invertTree1(root);
```

```
    cout << "翻转后: ";
    printTree(invertedRoot);
    cout << endl;

    // 释放内存
    // 注意：在实际应用中，应该实现完整的内存管理

    return 0;
}
```

==========

文件: LeetCode226_InvertBinaryTree.java

==========

```
package class036;

import java.util.*;

// LeetCode 226. 翻转二叉树
// 题目链接: https://leetcode.cn/problems/invert-binary-tree/
// 题目大意: 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。

public class LeetCode226_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;
        }
    }

    /**
     * 方法1: 递归实现翻转二叉树
     * 思路:
     * 1. 如果当前节点为空，直接返回
     * 2. 递归翻转左子树
```

```
     * 3. 递归翻转右子树
     * 4. 交换左右子树
     * 5. 返回根节点
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    public static TreeNode invertTree1(TreeNode root) {
        // 递归终止条件
        if (root == null) {
            return null;
        }

        // 递归翻转左子树和右子树
        TreeNode left = invertTree1(root.left);
        TreeNode right = invertTree1(root.right);

        // 交换左右子树
        root.left = right;
        root.right = left;

        return root;
    }

    /**
     * 方法 2: 迭代实现翻转二叉树
     * 思路:
     * 1. 使用队列进行层序遍历
     * 2. 对于每个节点，交换其左右子树
     * 3. 将左右子节点加入队列继续处理
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    public static TreeNode invertTree2(TreeNode root) {
        if (root == null) {
            return null;
        }

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

        while (!queue.isEmpty()) {
            // 取出队首节点
```

```
            TreeNode current = queue.poll();

            // 交换当前节点的左右子树
            TreeNode temp = current.left;
            current.left = current.right;
            current.right = temp;

            // 将左右子节点加入队列（如果存在）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        return root;
    }

    // 测试方法
    public static void main(String[] args) {
        // 构建测试二叉树:
        //     4
        //    / \
        //   2   7
        //  / \ / \
        // 1  3 6  9
        TreeNode root = new TreeNode(4);
        root.left = new TreeNode(2);
        root.right = new TreeNode(7);
        root.left.left = new TreeNode(1);
        root.left.right = new TreeNode(3);
        root.right.left = new TreeNode(6);
        root.right.right = new TreeNode(9);

        System.out.println("翻转前:");
        printTree(root);

        // 翻转二叉树
        TreeNode invertedRoot = invertTree1(root);

        System.out.println("翻转后:");
        printTree(invertedRoot);
```

```
    }

    // 辅助方法：打印二叉树（前序遍历）
    public static void printTree(TreeNode root) {
        if (root != null) {
            System.out.print(root.val + " ");
            printTree(root.left);
            printTree(root.right);
        }
    }
}
```

==========

文件: LeetCode226_InvertBinaryTree.py

==========

```
from typing import Optional

# LeetCode 226. 翻转二叉树
# 题目链接: https://leetcode.cn/problems/invert-binary-tree/
# 题目大意: 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。

# 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

# 辅助函数：打印二叉树（前序遍历）
def print_tree(root: Optional[TreeNode]) -> None:
    if root:
        print(root.val, end=" ")
        print_tree(root.left)
        print_tree(root.right)

class Solution:
    def invertTree1(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        """
        方法 1: 递归实现翻转二叉树
        思路:
        1. 如果当前节点为空，直接返回
        2. 递归翻转左子树
```

```
    3. 递归翻转右子树
    4. 交换左右子树
    5. 返回根节点
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
    """
    # 递归终止条件
    if not root:
        return None

    # 递归翻转左子树和右子树
    left = self.invertTree1(root.left)
    right = self.invertTree1(root.right)

    # 交换左右子树
    root.left = right
    root.right = left

    return root

def invertTree2(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
    """
    方法 2: 迭代实现翻转二叉树
    思路:
    1. 使用队列进行层序遍历
    2. 对于每个节点，交换其左右子树
    3. 将左右子节点加入队列继续处理
    时间复杂度: O(n) - n 是节点数量，每个节点访问一次
    空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
    """
    if not root:
        return None

    from collections import deque
    # 使用队列进行层序遍历
    queue = deque([root])

    while queue:
        # 取出队首节点
        current = queue.popleft()

        # 交换当前节点的左右子树
        current.left, current.right = current.right, current.left
```

```
            # 将左右子节点加入队列（如果存在）
            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)

        return root

# 测试代码
if __name__ == "__main__":
    # 构建测试二叉树:
    #     4
    #    / \
    #   2   7
    #  / \ / \
    # 1  3 6  9
    root = TreeNode(4)
    root.left = TreeNode(2)
    root.right = TreeNode(7)
    root.left.left = TreeNode(1)
    root.left.right = TreeNode(3)
    root.right.left = TreeNode(6)
    root.right.right = TreeNode(9)

    print("翻转前:")
    print_tree(root)

    # 翻转二叉树
    solution = Solution()
    inverted_root = solution.invertTree1(root)

    print("\n翻转后:")
    print_tree(inverted_root)

================================================

文件: LeetCode297_SerializeAndDeserializeBinaryTree.cpp
================================================
// LeetCode 297. 二叉树的序列化与反序列化
// 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
// 题目大意: 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，进而可以将转换后的数据存储在一个文件或内存缓
```

冲区中，
// 并且在需要的时候可以恢复成原来的数据结构或对象。

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

class Codec {
public:
    /**
     * 方法: 使用层序遍历进行序列化和反序列化
     * 思路:
     * 1. 序列化: 使用 BFS 层序遍历，将每个节点的值转换为字符串，空节点用"null"表示
     * 2. 反序列化: 将序列化的字符串解析为节点值列表，然后使用 BFS 方式重建树
     * 时间复杂度: O(n) - n 是树中节点的数量
     * 空间复杂度: O(n) - 存储序列化字符串和重建树所需的队列空间
     */

    /**
     * 序列化二叉树
     * @param root 二叉树的根节点
     * @return 序列化后的字符串
     */
    char* serialize(TreeNode* root) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要使用队列来进行层序遍历
        return 0;
    }

    /**
     * 反序列化二叉树
     * @param data 序列化后的字符串
     * @return 二叉树的根节点
     */
    TreeNode* deserialize(char* data) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要解析字符串并重建树
        return 0;
    }
```

```
};

==========================================

文件: LeetCode297_SerializeAndDeserializeBinaryTree.java
==========================================
package class036;

import java.util.*;

// LeetCode 297. 二叉树的序列化与反序列化
// 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
// 题目大意: 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，进而可以将转换后的数据存储在一个文件或内存缓冲区中，
// 并且在需要的时候可以恢复成原来的数据结构或对象。

public class LeetCode297_SerializeAndDeserializeBinaryTree {

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

    /**
     * 方法: 使用层序遍历进行序列化和反序列化
     * 思路:
     * 1. 序列化: 使用 BFS 层序遍历，将每个节点的值转换为字符串，空节点用"null"表示
     * 2. 反序列化: 将序列化的字符串解析为节点值列表，然后使用 BFS 方式重建树
     * 时间复杂度: O(n) - n 是树中节点的数量
     * 空间复杂度: O(n) - 存储序列化字符串和重建树所需的队列空间
     */

    /**
     * 序列化二叉树
     * @param root 二叉树的根节点
     * @return 序列化后的字符串
     */
    public static String serialize(TreeNode root) {
        if (root == null) {
```

```
        return "[]";
    }

    StringBuilder sb = new StringBuilder();
    sb.append("[");
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    boolean first = true;

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

        if (!first) {
            sb.append(",");
        }
        first = false;

        if (node == null) {
            sb.append("null");
        } else {
            sb.append(node.val);
            queue.offer(node.left);
            queue.offer(node.right);
        }
    }

    sb.append("]");
    return sb.toString();
}

/**
 * 反序列化二叉树
 * @param data 序列化后的字符串
 * @return 二叉树的根节点
 */
public static TreeNode deserialize(String data) {
    if (data == null || data.equals("[]")) {
        return null;
    }

    // 解析字符串为节点值列表
    String[] values = data.substring(1, data.length() - 1).split(",");
    if (values.length == 0 || values[0].equals("null")) {
```

```
        return null;
    }

    TreeNode root = new TreeNode(Integer.parseInt(values[0]));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < values.length) {
        TreeNode node = queue.poll();

        // 处理左子节点
        if (!values[i].equals("null")) {
            node.left = new TreeNode(Integer.parseInt(values[i]));
            queue.offer(node.left);
        }
        i++;

        // 处理右子节点
        if (i < values.length && !values[i].equals("null")) {
            node.right = new TreeNode(Integer.parseInt(values[i]));
            queue.offer(node.right);
        }
        i++;
    }

    return root;
}

// 测试方法
public static void main(String[] args) {
    // 构建测试二叉树:
    //       1
    //      / \
    //     2   3
    //        / \
    //       4   5
    TreeNode root = new TreeNode(1);
    root.left = new TreeNode(2);
    root.right = new TreeNode(3);
    root.right.left = new TreeNode(4);
    root.right.right = new TreeNode(5);
```

```
        // 序列化
        String serialized = serialize(root);
        System.out.println("序列化结果: " + serialized);

        // 反序列化
        TreeNode deserialized = deserialize(serialized);
        System.out.println("反序列化结果: " + serialize(deserialized));
    }
}
```

```
==========================================
```

文件: LeetCode297_SerializeAndDeserializeBinaryTree.py

```
==========================================
from collections import deque
from typing import Optional

# LeetCode 297. 二叉树的序列化与反序列化
# 题目链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
# 题目大意: 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，进而可以将转换后的数据存储在一个文件或内存缓冲区中，
# 并且在需要的时候可以恢复成原来的数据结构或对象。

# 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:
    def serialize(self, root: Optional[TreeNode]) -> str:
        """
        序列化二叉树
        方法: 使用层序遍历进行序列化
        思路: 使用BFS层序遍历，将每个节点的值转换为字符串，空节点用"null"表示
        时间复杂度: O(n) - n是树中节点的数量
        空间复杂度: O(n) - 存储序列化字符串和遍历所需的队列空间
        """
        if not root:
            return "[]"
```

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

    while queue:
        node = queue.popleft()
        if node:
            result.append(str(node.val))
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        else:
            result.append("null")

    # 移除末尾的null值
    while result and result[-1] == "null":
        result.pop()

    return "[" + ",".join(result) + "]"

def deserialize(self, data: str) -> Optional[TreeNode]:
    """
    反序列化二叉树
    方法: 使用层序遍历进行反序列化
    思路: 将序列化的字符串解析为节点值列表，然后使用BFS方式重建树
    时间复杂度: O(n) - n是树中节点的数量
    空间复杂度: O(n) - 存储节点值列表和重建树所需的队列空间
    """
    if not data or data == "[]":
        return None

    # 解析字符串为节点值列表
    values = data[1:-1].split(",")
    if not values or values[0] == "null":
        return None

    root = TreeNode(int(values[0]))
    queue = deque([root])
    i = 1

    while queue and i < len(values):
        node = queue.popleft()
```

```
            # 处理左子节点
            if values[i] != "null":
                node.left = TreeNode(int(values[i]))
                queue.append(node.left)
            i += 1

            # 处理右子节点
            if i < len(values) and values[i] != "null":
                node.right = TreeNode(int(values[i]))
                queue.append(node.right)
            i += 1

        return root

# 测试代码
if __name__ == "__main__":
    # 构建测试二叉树:
    #       1
    #      / \
    #     2   3
    #        / \
    #       4   5
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.right.left = TreeNode(4)
    root.right.right = TreeNode(5)

    codec = Codec()

    # 序列化
    serialized = codec.serialize(root)
    print("序列化结果:", serialized)

    # 反序列化
    deserialized = codec.deserialize(serialized)
    print("反序列化结果:", codec.serialize(deserialized))

==========================================

文件: LeetCode429_NaryTreeLevelOrderTraversal.cpp
==========================================
#include <vector>
```

```cpp
#include <queue>
#include <algorithm>

using namespace std;

// LeetCode 429. N 叉树的层序遍历
// 题目链接: https://leetcode.cn/problems/n-ary-tree-level-order-traversal/
// 题目大意: 给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）
// 树的序列化输入是用层序遍历，每组子节点都由 null 值分隔

// N 叉树节点定义
class Node {
public:
    int val;
    vector<Node*> children;

    Node() {}

    Node(int _val) {
        val = _val;
    }

    Node(int _val, vector<Node*> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
public:
    /**
     * 方法 1: 使用 BFS 层序遍历
     * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    vector<vector<int>> levelOrder1(Node* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
            return result;
        }

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

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

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

                // 将所有子节点加入队列
                for (Node* child : node->children) {
                    if (child != nullptr) {
                        q.push(child);
                    }
                }
            }

            result.push_back(level);
        }

        return result;
    }

    /**
     * 方法 2: 使用 DFS 递归遍历
     * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
     * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
     */
    vector<vector<int>> levelOrder2(Node* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
            return result;
        }

        dfs(root, 0, result);
        return result;
    }

private:
    void dfs(Node* node, int level, vector<vector<int>>& result) {
        if (node == nullptr) {
            return;
```

```
        }

        // 如果当前层级还没有对应的列表，创建一个新的
        if (result.size() <= level) {
            result.push_back(vector<int>());
        }

        // 将当前节点值添加到对应层级的列表中
        result[level].push_back(node->val);

        // 递归处理所有子节点
        for (Node* child : node->children) {
            dfs(child, level + 1, result);
        }
    }
};

==========================================

文件: LeetCode429_NaryTreeLevelOrderTraversal.java
==========================================
package class036;

import java.util.*;

// LeetCode 429. N 叉树的层序遍历
// 题目链接: https://leetcode.cn/problems/n-ary-tree-level-order-traversal/
// 题目大意: 给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）
// 树的序列化输入是用层序遍历，每组子节点都由 null 值分隔

public class LeetCode429_NaryTreeLevelOrderTraversal {

    // N 叉树节点定义
    static class Node {
        public int val;
        public List<Node> children;

        public Node() {}

        public Node(int _val) {
            val = _val;
        }
```

```
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};

/**
 * 方法 1: 使用 BFS 层序遍历
 * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static List<List<Integer>> levelOrder1(Node root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    Queue<Node> 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++) {
            Node node = queue.poll();
            level.add(node.val);

            // 将所有子节点加入队列
            if (node.children != null) {
                for (Node child : node.children) {
                    if (child != null) {
                        queue.offer(child);
                    }
                }
            }
        }

        result.add(level);
    }

    return result;
}
```

```
/**
 * 方法 2: 使用 DFS 递归遍历
 * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
 * 空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
 */
public static List<List<Integer>> levelOrder2(Node root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    dfs(root, 0, result);
    return result;
}

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

    // 如果当前层级还没有对应的列表，创建一个新的
    if (result.size() <= level) {
        result.add(new ArrayList<>());
    }

    // 将当前节点值添加到对应层级的列表中
    result.get(level).add(node.val);

    // 递归处理所有子节点
    if (node.children != null) {
        for (Node child : node.children) {
            dfs(child, level + 1, result);
        }
    }
}

// 测试方法
public static void main(String[] args) {
    // 构建测试 N 叉树:
    //       1
    //    /  |  \
    //   3   2   4
```

```
        //  / \
        // 5   6
        Node root = new Node(1);
        List<Node> children1 = new ArrayList<>();
        Node node3 = new Node(3);
        Node node2 = new Node(2);
        Node node4 = new Node(4);
        children1.add(node3);
        children1.add(node2);
        children1.add(node4);
        root.children = children1;

        List<Node> children3 = new ArrayList<>();
        Node node5 = new Node(5);
        Node node6 = new Node(6);
        children3.add(node5);
        children3.add(node6);
        node3.children = children3;

        System.out.println("方法 1 结果: " + levelOrder1(root));
        System.out.println("方法 2 结果: " + levelOrder2(root));
    }
}
```

==========

文件: LeetCode429_NaryTreeLevelOrderTraversal.py

==========

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

# LeetCode 429. N 叉树的层序遍历
# 题目链接: https://leetcode.cn/problems/n-ary-tree-level-order-traversal/
# 题目大意: 给定一个 N 叉树，返回其节点值的层序遍历。（即从左到右，逐层遍历）
# 树的序列化输入是用层序遍历，每组子节点都由 null 值分隔

# N 叉树节点定义
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

class Solution:
```

```
def levelOrder1(self, root: Optional[Node]) -> List[List[int]]:
    """
    方法 1: 使用 BFS 层序遍历
    时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
    空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
    """
    result = []
    if not root:
        return result

    queue = deque([root])

    while queue:
        size = len(queue)
        level = []

        for _ in range(size):
            node = queue.popleft()
            level.append(node.val)

            # 将所有子节点加入队列
            for child in node.children:
                if child:
                    queue.append(child)

        result.append(level)

    return result

def levelOrder2(self, root: Optional[Node]) -> List[List[int]]:
    """
    方法 2: 使用 DFS 递归遍历
    时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
    空间复杂度: O(h) - h 是树的高度，递归调用栈的深度
    """
    result = []
    if not root:
        return result

    self.dfs(root, 0, result)
    return result

def dfs(self, node: Optional[Node], level: int, result: List[List[int]]) -> None:
```

```
        """
        深度优先搜索辅助函数
        Args:
            node: 当前节点
            level: 当前层级
            result: 结果列表
        """
        if not node:
            return

        # 如果当前层级还没有对应的列表，创建一个新的
        if len(result) <= level:
            result.append([])

        # 将当前节点值添加到对应层级的列表中
        if node.val is not None:
            result[level].append(node.val)

        # 递归处理所有子节点
        for child in node.children:
            self.dfs(child, level + 1, result)

# 测试代码
if __name__ == "__main__":
    # 构建测试N叉树:
    #       1
    #    /  |  \
    #   3   2   4
    #  / \
    # 5   6
    root = Node(1)
    node3 = Node(3)
    node2 = Node(2)
    node4 = Node(4)
    root.children = [node3, node2, node4]

    node5 = Node(5)
    node6 = Node(6)
    node3.children = [node5, node6]

    solution = Solution()
    print("方法1结果:", solution.levelOrder1(root))
    print("方法2结果:", solution.levelOrder2(root))
```

```
==========================================

文件: LeetCode513_FindBottomLeftTreeValue.java
==========================================
package class036;

import java.util.*;

/**
 * LeetCode 513. 找树左下角的值
 * 题目链接: https://leetcode.cn/problems/find-bottom-left-tree-value/
 * 题目描述: 给定一个二叉树的根节点 root，请找出该二叉树的最后一行最左边的值。
 *
 * 核心算法思想:
 * 1. 层序遍历(BFS): 从右到左进行层序遍历，最后一个节点即为左下角值
 * 2. 深度优先遍历(DFS): 记录最大深度和对应值
 * 3. 优化的 BFS: 使用数组实现队列，提升性能
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，其中 N 是二叉树中的节点数
 *
 * 空间复杂度分析:
 * - 方法 1(层序遍历): O(W)，W 为树的最大宽度
 * - 方法 2(DFS 递归): O(H)，H 为树的高度
 * - 方法 3(优化 BFS): O(W)，W 为树的最大宽度
 *
 * 相关题目:
 * 1. LeetCode 199. 二叉树的右视图 - 类似的分层处理
 * 2. LeetCode 515. 在每个树行中找最大值 - 分层极值查找
 * 3. LeetCode 637. 二叉树的层平均值 - 分层统计
 *
 * 工程化考量:
 * 1. 边界处理: 处理空树和单节点树
 * 2. 性能优化: 选择合适的数据结构
 * 3. 可读性: 代码结构清晰
 */
public class LeetCode513_FindBottomLeftTreeValue {

    // 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: 层序遍历法 - 从右到左遍历
 * 思路: 从右到左进行层序遍历，最后一个节点即为左下角值
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度
 */
public static int findBottomLeftValue1(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    TreeNode result = root;

    while (!queue.isEmpty()) {
        result = queue.peek(); // 记录当前层第一个节点
        int size = queue.size();

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();

            // 先右后左加入队列
            if (current.right != null) {
                queue.offer(current.right);
            }
            if (current.left != null) {
                queue.offer(current.left);
            }
        }
    }

    return result.val;
}

/**
 * 方法 2: 深度优先遍历(DFS) - 记录最大深度
```

```
 * 思路: 使用 DFS 记录最大深度和对应值
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(H) - H 为树的高度
 */
public static int findBottomLeftValue2(TreeNode root) {
    int[] maxDepth = {-1};
    int[] result = {root.val};
    dfs(root, 0, maxDepth, result);
    return result[0];
}

private static void dfs(TreeNode node, int depth, int[] maxDepth, int[] result) {
    if (node == null) {
        return;
    }

    // 如果是更深层的第一个节点
    if (depth > maxDepth[0]) {
        maxDepth[0] = depth;
        result[0] = node.val;
    }

    // 先左后右，保证找到的是最左边的值
    dfs(node.left, depth + 1, maxDepth, result);
    dfs(node.right, depth + 1, maxDepth, result);
}

/**
 * 方法 3: 优化的 BFS - 使用数组实现队列
 * 思路: 使用数组代替 LinkedList，提升性能
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度
 */
public static int findBottomLeftValue3(TreeNode root) {
    TreeNode[] queue = new TreeNode[1000];
    int l = 0, r = 0;
    queue[r++] = root;
    TreeNode result = root;

    while (l < r) {
        result = queue[l]; // 记录当前层第一个节点
        int size = r - l;
```

```
            for (int i = 0; i < size; i++) {
                TreeNode current = queue[l++];

                if (current.right != null) {
                    queue[r++] = current.right;
                }
                if (current.left != null) {
                    queue[r++] = current.left;
                }
            }
        }

        return result.val;
    }

    /**
     * 辅助方法: 根据数组构建测试树
     */
    public static TreeNode buildTree(Integer[] arr) {
        if (arr == null || arr.length == 0 || arr[0] == null) {
            return null;
        }

        TreeNode root = new TreeNode(arr[0]);
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int i = 1;

        while (!queue.isEmpty() && i < arr.length) {
            TreeNode current = queue.poll();

            if (i < arr.length && arr[i] != null) {
                current.left = new TreeNode(arr[i]);
                queue.offer(current.left);
            }
            i++;

            if (i < arr.length && arr[i] != null) {
                current.right = new TreeNode(arr[i]);
                queue.offer(current.right);
            }
            i++;
        }
```

```
        return root;
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== LeetCode 513 测试 ==========");

        // 测试用例 1: 标准二叉树 [2,1,3]
        Integer[] arr1 = {2, 1, 3};
        TreeNode root1 = buildTree(arr1);

        System.out.println("方法 1 结果: " + findBottomLeftValue1(root1));
        System.out.println("方法 2 结果: " + findBottomLeftValue2(root1));
        System.out.println("方法 3 结果: " + findBottomLeftValue3(root1));

        // 测试用例 2: 复杂二叉树 [1,2,3,4,null,5,6,null,null,7]
        Integer[] arr2 = {1, 2, 3, 4, null, 5, 6, null, null, 7};
        TreeNode root2 = buildTree(arr2);
        System.out.println("方法 1 结果: " + findBottomLeftValue1(root2));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，逻辑清晰");
        System.out.println("2. 方法 2（DFS 递归）: 空间复杂度优，适合平衡树");
        System.out.println("3. 方法 3（优化 BFS）: 性能最优，适合大数据量");
    }
}

/*
Python 实现:

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

class Solution:
    def findBottomLeftValue(self, root: TreeNode) -> int:
        from collections import deque
```

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

        while queue:
            result = queue[0]
            size = len(queue)

            for i in range(size):
                current = queue.popleft()

                if current.right:
                    queue.append(current.right)
                if current.left:
                    queue.append(current.left)

        return result.val
```

C++实现:

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

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:
    int findBottomLeftValue(TreeNode* root) {
        queue<TreeNode*> q;
        q.push(root);
        TreeNode* result = root;

        while (!q.empty()) {
            result = q.front();
            int size = q.size();
```

```
            for (int i = 0; i < size; ++i) {
                TreeNode* current = q.front();
                q.pop();

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

        return result->val;
    }
};
*/
```

==========

文件: LeetCode515_FindLargestValueInEachTreeRow.java

==========

```
package class036;

import java.util.*;

/**
 * LeetCode 515. 在每个树行中找最大值
 * 题目链接: https://leetcode.cn/problems/find-largest-value-in-each-tree-row/
 * 题目描述: 给定一棵二叉树的根节点 root，请找出该二叉树中每一层的最大值。
 *
 * 核心算法思想:
 * 1. 层序遍历(BFS): 使用队列进行层序遍历，记录每层的最大值
 * 2. 深度优先遍历(DFS): 使用递归记录每层的最大值
 * 3. 优化的BFS: 使用数组实现队列，提升性能
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，其中N是二叉树中的节点数
 *
 * 空间复杂度分析:
 * - 方法1(层序遍历): O(W)，W为树的最大宽度
 * - 方法2(DFS递归): O(H)，H为树的高度
 * - 方法3(优化BFS): O(W)，W为树的最大宽度
 *
 * 极值处理考量:
 * 1. 处理空树和单节点树的边界情况
 * 2. 考虑节点值为Integer.MIN_VALUE的情况
```

```
 * 3. 处理大数比较的边界条件
 *
 * 相关题目:
 * 1. LeetCode 102. 二叉树的层序遍历 - 基础层序遍历
 * 2. LeetCode 199. 二叉树的右视图 - 分层处理
 * 3. LeetCode 637. 二叉树的层平均值 - 分层统计
 * 4. LeetCode 513. 找树左下角的值 - 分层极值查找
 *
 * 工程化考量:
 * 1. 边界处理: 完善各种边界情况的处理
 * 2. 性能优化: 选择合适的数据结构
 * 3. 可读性: 代码结构清晰，注释完整
 */
public class LeetCode515_FindLargestValueInEachTreeRow {

    // 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: 层序遍历法 - 基础BFS实现
     * 思路: 使用队列进行层序遍历，记录每层的最大值
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(W) - W为树的最大宽度
     */
    public static List<Integer> largestValues1(TreeNode root) {
        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();
        int maxVal = Integer.MIN_VALUE;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            maxVal = Math.max(maxVal, current.val);

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

        result.add(maxVal);
    }

    return result;
}

/**
 * 方法 2: 深度优先遍历(DFS) - 递归实现
 * 思路: 使用递归进行 DFS，记录每层的最大值
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(H) - H 为树的高度
 */
public static List<Integer> largestValues2(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    dfs(root, 0, result);
    return result;
}

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

    if (level == result.size()) {
        result.add(node.val);
    } else {
```

```
        result.set(level, Math.max(result.get(level), node.val));
    }

    dfs(node.left, level + 1, result);
    dfs(node.right, level + 1, result);
}

/**
 * 方法 3: 优化的 BFS - 使用数组实现队列
 * 思路: 使用数组代替 LinkedList，提升性能
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度
 */
public static List<Integer> largestValues3(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    TreeNode[] queue = new TreeNode[1000];
    int l = 0, r = 0;
    queue[r++] = root;

    while (l < r) {
        int size = r - l;
        int maxVal = Integer.MIN_VALUE;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue[l++];
            maxVal = Math.max(maxVal, current.val);

            if (current.left != null) {
                queue[r++] = current.left;
            }
            if (current.right != null) {
                queue[r++] = current.right;
            }
        }

        result.add(maxVal);
    }

    return result;
```

```
}

/**
 * 辅助方法: 根据数组构建测试树
 */
public static TreeNode buildTree(Integer[] arr) {
    if (arr == null || arr.length == 0 || arr[0] == null) {
        return null;
    }

    TreeNode root = new TreeNode(arr[0]);
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty() && i < arr.length) {
        TreeNode current = queue.poll();

        if (i < arr.length && arr[i] != null) {
            current.left = new TreeNode(arr[i]);
            queue.offer(current.left);
        }
        i++;

        if (i < arr.length && arr[i] != null) {
            current.right = new TreeNode(arr[i]);
            queue.offer(current.right);
        }
        i++;
    }

    return root;
}

/**
 * 测试方法: 包含多种测试用例
 */
public static void main(String[] args) {
    System.out.println("========== LeetCode 515 测试 ==========");

    // 测试用例 1: 标准二叉树 [1,3,2,5,3,null,9]
    Integer[] arr1 = {1, 3, 2, 5, 3, null, 9};
    TreeNode root1 = buildTree(arr1);
```

```
        System.out.println("方法 1 结果: " + largestValues1(root1));
        System.out.println("方法 2 结果: " + largestValues2(root1));
        System.out.println("方法 3 结果: " + largestValues3(root1));

        // 测试用例 2: 单节点树
        TreeNode root2 = new TreeNode(5);
        System.out.println("方法 1 结果: " + largestValues1(root2));

        // 测试用例 3: 斜树
        Integer[] arr3 = {1, 2, null, 3, null, 4, null};
        TreeNode root3 = buildTree(arr3);
        System.out.println("方法 1 结果: " + largestValues1(root3));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，逻辑清晰");
        System.out.println("2. 方法 2（DFS 递归）: 空间复杂度优，适合平衡树");
        System.out.println("3. 方法 3（优化 BFS）: 性能最优，适合大数据量");
    }
}

/*
Python 实现:

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

class Solution:
    def largestValues(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            size = len(queue)
            max_val = float('-inf')
```

```
            for i in range(size):
                current = queue.popleft()
                max_val = max(max_val, current.val)

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(max_val)

        return result
```

C++实现:

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

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:
    vector<int> largestValues(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

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

        while (!q.empty()) {
            int size = q.size();
            int maxVal = INT_MIN;
```

```
            for (int i = 0; i < size; ++i) {
                TreeNode* current = q.front();
                q.pop();
                maxVal = max(maxVal, current->val);

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

            result.push_back(maxVal);
        }

        return result;
    }
};
*/
```

==========

文件: LeetCode637_AverageOfLevelsInBinaryTree.java

==========

```
package class036;

import java.util.*;

/**
 * LeetCode 637. 二叉树的层平均值
 * 题目链接: https://leetcode.cn/problems/average-of-levels-in-binary-tree/
 * 题目描述: 给定一个非空二叉树的根节点 root，以数组的形式返回每一层节点的平均值。
 *
 * 核心算法思想:
 * 1. 层序遍历(BFS): 使用队列进行层序遍历，计算每层节点的平均值
 * 2. 深度优先遍历(DFS): 使用递归记录每层的节点值和数量
 * 3. 优化的 BFS: 使用数组实现队列，减少对象创建开销
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，其中 N 是二叉树中的节点数
 *
 * 空间复杂度分析:
 * - 方法 1(层序遍历): O(W)，W 为树的最大宽度
 * - 方法 2(DFS 递归): O(H)，H 为树的高度
 * - 方法 3(优化 BFS): O(W)，W 为树的最大宽度
```

```
 *
 * 精度处理考量:
 * 1. 使用 double 类型避免整数除法精度损失
 * 2. 处理大数相加时的溢出问题
 * 3. 考虑极端情况下的数值稳定性
 *
 * 相关题目:
 * 1. LeetCode 102. 二叉树的层序遍历 - 基础层序遍历
 * 2. LeetCode 107. 二叉树的层序遍历 II - 自底向上层序遍历
 * 3. LeetCode 199. 二叉树的右视图 - 分层处理
 * 4. LeetCode 515. 在每个树行中找最大值 - 分层统计极值
 *
 * 工程化考量:
 * 1. 数值精度: 使用 double 类型保证计算精度
 * 2. 溢出处理: 对于大数据量考虑使用 BigDecimal
 * 3. 性能优化: 选择合适的数据结构提升性能
 */
public class LeetCode637_AverageOfLevelsInBinaryTree {

    // 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: 层序遍历法 - 基础 BFS 实现
     * 思路: 使用队列进行层序遍历，计算每层节点的平均值
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(W) - W 为树的最大宽度
     *
     * 优点:
     * - 逻辑清晰，易于理解和实现
     * - 适用于各种二叉树结构
     * 缺点:
```

```
 * - 需要额外的队列空间
 *
 * 关键步骤:
 * 1. 使用队列存储当前层的所有节点
 * 2. 记录每层的节点数量和总和
 * 3. 计算平均值并加入结果列表
 */
public static List<Double> averageOfLevels1(TreeNode root) {
    List<Double> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

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

    while (!queue.isEmpty()) {
        int size = queue.size();
        double sum = 0;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            sum += current.val;

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

        // 计算平均值，注意使用 double 避免精度损失
        result.add(sum / size);
    }

    return result;
}

/**
 * 方法 2: 深度优先遍历(DFS) - 递归实现
 * 思路: 使用递归进行 DFS，记录每层的节点总和和数量
 * 时间复杂度: O(N) - 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
 *
 * 核心思想:
 * 1. 使用两个列表分别记录每层的总和和节点数量
 * 2. 递归遍历时更新对应层级的统计信息
 * 3. 遍历完成后计算每层的平均值
 *
 * 优点:
 * - 空间复杂度较低（树的高度通常远小于节点数）
 * - 避免使用队列，减少对象创建
 * 缺点:
 * - 递归深度可能较大
 * - 需要额外的存储空间记录统计信息
 */
public static List<Double> averageOfLevels2(TreeNode root) {
    // 存储每层的节点值总和
    List<Double> sums = new ArrayList<>();
    // 存储每层的节点数量
    List<Integer> counts = new ArrayList<>();

    dfs(root, 0, sums, counts);

    // 计算每层的平均值
    List<Double> result = new ArrayList<>();
    for (int i = 0; i < sums.size(); i++) {
        result.add(sums.get(i) / counts.get(i));
    }

    return result;
}

private static void dfs(TreeNode node, int level, List<Double> sums, List<Integer> counts) {
    if (node == null) {
        return;
    }

    // 如果当前层级还没有统计信息，初始化
    if (level == sums.size()) {
        sums.add(0.0);
        counts.add(0);
    }

    // 更新当前层级的统计信息
```

```
        sums.set(level, sums.get(level) + node.val);
        counts.set(level, counts.get(level) + 1);

        // 递归处理左右子树
        dfs(node.left, level + 1, sums, counts);
        dfs(node.right, level + 1, sums, counts);
    }

    /**
     * 方法 3: 优化的 BFS - 使用数组实现队列
     * 思路: 使用数组代替 LinkedList，减少对象创建和内存分配开销
     * 时间复杂度: O(N) - 每个节点访问一次
     * 空间复杂度: O(W) - W 为树的最大宽度
     *
     * 工程化优化:
     * 1. 使用数组代替 LinkedList 减少内存分配
     * 2. 预分配队列大小，避免动态扩容
     * 3. 减少对象创建，提升性能
     */
    public static List<Double> averageOfLevels3(TreeNode root) {
        List<Double> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        // 预分配队列大小
        TreeNode[] queue = new TreeNode[1000];
        int l = 0, r = 0;
        queue[r++] = root;

        while (l < r) {
            int size = r - l;
            double sum = 0;

            for (int i = 0; i < size; i++) {
                TreeNode current = queue[l++];
                sum += current.val;

                if (current.left != null) {
                    queue[r++] = current.left;
                }
                if (current.right != null) {
                    queue[r++] = current.right;
```

```
            }
        }

        result.add(sum / size);
    }

    return result;
}

/**
 * 方法 4: 防止数值溢出的安全版本
 * 思路: 使用 BigDecimal 处理大数相加，避免溢出问题
 * 适用场景: 节点值很大或层节点数很多时
 * 时间复杂度: O(N) - 每个节点访问一次
 * 空间复杂度: O(W) - W 为树的最大宽度
 */
public static List<Double> averageOfLevels4(TreeNode root) {
    List<Double> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

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

    while (!queue.isEmpty()) {
        int size = queue.size();
        // 使用 long 避免整数溢出
        long sum = 0;

        for (int i = 0; i < size; i++) {
            TreeNode current = queue.poll();
            sum += current.val;

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

        // 使用 double 计算平均值
```

```
            result.add((double) sum / size);
        }

        return result;
    }

    /**
     * 辅助方法: 根据数组构建测试树
     */
    public static TreeNode buildTree(Integer[] arr) {
        if (arr == null || arr.length == 0 || arr[0] == null) {
            return null;
        }

        TreeNode root = new TreeNode(arr[0]);
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int i = 1;

        while (!queue.isEmpty() && i < arr.length) {
            TreeNode current = queue.poll();

            if (i < arr.length && arr[i] != null) {
                current.left = new TreeNode(arr[i]);
                queue.offer(current.left);
            }
            i++;

            if (i < arr.length && arr[i] != null) {
                current.right = new TreeNode(arr[i]);
                queue.offer(current.right);
            }
            i++;
        }

        return root;
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== LeetCode 637 测试 ==========");
```

```
        // 测试用例 1: 标准二叉树 [3,9,20,null,null,15,7]
        System.out.println("\n 测试用例 1: 标准二叉树");
        Integer[] arr1 = {3, 9, 20, null, null, 15, 7};
        TreeNode root1 = buildTree(arr1);

        System.out.println("方法 1 结果: " + averageOfLevels1(root1));
        System.out.println("方法 2 结果: " + averageOfLevels2(root1));
        System.out.println("方法 3 结果: " + averageOfLevels3(root1));
        System.out.println("方法 4 结果: " + averageOfLevels4(root1));

        // 测试用例 2: 单节点树
        System.out.println("\n 测试用例 2: 单节点树");
        TreeNode root2 = new TreeNode(5);
        System.out.println("方法 1 结果: " + averageOfLevels1(root2));

        // 测试用例 3: 斜树
        System.out.println("\n 测试用例 3: 斜树");
        Integer[] arr3 = {1, 2, null, 3, null, 4, null};
        TreeNode root3 = buildTree(arr3);
        System.out.println("方法 1 结果: " + averageOfLevels1(root3));

        // 测试用例 4: 大数测试（避免溢出）
        System.out.println("\n 测试用例 4: 大数测试");
        TreeNode root4 = new TreeNode(Integer.MAX_VALUE);
        root4.left = new TreeNode(Integer.MAX_VALUE);
        root4.right = new TreeNode(Integer.MAX_VALUE);
        System.out.println("方法 4 结果: " + averageOfLevels4(root4));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（层序遍历）: 通用性强，逻辑清晰");
        System.out.println("2. 方法 2（DFS 递归）: 空间复杂度优，适合平衡树");
        System.out.println("3. 方法 3（优化 BFS）: 性能最优，适合大数据量");
        System.out.println("4. 方法 4（安全版本）: 防止数值溢出，适合大数场景");
    }
}

/*
Python 实现:

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: 层序遍历法
    def averageOfLevels1(self, root: TreeNode) -> List[float]:
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            size = len(queue)
            level_sum = 0

            for i in range(size):
                current = queue.popleft()
                level_sum += current.val

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(level_sum / size)

        return result

    # 方法 2: DFS 递归
    def averageOfLevels2(self, root: TreeNode) -> List[float]:
        sums = []
        counts = []

        def dfs(node, level):
            if not node:
                return

            if level == len(sums):
                sums.append(0)
                counts.append(0)
```

```
            sums[level] += node.val
            counts[level] += 1

            dfs(node.left, level + 1)
            dfs(node.right, level + 1)

        dfs(root, 0)
        return [s / c for s, c in zip(sums, counts)]
```

C++实现:

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

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: 层序遍历法
    vector<double> averageOfLevels(TreeNode* root) {
        vector<double> result;
        if (!root) return result;

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

        while (!q.empty()) {
            int size = q.size();
            double sum = 0;

            for (int i = 0; i < size; ++i) {
                TreeNode* current = q.front();
                q.pop();
```

```
                sum += current->val;

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

            result.push_back(sum / size);
        }

        return result;
    }
};
*/
```

==========

文件: LeetCode662_MaximumWidthOfBinaryTree.cpp

==========

```
// LeetCode 662. 二叉树最大宽度
// 题目链接: https://leetcode.cn/problems/maximum-width-of-binary-tree/
// 题目大意: 给你一棵二叉树的根节点 root ，返回树的 最大宽度 。
// 树的 最大宽度 是所有层中最大的 宽度 。
// 每一层的 宽度 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。
// 将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 null 节点，这些 null 节点也计入长度。

// 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: 使用 BFS 层序遍历，给每个节点分配位置索引
     * 思路: 在完全二叉树中，如果父节点的位置是 i，那么左子节点的位置是 2*i，右子节点的位置是 2*i+1
     * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
```

```
     */
    int widthOfBinaryTree1(TreeNode* root) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要使用队列来存储节点和索引
        return 0;
    }

    /**
     * 方法2: 优化的BFS，避免索引过大导致的整数溢出
     * 思路: 每层重新编号，将最左边节点的索引作为基准(1)，其他节点相对编号
     * 时间复杂度: O(n) - n是树中节点的数量，每个节点访问一次
     * 空间复杂度: O(w) - w是树的最大宽度，队列中最多存储一层的节点
     */
    int widthOfBinaryTree2(TreeNode* root) {
        // 由于缺少标准库支持，这里只提供函数签名
        // 实际实现需要使用队列来存储节点和索引
        return 0;
    }
};
```

```
==============================================
```

文件: LeetCode662_MaximumWidthOfBinaryTree.java

```
==============================================
package class036;

import java.util.*;

// LeetCode 662. 二叉树最大宽度
// 题目链接: https://leetcode.cn/problems/maximum-width-of-binary-tree/
// 题目大意: 给你一棵二叉树的根节点 root ，返回树的 最大宽度 。
// 树的 最大宽度 是所有层中最大的 宽度 。
// 每一层的 宽度 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。
// 将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 null 节点，这些 null 节点也计入长度。

public class LeetCode662_MaximumWidthOfBinaryTree {

    // 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: 使用 BFS 层序遍历，给每个节点分配位置索引
 * 思路: 在完全二叉树中，如果父节点的位置是 i，那么左子节点的位置是 2*i，右子节点的位置是 2*i+1
 * 时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static int widthOfBinaryTree1(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 使用队列存储节点及其位置索引
    Queue<TreeNode> nodeQueue = new LinkedList<>();
    Queue<Integer> indexQueue = new LinkedList<>();
    nodeQueue.offer(root);
    indexQueue.offer(1);

    int maxWidth = 0;

    while (!nodeQueue.isEmpty()) {
        int size = nodeQueue.size();
        int leftIndex = indexQueue.peek(); // 当前层最左边节点的索引
        int rightIndex = leftIndex; // 初始化为最左边节点的索引

        for (int i = 0; i < size; i++) {
            TreeNode node = nodeQueue.poll();
            int index = indexQueue.poll();
            rightIndex = index; // 更新最右边节点的索引

            // 添加子节点到队列
            if (node.left != null) {
                nodeQueue.offer(node.left);
                indexQueue.offer(index * 2);
            }
```

```
            if (node.right != null) {
                nodeQueue.offer(node.right);
                indexQueue.offer(index * 2 + 1);
            }
        }

        // 计算当前层的宽度
        maxWidth = Math.max(maxWidth, rightIndex - leftIndex + 1);
    }

    return maxWidth;
}

/**
 * 方法2: 优化的BFS，避免索引过大导致的整数溢出
 * 思路: 每层重新编号，将最左边节点的索引作为基准(1)，其他节点相对编号
 * 时间复杂度: O(n) - n是树中节点的数量，每个节点访问一次
 * 空间复杂度: O(w) - w是树的最大宽度，队列中最多存储一层的节点
 */
public static int widthOfBinaryTree2(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 使用队列存储节点及其位置索引
    Queue<TreeNode> nodeQueue = new LinkedList<>();
    Queue<Integer> indexQueue = new LinkedList<>();
    nodeQueue.offer(root);
    indexQueue.offer(1);

    int maxWidth = 0;

    while (!nodeQueue.isEmpty()) {
        int size = nodeQueue.size();
        int leftIndex = indexQueue.peek(); // 当前层最左边节点的索引

        for (int i = 0; i < size; i++) {
            TreeNode node = nodeQueue.poll();
            int index = indexQueue.poll();

            // 重新编号，避免索引过大
            int normalizedIndex = index - leftIndex + 1;
```

```
                // 添加子节点到队列
                if (node.left != null) {
                    nodeQueue.offer(node.left);
                    indexQueue.offer(normalizedIndex * 2);
                }
                if (node.right != null) {
                    nodeQueue.offer(node.right);
                    indexQueue.offer(normalizedIndex * 2 + 1);
                }
            }

            // 计算当前层的宽度
            int rightIndex = indexQueue.isEmpty() ? 1 : indexQueue.peek();
            maxWidth = Math.max(maxWidth, rightIndex);
        }

        return maxWidth;
    }

    // 测试方法
    public static void main(String[] args) {
        // 构建测试二叉树: [1,3,2,5,3,null,9]
        //       1
        //      / \
        //     3   2
        //    / \   \
        //   5   3   9
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(3);
        root.right = new TreeNode(2);
        root.left.left = new TreeNode(5);
        root.left.right = new TreeNode(3);
        root.right.right = new TreeNode(9);

        System.out.println("方法1结果: " + widthOfBinaryTree1(root));
        System.out.println("方法2结果: " + widthOfBinaryTree2(root));
    }
}
```

```
==========================================
```

文件: LeetCode662_MaximumWidthOfBinaryTree.py

```
==========================================
```

```
from collections import deque
from typing import Optional

# LeetCode 662. 二叉树最大宽度
# 题目链接: https://leetcode.cn/problems/maximum-width-of-binary-tree/
# 题目大意: 给你一棵二叉树的根节点 root ，返回树的 最大宽度 。
# 树的 最大宽度 是所有层中最大的 宽度 。
# 每一层的 宽度 被定义为该层最左和最右的非空节点（即，两个端点）之间的长度。
# 将这个二叉树视作与满二叉树结构相同，两端点间会出现一些延伸到这一层的 null 节点，这些 null 节点也计入长度。

# 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 widthOfBinaryTree1(self, root: Optional[TreeNode]) -> int:
        """
        方法1: 使用BFS层序遍历，给每个节点分配位置索引
        思路: 在完全二叉树中，如果父节点的位置是i，那么左子节点的位置是2*i，右子节点的位置是2*i+1
        时间复杂度: O(n) - n是树中节点的数量，每个节点访问一次
        空间复杂度: O(w) - w是树的最大宽度，队列中最多存储一层的节点
        """
        if not root:
            return 0

        # 使用队列存储节点及其位置索引
        queue = deque([(root, 1)])
        max_width = 0

        while queue:
            size = len(queue)
            left_index = queue[0][1]  # 当前层最左边节点的索引
            right_index = left_index  # 初始化为最左边节点的索引

            for _ in range(size):
                node, index = queue.popleft()
                right_index = index  # 更新最右边节点的索引
```

```
            # 添加子节点到队列
            if node.left:
                queue.append((node.left, index * 2))
            if node.right:
                queue.append((node.right, index * 2 + 1))

        # 计算当前层的宽度
        max_width = max(max_width, right_index - left_index + 1)

    return max_width

def widthOfBinaryTree2(self, root: Optional[TreeNode]) -> int:
    """
    方法 2: 优化的 BFS，避免索引过大导致的整数溢出
    思路: 每层重新编号，将最左边节点的索引作为基准(1)，其他节点相对编号
    时间复杂度: O(n) - n 是树中节点的数量，每个节点访问一次
    空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
    """
    if not root:
        return 0

    # 使用队列存储节点及其位置索引
    queue = deque([(root, 1)])
    max_width = 0

    while queue:
        size = len(queue)
        left_index = queue[0][1]  # 当前层最左边节点的索引

        for _ in range(size):
            node, index = queue.popleft()

            # 重新编号，避免索引过大
            normalized_index = index - left_index + 1

            # 添加子节点到队列
            if node.left:
                queue.append((node.left, normalized_index * 2))
            if node.right:
                queue.append((node.right, normalized_index * 2 + 1))

        # 计算当前层的宽度
        right_index = queue[-1][1] if queue else 1
```

```
            max_width = max(max_width, right_index)

        return max_width

# 测试代码
if __name__ == "__main__":
    # 构建测试二叉树: [1,3,2,5,3,null,9]
    #       1
    #      / \
    #     3   2
    #    / \   \
    #   5   3   9
    root = TreeNode(1)
    root.left = TreeNode(3)
    root.right = TreeNode(2)
    root.left.left = TreeNode(5)
    root.left.right = TreeNode(3)
    root.right.right = TreeNode(9)

    solution = Solution()
    print("方法 1 结果:", solution.widthOfBinaryTree1(root))
    print("方法 2 结果:", solution.widthOfBinaryTree2(root))

==============================================

文件: LintCode69_二叉树的层次遍历.java
==============================================
package class036;

import java.util.*;

/**
 * LintCode 69. 二叉树的层次遍历
 * 题目链接: https://www.lintcode.com/problem/69/
 * 题目大意: 给出一棵二叉树，返回其节点值的层次遍历（逐层从左到右访问所有节点）

 */
public class LintCode69_二叉树的层次遍历 {

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

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

/**
 * 层次遍历实现
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 从根节点开始，将节点加入队列
 * 3. 当队列不为空时，记录当前层的节点数量
 * 4. 处理当前层的所有节点，将它们的值加入当前层列表
 * 5. 将节点的左右子节点（如果存在）加入队列
 * 6. 重复步骤 3-5 直到队列为空
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 使用队列存储待访问的节点
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    // 当队列不为空时继续遍历
    while (!queue.isEmpty()) {
        // 记录当前层的节点数量
        int size = queue.size();

        // 存储当前层的节点值
        List<Integer> level = new ArrayList<>();

        // 处理当前层的所有节点
        for (int i = 0; i < size; i++) {
            // 取出队首节点
            TreeNode current = queue.poll();

            // 将当前节点的值加入当前层列表
            level.add(current.val);
```

```
            // 将左右子节点加入队列（如果存在）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        // 将当前层的结果加入最终结果
        result.add(level);
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    // 构建测试二叉树:
    //     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);

    List<List<Integer>> result = levelOrder(root);
    System.out.println("层次遍历结果:");
    for (List<Integer> level : result) {
        System.out.println(level);
    }

    // 测试空树
    TreeNode emptyRoot = null;
    List<List<Integer>> emptyResult = levelOrder(emptyRoot);
    System.out.println("空树遍历结果: " + emptyResult);
}
}
```

```
/*
Python 实现:

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

class Solution:
    def levelOrder(self, root):
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            size = len(queue)
            level = []

            for i in range(size):
                current = queue.popleft()
                level.append(current.val)

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(level)

        return result

C++实现:

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

class TreeNode {
```

```
public:
    int val;
    TreeNode *left, *right;
    TreeNode(int val) {
        this->val = val;
        this->left = this->right = NULL;
    }
};

class Solution {
public:
    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* current = q.front();
                q.pop();
                level.push_back(current->val);

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

            result.push_back(level);
        }

        return result;
    }
};
*/
```

==========

文件: POJ3278_CatchThatCow.java

==========

```
package class036;

import java.util.*;

/**
 * POJ 3278. Catch That Cow
 * 题目链接: http://poj.org/problem?id=3278
 * 题目描述: 农夫在位置 N，牛在位置 K。农夫每次可以移动到 N-1, N+1, 或 2*N。
 * 求农夫抓到牛所需的最少移动次数。
 *
 * 核心算法思想:
 * 1. 广度优先搜索(BFS): 寻找最短路径
 * 2. 双向 BFS: 从农夫和牛同时开始搜索
 * 3. 剪枝优化: 减少不必要的状态扩展
 *
 * 时间复杂度分析:
 * - 方法 1(BFS): O(K) - 最坏情况需要遍历到 K 位置
 * - 方法 2(双向 BFS): O(K^(1/2)) - 搜索空间减半
 * - 方法 3(优化 BFS): O(K) - 带剪枝的 BFS
 *
 * 空间复杂度分析:
 * - 所有方法: O(K) - 需要记录访问状态
 *
 * 相关题目:
 * 1. LeetCode 127. 单词接龙 - 类似的 BFS 最短路径
 * 2. LeetCode 433. 最小基因变化 - 状态转换 BFS
 * 3. POJ 3126. Prime Path - 素数路径 BFS
 *
 * 工程化考量:
 * 1. 边界处理: N 和 K 的范围限制
 * 2. 性能优化: 使用数组代替 HashMap
 * 3. 内存管理: 预分配足够大的数组
 */
public class POJ3278_CatchThatCow {

    /**
     * 方法 1: 广度优先搜索(BFS) - 基础实现
     * 思路: 使用 BFS 寻找从 N 到 K 的最短路径
     * 时间复杂度: O(K) - 最坏情况需要遍历到 K 位置
     * 空间复杂度: O(K) - 队列和访问数组的空间
     */
    public static int catchCow1(int N, int K) {
        if (N >= K) {
```

```
        return N - K; // 只能向左移动
    }

    final int MAX = 100000;
    boolean[] visited = new boolean[MAX + 1];
    int[] dist = new int[MAX + 1];
    Queue<Integer> queue = new LinkedList<>();

    queue.offer(N);
    visited[N] = true;
    dist[N] = 0;

    while (!queue.isEmpty()) {
        int current = queue.poll();

        if (current == K) {
            return dist[current];
        }

        // 三种移动方式
        int[] nextPositions = {current - 1, current + 1, current * 2};

        for (int next : nextPositions) {
            if (next >= 0 && next <= MAX && !visited[next]) {
                visited[next] = true;
                dist[next] = dist[current] + 1;
                queue.offer(next);

                if (next == K) {
                    return dist[next];
                }
            }
        }
    }

    return -1; // 理论上不会执行到这里
}

/**
 * 方法2: 双向BFS - 优化解法
 * 思路: 从农夫位置和牛位置同时开始BFS
 * 时间复杂度: O(K^(1/2)) - 搜索空间减半
 * 空间复杂度: O(K) - 两个队列和访问数组
```

```
 */
public static int catchCow2(int N, int K) {
    if (N >= K) {
        return N - K;
    }

    final int MAX = 100000;
    int[] distFromN = new int[MAX + 1];
    int[] distFromK = new int[MAX + 1];
    boolean[] visitedFromN = new boolean[MAX + 1];
    boolean[] visitedFromK = new boolean[MAX + 1];

    Queue<Integer> queueN = new LinkedList<>();
    Queue<Integer> queueK = new LinkedList<>();

    queueN.offer(N);
    visitedFromN[N] = true;
    distFromN[N] = 0;

    queueK.offer(K);
    visitedFromK[K] = true;
    distFromK[K] = 0;

    while (!queueN.isEmpty() && !queueK.isEmpty()) {
        // 从 N 方向扩展
        int sizeN = queueN.size();
        for (int i = 0; i < sizeN; i++) {
            int current = queueN.poll();

            if (visitedFromK[current]) {
                return distFromN[current] + distFromK[current];
            }

            int[] nextPositions = {current - 1, current + 1, current * 2};
            for (int next : nextPositions) {
                if (next >= 0 && next <= MAX && !visitedFromN[next]) {
                    visitedFromN[next] = true;
                    distFromN[next] = distFromN[current] + 1;
                    queueN.offer(next);
                }
            }
        }
```

```
        // 从K方向扩展
        int sizeK = queueK.size();
        for (int i = 0; i < sizeK; i++) {
            int current = queueK.poll();

            if (visitedFromN[current]) {
                return distFromN[current] + distFromK[current];
            }

            // 反向移动：只能向左移动（因为牛不动）
            if (current - 1 >= 0 && !visitedFromK[current - 1]) {
                visitedFromK[current - 1] = true;
                distFromK[current - 1] = distFromK[current] + 1;
                queueK.offer(current - 1);
            }

            if (current + 1 <= MAX && !visitedFromK[current + 1]) {
                visitedFromK[current + 1] = true;
                distFromK[current + 1] = distFromK[current] + 1;
                queueK.offer(current + 1);
            }

            if (current % 2 == 0 && !visitedFromK[current / 2]) {
                visitedFromK[current / 2] = true;
                distFromK[current / 2] = distFromK[current] + 1;
                queueK.offer(current / 2);
            }
        }
    }

    return -1;
}

/**
 * 方法3: 优化BFS - 带剪枝
 * 思路: 添加剪枝条件，减少不必要的状态扩展
 * 时间复杂度: O(K) - 但常数因子更小
 * 空间复杂度: O(K) - 队列和访问数组
 */
public static int catchCow3(int N, int K) {
    if (N >= K) {
        return N - K;
    }
```

```
final int MAX = 100000;
boolean[] visited = new boolean[MAX + 1];
int[] dist = new int[MAX + 1];
Queue<Integer> queue = new LinkedList<>();

queue.offer(N);
visited[N] = true;
dist[N] = 0;

while (!queue.isEmpty()) {
    int current = queue.poll();

    if (current == K) {
        return dist[current];
    }

    // 剪枝：如果当前位置已经超过 K，只能向左移动
    if (current > K) {
        if (!visited[current - 1]) {
            visited[current - 1] = true;
            dist[current - 1] = dist[current] + 1;
            queue.offer(current - 1);
        }
        continue;
    }

    // 正常三种移动方式
    int[] nextPositions = {current - 1, current + 1, current * 2};

    for (int next : nextPositions) {
        if (next >= 0 && next <= MAX && !visited[next]) {
            visited[next] = true;
            dist[next] = dist[current] + 1;
            queue.offer(next);

            if (next == K) {
                return dist[next];
            }
        }
    }
}
```

```
        return -1;
    }

    /**
     * 测试方法: 包含多种测试用例
     */
    public static void main(String[] args) {
        System.out.println("========== POJ 3278 测试 ==========");

        // 测试用例 1: N=5, K=17
        System.out.println("测试用例 1: N=5, K=17");
        System.out.println("方法 1 结果: " + catchCow1(5, 17));
        System.out.println("方法 2 结果: " + catchCow2(5, 17));
        System.out.println("方法 3 结果: " + catchCow3(5, 17));

        // 测试用例 2: N=10, K=10
        System.out.println("\n 测试用例 2: N=10, K=10");
        System.out.println("方法 1 结果: " + catchCow1(10, 10));

        // 测试用例 3: N=1, K=100000
        System.out.println("\n 测试用例 3: N=1, K=100000");
        System.out.println("方法 1 结果: " + catchCow1(1, 100000));

        // 性能对比说明
        System.out.println("\n========== 性能对比说明 ==========");
        System.out.println("1. 方法 1（BFS）: 通用性强，逻辑清晰");
        System.out.println("2. 方法 2（双向 BFS）: 性能最优，适合大数据量");
        System.out.println("3. 方法 3（优化 BFS）: 带剪枝，实际运行更快");
    }
}

/*
Python 实现:

from collections import deque

def catchCow(N, K):
    if N >= K:
        return N - K

    MAX = 100000
    visited = [False] * (MAX + 1)
    dist = [0] * (MAX + 1)
```

```
    queue = deque([N])
    visited[N] = True

    while queue:
        current = queue.popleft()

        if current == K:
            return dist[current]

        for next_pos in [current-1, current+1, current*2]:
            if 0 <= next_pos <= MAX and not visited[next_pos]:
                visited[next_pos] = True
                dist[next_pos] = dist[current] + 1
                queue.append(next_pos)

    return -1
```

C++实现:

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

int catchCow(int N, int K) {
    if (N >= K) return N - K;

    const int MAX = 100000;
    vector<bool> visited(MAX + 1, false);
    vector<int> dist(MAX + 1, 0);
    queue<int> q;

    q.push(N);
    visited[N] = true;

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        if (current == K) return dist[current];

        int nextPositions[3] = {current-1, current+1, current*2};
        for (int i = 0; i < 3; i++) {
```

```
            int next = nextPositions[i];
            if (next >= 0 && next <= MAX && !visited[next]) {
                visited[next] = true;
                dist[next] = dist[current] + 1;
                q.push(next);
            }
        }
    }

    return -1;
}
*/
```

==========

文件: UVA122_TreesOnTheLevel.cpp

==========

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

// UVA 122. Trees on the Level
// 题目链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=58
// 题目大意: 按照层序遍历的方式构建二叉树并输出节点值。输入格式为(value, path)，其中 path 是由 L 和 R 组成的字符串，
// 表示从根节点到该节点的路径，L 表示左子节点，R 表示右子节点。

// 二叉树节点定义
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int v = 0) : val(v), left(nullptr), right(nullptr) {}
};

// 节点信息结构体，用于存储节点值和路径
struct TreeNodeInfo {
    int val;
```

```
    string path;

    TreeNodeInfo(int v, const string& p) : val(v), path(p) {}
};

/**
 * 根据路径插入节点
 * @param root 根节点
 * @param val 节点值
 * @param path 路径字符串
 * @return 是否插入成功
 */
bool insertNode(TreeNode* root, int val, const string& path) {
    TreeNode* current = root;

    // 根据路径找到要插入的位置
    for (char direction : path) {
        if (direction == 'L') {
            if (current->left == nullptr) {
                current->left = new TreeNode(0); // 临时节点
            }
            current = current->left;
        } else if (direction == 'R') {
            if (current->right == nullptr) {
                current->right = new TreeNode(0); // 临时节点
            }
            current = current->right;
        } else {
            // 无效路径字符
            return false;
        }
    }

    // 检查节点是否已经被赋值
    if (current->val != 0) {
        // 节点已经被赋值，说明重复
        return false;
    }

    // 赋值
    current->val = val;
    return true;
}
```

```
/**
 * 层序遍历
 * @param root 根节点
 * @return 遍历结果，如果树不完整则返回空向量
 */
vector<int> bfs(TreeNode* root) {
    vector<int> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        TreeNode* current = q.front();
        q.pop();

        // 如果节点值为0，说明是临时节点，树不完整
        if (current->val == 0) {
            return vector<int>(); // 返回空向量表示树不完整
        }

        result.push_back(current->val);

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

    return result;
}

/**
 * 构建二叉树并进行层序遍历
 * 思路:
 * 1. 解析输入的节点信息，按照路径构建二叉树
 * 2. 对构建的二叉树进行层序遍历
 * 3. 如果构建过程中发现节点重复或缺失，返回空向量
 * 时间复杂度: O(n) - n是节点数量
 * 空间复杂度: O(n) - 存储节点和队列
 */
vector<int> levelOrderTraversal(const vector<TreeNodeInfo>& nodes) {
```

```
    // 创建根节点
    TreeNode* root = new TreeNode(0); // 临时根节点

    // 根据路径信息构建树
    for (const TreeNodeInfo& nodeInfo : nodes) {
        if (!insertNode(root, nodeInfo.val, nodeInfo.path)) {
            // 如果插入失败，返回空向量
            return vector<int>();
        }
    }

    // 进行层序遍历
    vector<int> result = bfs(root);

    // 释放内存
    // 注意：在实际应用中，应该实现完整的内存管理

    return result;
}

// 测试方法
int main() {
    // 示例测试
    vector<TreeNodeInfo> nodes;
    nodes.push_back(TreeNodeInfo(5, ""));
    nodes.push_back(TreeNodeInfo(3, "L"));
    nodes.push_back(TreeNodeInfo(4, "LL"));
    nodes.push_back(TreeNodeInfo(7, "LR"));

    vector<int> result = levelOrderTraversal(nodes);
    if (result.empty()) {
        cout << "not complete" << endl;
    } else {
        for (size_t i = 0; i < result.size(); i++) {
            if (i > 0) cout << " ";
            cout << result[i];
        }
        cout << endl;
    }

    return 0;
}
```

==========

文件: UVA122_TreesOnTheLevel.java
==========

```
package class036;

import java.util.*;

// UVA 122. Trees on the Level
// 题目链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=58
// 题目大意: 按照层序遍历的方式构建二叉树并输出节点值。输入格式为(value, path)，其中 path 是由 L 和 R 组成的字符串，
// 表示从根节点到该节点的路径，L 表示左子节点，R 表示右子节点。

public class UVA122_TreesOnTheLevel {

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

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

    // 节点信息类，用于存储节点值和路径
    static class TreeNodeInfo {
        int val;
        String path;

        TreeNodeInfo(int val, String path) {
            this.val = val;
            this.path = path;
        }
    }

    /**
```

```
 * 构建二叉树并进行层序遍历
 * 思路:
 * 1. 解析输入的节点信息，按照路径构建二叉树
 * 2. 对构建的二叉树进行层序遍历
 * 3. 如果构建过程中发现节点重复或缺失，返回空列表
 * 时间复杂度: O(n) - n 是节点数量
 * 空间复杂度: O(n) - 存储节点和队列
 */
public static List<Integer> levelOrderTraversal(List<TreeNodeInfo> nodes) {
    // 创建根节点
    TreeNode root = new TreeNode(0); // 临时根节点

    // 根据路径信息构建树
    for (TreeNodeInfo nodeInfo : nodes) {
        if (!insertNode(root, nodeInfo.val, nodeInfo.path)) {
            // 如果插入失败，返回空列表
            return new ArrayList<>();
        }
    }

    // 进行层序遍历
    return bfs(root);
}

/**
 * 根据路径插入节点
 * @param root 根节点
 * @param val 节点值
 * @param path 路径字符串
 * @return 是否插入成功
 */
private static boolean insertNode(TreeNode root, int val, String path) {
    TreeNode current = root;

    // 根据路径找到要插入的位置
    for (int i = 0; i < path.length(); i++) {
        char direction = path.charAt(i);
        if (direction == 'L') {
            if (current.left == null) {
                current.left = new TreeNode(0); // 临时节点
            }
            current = current.left;
        } else if (direction == 'R') {
```

```
            if (current.right == null) {
                current.right = new TreeNode(0); // 临时节点
            }
            current = current.right;
        } else {
            // 无效路径字符
            return false;
        }
    }

    // 检查节点是否已经被赋值
    if (current.val != 0) {
        // 节点已经被赋值，说明重复
        return false;
    }

    // 赋值
    current.val = val;
    return true;
}

/**
 * 层序遍历
 * @param root 根节点
 * @return 遍历结果
 */
private static List<Integer> bfs(TreeNode root) {
    List<Integer> result = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

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

        // 如果节点值为0，说明是临时节点，树不完整
        if (current.val == 0) {
            return new ArrayList<>(); // 返回空列表表示树不完整
        }

        result.add(current.val);

        if (current.left != null) {
            queue.offer(current.left);
```

```
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 示例测试
        List<TreeNodeInfo> nodes = new ArrayList<>();
        nodes.add(new TreeNodeInfo(5, ""));
        nodes.add(new TreeNodeInfo(3, "L"));
        nodes.add(new TreeNodeInfo(4, "LL"));
        nodes.add(new TreeNodeInfo(7, "LR"));

        List<Integer> result = levelOrderTraversal(nodes);
        if (result.isEmpty()) {
            System.out.println("not complete");
        } else {
            for (int i = 0; i < result.size(); i++) {
                if (i > 0) System.out.print(" ");
                System.out.print(result.get(i));
            }
            System.out.println();
        }
    }
}

==========

文件: UVA122_TreesOnTheLevel.py
==========
from collections import deque
from typing import List, Optional

# UVA 122. Trees on the Level
# 题目链接:
https://onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=24&page=show_problem&problem=58
# 题目大意: 按照层序遍历的方式构建二叉树并输出节点值。输入格式为(value, path)，其中 path 是由 L 和 R
```

组成的字符串，
表示从根节点到该节点的路径，L 表示左子节点，R 表示右子节点。

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

# 节点信息类，用于存储节点值和路径
class TreeNodeInfo:
    def __init__(self, val: int, path: str):
        self.val = val
        self.path = path

def level_order_traversal(nodes: List[TreeNodeInfo]) -> List[int]:
    """
    构建二叉树并进行层序遍历
    思路:
    1. 解析输入的节点信息，按照路径构建二叉树
    2. 对构建的二叉树进行层序遍历
    3. 如果构建过程中发现节点重复或缺失，返回空列表
    时间复杂度: O(n) - n 是节点数量
    空间复杂度: O(n) - 存储节点和队列
    """
    # 创建根节点
    root = TreeNode(0)  # 临时根节点

    # 根据路径信息构建树
    for node_info in nodes:
        if not insert_node(root, node_info.val, node_info.path):
            # 如果插入失败，返回空列表
            return []

    # 进行层序遍历
    return bfs(root)

def insert_node(root: TreeNode, val: int, path: str) -> bool:
    """
    根据路径插入节点
    :param root: 根节点
    :param val: 节点值
```

```
    :param path: 路径字符串
    :return: 是否插入成功
    """
    current = root

    # 根据路径找到要插入的位置
    for direction in path:
        if direction == 'L':
            if current.left is None:
                current.left = TreeNode(0)  # 临时节点
            current = current.left
        elif direction == 'R':
            if current.right is None:
                current.right = TreeNode(0)  # 临时节点
            current = current.right
        else:
            # 无效路径字符
            return False

    # 检查节点是否已经被赋值
    if current.val != 0:
        # 节点已经被赋值，说明重复
        return False

    # 赋值
    current.val = val
    return True

def bfs(root: TreeNode) -> List[int]:
    """
    层序遍历
    :param root: 根节点
    :return: 遍历结果
    """
    result = []
    queue = deque([root])

    while queue:
        current = queue.popleft()

        # 如果节点值为0，说明是临时节点，树不完整
        if current.val == 0:
            return []  # 返回空列表表示树不完整
```

```
        result.append(current.val)

        if current.left is not None:
            queue.append(current.left)
        if current.right is not None:
            queue.append(current.right)

    return result

# 测试方法
if __name__ == "__main__":
    # 示例测试
    nodes = [
        TreeNodeInfo(5, ""),
        TreeNodeInfo(3, "L"),
        TreeNodeInfo(4, "LL"),
        TreeNodeInfo(7, "LR")
    ]

    result = level_order_traversal(nodes)
    if not result:
        print("not complete")
    else:
        print(" ".join(map(str, result)))

==========================================

文件: 剑指Offer32_III_从上到下打印二叉树III.java
==========================================
package class036;

import java.util.*;

// 剑指Offer 32-III. 从上到下打印二叉树 III
// 题目链接: https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/
// 题目大意: 请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，
// 第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。

public class 剑指Offer32_III_从上到下打印二叉树III {

    // 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;
    }
}

/**
 * 之字形层序遍历实现
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 使用一个布尔变量记录当前层的打印方向
 * 3. 对于需要从右到左打印的层，将节点值插入到列表的开头
 * 4. 每层处理完后切换打印方向
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 使用队列存储待访问的节点
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

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

    // 当队列不为空时继续遍历
    while (!queue.isEmpty()) {
        // 记录当前层的节点数量
        int size = queue.size();

        // 存储当前层的节点值
        List<Integer> level = new ArrayList<>();
```

```
            // 处理当前层的所有节点
            for (int i = 0; i < size; i++) {
                // 取出队首节点
                TreeNode current = queue.poll();

                // 根据打印方向决定节点值的插入位置
                if (leftToRight) {
                    // 从左到右：添加到列表末尾
                    level.add(current.val);
                } else {
                    // 从右到左：添加到列表开头
                    level.add(0, current.val);
                }

                // 将左右子节点加入队列（如果存在）
                if (current.left != null) {
                    queue.offer(current.left);
                }
                if (current.right != null) {
                    queue.offer(current.right);
                }
            }

            // 将当前层的结果加入最终结果
            result.add(level);

            // 切换打印方向
            leftToRight = !leftToRight;
        }

        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 构建测试二叉树:
        //       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);

        List<List<Integer>> result = levelOrder(root);
        System.out.println("之字形遍历结果:");
        for (List<Integer> level : result) {
            System.out.println(level);
        }

        // 测试空树
        TreeNode emptyRoot = null;
        List<List<Integer>> emptyResult = levelOrder(emptyRoot);
        System.out.println("空树遍历结果: " + emptyResult);
    }
}

/*
Python 实现:

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

class Solution:
    # 方法 2: 双端队列法
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])
        left_to_right = True

        while queue:
            size = len(queue)
            level = deque()

            for i in range(size):
                current = queue.popleft()
```

```
                if left_to_right:
                    level.append(current.val)
                else:
                    level.appendleft(current.val)

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(list(level))
            left_to_right = not left_to_right

        return result
```

C++实现:

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (!root) return result;

        queue<TreeNode*> q;
        q.push(root);
        bool leftToRight = true;

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

```
            deque<int> level;

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

                if (leftToRight) {
                    level.push_back(current->val);
                } else {
                    level.push_front(current->val);
                }

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

            result.push_back(vector<int>(level.begin(), level.end()));
            leftToRight = !leftToRight;
        }

        return result;
    }
};
*/
```

==========

文件: 剑指 Offer32_II_从上到下打印二叉树 II.java

==========

```
package class036;

import java.util.*;

// 剑指 Offer 32-II. 从上到下打印二叉树 II
// 题目链接: https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/
// 题目大意: 从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。

public class 剑指 Offer32_II_从上到下打印二叉树 II {

    // 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;
    }
}

/**
 * 分层层序遍历实现
 * 思路:
 * 1. 使用队列进行层序遍历
 * 2. 记录每一层的节点数量
 * 3. 每次处理完一层的所有节点后，将该层的结果作为一个列表加入最终结果
 * 4. 重复直到队列为空
 * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
 * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
 */
public static List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 使用队列存储待访问的节点
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    // 当队列不为空时继续遍历
    while (!queue.isEmpty()) {
        // 记录当前层的节点数量
        int size = queue.size();

        // 存储当前层的节点值
        List<Integer> level = new ArrayList<>();

        // 处理当前层的所有节点
        for (int i = 0; i < size; i++) {
            // 取出队首节点
            TreeNode current = queue.poll();
```

```
            // 将当前节点的值加入当前层列表
            level.add(current.val);

            // 将左右子节点加入队列（如果存在）
            if (current.left != null) {
                queue.offer(current.left);
            }
            if (current.right != null) {
                queue.offer(current.right);
            }
        }

        // 将当前层的结果加入最终结果
        result.add(level);
    }

    return result;
}

// 测试方法
public static void main(String[] args) {
    // 构建测试二叉树:
    //     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);

    List<List<Integer>> result = levelOrder(root);
    System.out.println("分层遍历结果:");
    for (List<Integer> level : result) {
        System.out.println(level);
    }

    // 测试空树
    TreeNode emptyRoot = null;
    List<List<Integer>> emptyResult = levelOrder(emptyRoot);
    System.out.println("空树遍历结果: " + emptyResult);
```

```
    }
}

/*
Python 实现:

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

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            size = len(queue)
            level = []

            for i in range(size):
                current = queue.popleft()
                level.append(current.val)

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(level)

        return result

C++实现:

#include <iostream>
#include <vector>
#include <queue>
```

```
using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    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* current = q.front();
                q.pop();
                level.push_back(current->val);

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

            result.push_back(level);
        }

        return result;
    }
};
*/
```

==========

文件: 剑指 Offer32_I_从上到下打印二叉树.java

==========

```
package class036;

import java.util.*;

// 剑指 Offer 32-I. 从上到下打印二叉树
// 题目链接: https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/
// 题目大意: 从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。

public class 剑指Offer32_I_从上到下打印二叉树 {

    // 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;
        }
    }

    /**
     * 层序遍历实现（从上到下打印二叉树）
     * 思路:
     * 1. 使用队列进行层序遍历
     * 2. 从根节点开始，将节点加入队列
     * 3. 当队列不为空时，取出队首节点并将其值加入结果列表
     * 4. 将该节点的左右子节点（如果存在）加入队列
     * 5. 重复步骤 3-4 直到队列为空
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    public static int[] levelOrder(TreeNode root) {
        if (root == null) {
            return new int[0];
        }

        // 使用列表存储结果，方便动态添加
        List<Integer> result = new ArrayList<>();
```

```
    // 使用队列存储待访问的节点
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);

    // 当队列不为空时继续遍历
    while (!queue.isEmpty()) {
        // 取出队首节点
        TreeNode current = queue.poll();

        // 将当前节点的值加入结果列表
        result.add(current.val);

        // 将左右子节点加入队列（如果存在）
        if (current.left != null) {
            queue.offer(current.left);
        }
        if (current.right != null) {
            queue.offer(current.right);
        }
    }

    // 将列表转换为数组
    return result.stream().mapToInt(Integer::intValue).toArray();
}

// 测试方法
public static void main(String[] args) {
    // 构建测试二叉树:
    //     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);

    int[] result = levelOrder(root);
    System.out.println("层序遍历结果: " + Arrays.toString(result));

    // 测试空树
```

```
        TreeNode emptyRoot = null;
        int[] emptyResult = levelOrder(emptyRoot);
        System.out.println("空树遍历结果: " + Arrays.toString(emptyResult));
    }
}

/*
Python 实现:

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

class Solution:
    def levelOrder(self, root: TreeNode) -> List[int]:
        if not root:
            return []

        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            current = queue.popleft()
            result.append(current.val)

            if current.left:
                queue.append(current.left)
            if current.right:
                queue.append(current.right)

        return result

C++实现:

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

struct TreeNode {
```

```
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    vector<int> levelOrder(TreeNode* root) {
        vector<int> result;
        if (!root) return result;

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

        while (!q.empty()) {
            TreeNode* current = q.front();
            q.pop();
            result.push_back(current->val);

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

        return result;
    }
};
*/

==============================================

文件: 牛客NC15_求二叉树的层序遍历.java
==============================================
package class036;

import java.util.*;

/**
 * 牛客网 NC15. 求二叉树的层序遍历
 * 题目链接: https://www.nowcoder.com/practice/04a5560e43e24e9db4595865dc9c63a3
 * 题目描述: 给定一个二叉树，返回该二叉树层序遍历的结果（从左到右，一层一层地遍历）
 *
 * 核心算法思想:
```

```
 * 1. 分层层序遍历: 使用队列进行层序遍历，分层收集节点值
 * 2. 数组优化: 使用数组实现队列，提升性能
 *
 * 时间复杂度分析:
 * - 所有方法: O(N)，其中 N 是二叉树中的节点数
 *
 * 空间复杂度分析:
 * - 方法 1(层序遍历): O(W)，W 为树的最大宽度
 * - 方法 2(数组优化): O(W)，W 为树的最大宽度
 *
 * 相关题目:
 * 1. LeetCode 102. 二叉树的层序遍历 - 相同题目
 * 2. 剑指 Offer 32 - II. 从上到下打印二叉树 II - 相同题目
 * 3. LeetCode 107. 二叉树的层序遍历 II - 自底向上遍历
 *
 * 工程化考量:
 * 1. 结果格式: 返回 ArrayList<ArrayList<Integer>>格式
 * 2. 边界处理: 空树返回空列表
 * 3. 性能优化: 预分配内存空间
 */
public class 牛客NC15_求二叉树的层序遍历 {

    // 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;
        }
    }

    /**
     * 层序遍历实现
     * 思路:
     * 1. 使用队列进行层序遍历
     * 2. 从根节点开始，将节点加入队列
     * 3. 当队列不为空时，取出队首节点并将其值加入结果列表
     * 4. 将该节点的左右子节点（如果存在）加入队列
```

```
     * 5. 重复步骤 3-4 直到队列为空
     * 时间复杂度: O(n) - n 是节点数量，每个节点访问一次
     * 空间复杂度: O(w) - w 是树的最大宽度，队列中最多存储一层的节点
     */
    public static ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        // 使用队列存储待访问的节点
        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        // 当队列不为空时继续遍历
        while (!queue.isEmpty()) {
            // 记录当前层的节点数量
            int size = queue.size();

            // 存储当前层的节点值
            ArrayList<Integer> level = new ArrayList<>();

            // 处理当前层的所有节点
            for (int i = 0; i < size; i++) {
                // 取出队首节点
                TreeNode current = queue.poll();

                // 将当前节点的值加入当前层列表
                level.add(current.val);

                // 将左右子节点加入队列（如果存在）
                if (current.left != null) {
                    queue.offer(current.left);
                }
                if (current.right != null) {
                    queue.offer(current.right);
                }
            }

            // 将当前层的结果加入最终结果
            result.add(level);
        }
```

```
        return result;
    }

    // 测试方法
    public static void main(String[] args) {
        // 构建测试二叉树:
        //     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);

        ArrayList<ArrayList<Integer>> result = levelOrder(root);
        System.out.println("层序遍历结果:");
        for (ArrayList<Integer> level : result) {
            System.out.println(level);
        }

        // 测试空树
        TreeNode emptyRoot = null;
        ArrayList<ArrayList<Integer>> emptyResult = levelOrder(emptyRoot);
        System.out.println("空树遍历结果: " + emptyResult);
    }
}

/*
Python 实现:

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

class Solution:
    def levelOrder(self , root: TreeNode) -> List[List[int]]:
        if not root:
            return []
```

```
        from collections import deque
        result = []
        queue = deque([root])

        while queue:
            size = len(queue)
            level = []

            for i in range(size):
                current = queue.popleft()
                level.append(current.val)

                if current.left:
                    queue.append(current.left)
                if current.right:
                    queue.append(current.right)

            result.append(level)

        return result
```

C++实现:

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

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};

class Solution {
public:
    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* current = q.front();
                q.pop();
                level.push_back(current->val);

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

            result.push_back(level);
        }

        return result;
    }
};
*/

==========================================
```