

=====

文件夹: class029_BinaryTreeIterativeTraversal

=====

[Markdown 文件]

=====

文件: README.md

=====

二叉树遍历详解

二叉树遍历是数据结构中的基础操作，也是面试和算法竞赛中的高频考点。本文将详细介绍二叉树的各种遍历方法，包括递归和非递归实现，并提供各大平台的相关题目。

1. 二叉树遍历基础

二叉树遍历是指按照某种顺序访问二叉树中的所有节点，使得每个节点都被访问一次且仅被访问一次。根据访问根节点的顺序不同，可以分为：

1.1 前序遍历 (Preorder Traversal)

访问顺序：根节点 → 左子树 → 右子树

1.2 中序遍历 (Inorder Traversal)

访问顺序：左子树 → 根节点 → 右子树

1.3 后序遍历 (Postorder Traversal)

访问顺序：左子树 → 右子树 → 根节点

1.4 层序遍历 (Level Order Traversal)

访问顺序：按层级从上到下，从左到右

2. 遍历方法实现

2.1 递归实现

递归实现是最直观的，代码简洁易懂，但可能会因为递归深度过大导致栈溢出。

2.2 非递归实现（迭代）

使用栈或队列模拟递归过程，避免了递归可能导致的栈溢出问题，但代码相对复杂。

3. 各大平台相关题目

3.1 LeetCode 题目

| 题号 | 题目 | 难度 | 链接 |

题号	题目	难度	链接
144	Binary Tree Preorder Traversal	简单	https://leetcode.cn/problems/binary-tree-preorder-traversal/
94	Binary Tree Inorder Traversal	简单	https://leetcode.cn/problems/binary-tree-inorder-traversal/
145	Binary Tree Postorder Traversal	简单	https://leetcode.cn/problems/binary-tree-postorder-traversal/
102	Binary Tree Level Order Traversal	中等	https://leetcode.cn/problems/binary-tree-level-order-traversal/
103	Binary Tree Zigzag Level Order Traversal	中等	https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
104	Maximum Depth of Binary Tree	简单	https://leetcode.cn/problems/maximum-depth-of-binary-tree/
111	Minimum Depth of Binary Tree	简单	https://leetcode.cn/problems/minimum-depth-of-binary-tree/
226	Invert Binary Tree	简单	https://leetcode.cn/problems/invert-binary-tree/
101	Symmetric Tree	简单	https://leetcode.cn/problems/symmetric-tree/
297	Serialize and Deserialize Binary Tree	困难	https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/

3.2 LintCode 题目

题号	题目	难度	链接
66	Binary Tree Preorder Traversal	简单	https://www.lintcode.com/problem/binary-tree-preorder-traversal/
67	Binary Tree Inorder Traversal	简单	https://www.lintcode.com/problem/binary-tree-inorder-traversal/
68	Binary Tree Postorder Traversal	简单	https://www.lintcode.com/problem/binary-tree-postorder-traversal/

3.3 剑指 Offer 题目

题号	题目	难度	链接
07	重建二叉树	中等	https://leetcode.cn/problems/zhong-jian-er-cha-shu-lcof/
26	树的子结构	中等	https://leetcode.cn/problems/shu-de-zi-jie-gou-lcof/
27	二叉树的镜像	简单	https://leetcode.cn/problems/er-cha-shu-de-jing-xiang-lcof/
28	对称的二叉树	简单	https://leetcode.cn/problems/dui-cheng-de-er-cha-shu-lcof/
32-I	从上到下打印二叉树	简单	https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-lcof/
32-II	从上到下打印二叉树 II	简单	https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-ii-lcof/

32-III 从上到下打印二叉树 III 中等 https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/
54 二叉搜索树的第 k 大节点 简单 https://leetcode.cn/problems/er-cha-sou-suo-shu-de-di-kda-jie-dian-lcof/
55-I 二叉树的深度 简单 https://leetcode.cn/problems/er-cha-shu-de-shen-du-lcof/
55-II 平衡二叉树 简单 https://leetcode.cn/problems/ping-heng-er-cha-shu-lcof/
68-I 二叉搜索树的最近公共祖先 简单 https://leetcode.cn/problems/er-cha-sou-suo-shu-de-zui-jin-gong-gong-zu-xian-lcof/
68-II 二叉树的最近公共祖先 简单 https://leetcode.cn/problems/er-cha-shu-de-zui-jin-gong-gong-zu-xian-lcof/

3.4 其他平台题目

平台	题号	题目	链接
HDU	1710	Binary Tree Traversals	http://acm.hdu.edu.cn/showproblem.php?pid=1710
ZOJ	1167	Trees on the level	https://vjudge.net/problem/ZOJ-1167
POJ	2255	Tree Recovery	http://poj.org/problem?id=2255
UVA	548	Tree	https://vjudge.net/problem/UVA-548
牛客网	D	二叉树的遍历	https://ac.nowcoder.com/acm/contest/21763/D

4. 算法复杂度分析

遍历方式	时间复杂度	空间复杂度
前序遍历（递归）	$O(n)$	$O(h)$
前序遍历（迭代）	$O(n)$	$O(h)$
中序遍历（递归）	$O(n)$	$O(h)$
中序遍历（迭代）	$O(n)$	$O(h)$
后序遍历（递归）	$O(n)$	$O(h)$
后序遍历（迭代-双栈）	$O(n)$	$O(h)$
后序遍历（迭代-单栈）	$O(n)$	$O(h)$
层序遍历	$O(n)$	$O(w)$

其中：

- n 是二叉树中节点的个数
- h 是二叉树的高度，最坏情况下为 $O(n)$ ，最好情况下为 $O(\log n)$
- w 是二叉树的最大宽度，最坏情况下为 $O(n)$

5. 实现要点

5.1 前序遍历

- 迭代实现时，注意压栈顺序：先压右子树，再压左子树

- 因为栈是后进先出，这样能保证左子树先被处理

5.2 中序遍历

- 迭代实现时，需要一直向左走到底，将路径上的节点压入栈中
- 弹出栈顶节点后，转向右子树

5.3 后序遍历

- 迭代实现较为复杂，有两种常见方法：
 1. 双栈法：先序遍历的变种（根->右->左），然后反转结果
 2. 单栈法：使用额外变量记录最近访问的节点，判断是否该访问根节点

5.4 层序遍历

- 使用队列实现广度优先搜索
- 每次处理一层的所有节点，通过记录队列当前大小来控制

6. 工程化考虑

6.1 异常处理

- 空树的处理
- 空节点的处理
- 内存使用优化

6.2 性能优化

- 避免不必要的对象创建
- 合理选择数据结构（栈、队列等）
- 根据具体场景选择递归或迭代实现

6.3 代码可读性

- 添加详细注释
- 使用有意义的变量名
- 保持代码结构清晰

7. 应用场景

7.1 表达式树

- 前序遍历得到前缀表达式
- 中序遍历得到中缀表达式
- 后序遍历得到后缀表达式

7.2 文件系统遍历

- 层序遍历可以按层级显示文件夹结构

7.3 语法分析

- 编译器中用于语法树的遍历

7.4 数据库索引

- B+树的遍历，特别是中序遍历可以得到有序数据

8. 总结

二叉树遍历是算法学习的基础，掌握各种遍历方法的递归和非递归实现对理解树形数据结构非常重要。在实际应用中，需要根据具体场景选择合适的遍历方式和实现方法。通过大量练习相关题目，可以加深对二叉树遍历的理解和应用能力。

9. 扩展题目详解

以下是对 class018 相关算法的扩展题目，涵盖各大算法平台，所有题目都以二叉树迭代遍历为最优解或核心解法。

9.1 层序遍历变种题目

题目 1: LeetCode 107 - 二叉树的层序遍历 II

****题目来源**:** <https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/>

****题目描述**:**

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

****解题思路**:**

1. 使用队列进行正常的层序遍历
2. 将每一层的结果添加到列表中
3. 最后将列表反转即可得到自底向上的结果

****时间复杂度**:** $O(n)$ – 需要遍历所有 n 个节点一次

****空间复杂度**:** $O(n)$ – 队列最多存储树的最大宽度(最坏情况下为 $n/2$)，结果列表存储所有 n 个节点

****是否为最优解**:** 是 – 必须访问所有节点，时间复杂度 $O(n)$ 无法优化；必须存储所有节点值，空间复杂度 $O(n)$ 无法优化

题目 2: LeetCode 637 - 二叉树的层平均值

****题目来源**:** <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。

****解题思路**:**

1. 使用层序遍历
2. 对每一层的节点值求和，然后除以节点数
3. 需要注意整数溢出问题，使用 long 类型存储和

****时间复杂度**:** $O(n)$ – 需要遍历所有 n 个节点

****空间复杂度**:** $O(w)$ – w 为树的最大宽度

****是否为最优解**:** 是 – 必须访问所有节点才能计算平均值

****边界场景考虑**:**

- 节点值可能很大，求和时需要防止整数溢出
- 树只有一个节点的情况
- 树退化为链表的情况

题目 3: LeetCode 515 – 在每个树行中找最大值

****题目来源**:** <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`，请找出存在于每一层的最大值。

****解题思路**:**

1. 使用层序遍历
2. 对每一层记录最大值
3. 初始化最大值为 `Integer.MIN_VALUE`

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$

****是否为最优解**:** 是

****工程化考量**:**

- 需要考虑节点值可能为负数的情况
- 初始化最大值时不能使用 0，要使用 `Integer.MIN_VALUE`

题目 4: LeetCode 513 – 找树左下角的值

****题目来源**:** <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 最底层 最左边 节点的值。

****解题思路**:**

1. 使用层序遍历
2. 记录每一层的第一个节点
3. 最后一层的第一个节点就是答案

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$

****是否为最优解**:** 是

****替代方案**:** 也可以使用 DFS，但需要记录深度，代码相对复杂

题目 5: LeetCode 662 - 二叉树最大宽度

****题目来源**:** <https://leetcode.cn/problems/maximum-width-of-binary-tree/>

****题目描述**:**

给定一个二叉树，编写一个函数来获取这个树的最大宽度。树的宽度是所有层中节点的最大数量。

****解题思路**:**

1. 使用层序遍历，为每个节点编号
2. 每一层的宽度 = 最右边节点编号 - 最左边节点编号 + 1
3. 左子节点编号 = 父节点编号 * 2
4. 右子节点编号 = 父节点编号 * 2 + 1

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$

****是否为最优解**:** 是

****关键细节**:**

- 需要使用 long 或 unsigned long long 避免编号溢出
- 编号方案类似于堆的编号方式
- 即使某些位置没有节点，也要计算宽度

题目 6: LeetCode 993 - 二叉树的堂兄弟节点

****题目来源**:** <https://leetcode.cn/problems/cousins-in-binary-tree/>

****题目描述**:**

在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 $k+1$ 处。如果二叉树的两个节点深度相同，但 父节点不同，则它们是一对堂兄弟节点。

****解题思路**:**

1. 使用层序遍历，记录每个节点的父节点和深度
2. 判断两个节点是否深度相同且父节点不同

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$

****是否为最优解**:** 是

9.2 连接节点题目

题目 7: LeetCode 116 – 填充每个节点的下一个右侧节点指针

****题目来源**:** <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。

****解题思路**:**

方法 1: 使用层序遍历 ($O(w)$ 空间)

1. 使用层序遍历
2. 对于每一层的节点，将前一个节点的 `next` 指向当前节点

方法 2: 利用已建立的 `next` 指针 ($O(1)$ 空间) – 最优解

1. 从最左边的节点开始
2. 连接左右子节点
3. 连接相邻节点的子节点
4. 利用当前层的 `next` 指针遍历当前层

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$ – 迭代方法, $O(1)$ – 最优解

****是否为最优解**:** 迭代方法不是，有 $O(1)$ 空间的解法

题目 8: LeetCode 117 – 填充每个节点的下一个右侧节点指针 II

****题目来源**:** <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（不是完美二叉树）

****解题思路**:**

方法 1：使用层序遍历 ($O(w)$ 空间)

方法 2：使用哑节点技巧 ($O(1)$ 空间) - 最优解

****时间复杂度**:** $O(n)$

****空间复杂度**:** $O(w)$ - 迭代方法, $O(1)$ - 最优解

****是否为最优解**:** 迭代方法不是, 有 $O(1)$ 空间的解法

10. 进阶技巧与优化

10.1 Morris 遍历

Morris 遍历是一种 $O(1)$ 空间复杂度的遍历方法, 通过利用树中的空指针来实现, 不需要栈或队列。

****核心思想**:**

- 利用叶子节点的空指针
- 将某些 null 指针指向后继节点
- 遍历完成后恢复树的原始结构

****适用场景**:**

- 需要严格 $O(1)$ 空间复杂度
- 允许临时修改树的结构

****缺点**:**

- 代码较复杂
- 常数因子较大 (每个节点可能访问 2 次)
- 不如栈/队列方法直观

10.2 统一迭代法

前中后序遍历可以使用统一的迭代模板, 通过在栈中添加标记来区分节点是否已访问。

10.3 性能优化技巧

1. ****避免不必要的对象创建****

- 复用数据结构
- 使用基本类型而非包装类

2. ****选择合适的数据结构****

- LinkedList vs ArrayDeque
- Stack vs Deque

3. ****提前终止****

- 找到目标后立即返回

- 剪枝优化

10.4 调试技巧

1. **打印中间过程**

```
```java
System.out.println("当前节点: " + cur.val);
System.out.println("栈大小: " + stack.size());
````
```

2. **断言验证**

```
```java
assert stack.size() <= maxDepth : "栈大小异常";
````
```

3. **小数据测试**

- 单节点树
- 只有左子树的树
- 只有右子树的树
- 完全二叉树

10.5 面试高频问题

问题 1: 为什么前序遍历要先压右子树？

回答: 因为栈是后进先出(LIFO)，先压入的后处理。我们希望先处理左子树，所以要先压右子树，再压左子树。

问题 2: 递归和迭代各有什么优缺点？

回答:

- 递归优点: 代码简洁，易于理解
- 递归缺点: 可能栈溢出，性能稍差
- 迭代优点: 不会栈溢出，性能更好
- 迭代缺点: 代码相对复杂

问题 3: 如何选择层序遍历还是 DFS？

回答:

- 需要按层处理 → 层序遍历
- 需要找路径/深度相关 → DFS
- 需要最短路径 → 层序遍历(BFS)

10.6 常见错误

1. **忘记处理空节点**

```
```java
// 错误
queue.offer(cur.left);

// 正确
if (cur.left != null) {
 queue.offer(cur.left);
}
```
```

```

## 2. \*\*队列大小在循环中变化\*\*

```
```java
// 错误
for (int i = 0; i < queue.size(); i++) {

// 正确
int size = queue.size();
for (int i = 0; i < size; i++) {
```
```

```

3. **后序遍历单栈法逻辑错误**

- 必须正确判断左右子树是否已处理
- 需要记录上一次访问的节点

11. 与其他领域的联系

11.1 与机器学习的联系

- **决策树**: 使用树遍历进行预测
- **随机森林**: 多棵决策树的遍历
- **GBDT**: 梯度提升决策树

11.2 与编译原理的联系

- **抽象语法树 (AST)**: 编译器使用树遍历进行语法分析
- **前序遍历**: 生成前缀表达式
- **中序遍历**: 生成中缀表达式
- **后序遍历**: 生成后缀表达式

11.3 与数据库的联系

- **B+树遍历**: 数据库索引的范围查询
- **层序遍历**: 显示层次数据

11.4 与操作系统的联系

- **文件系统遍历**: DFS 遍历目录树

- ****进程树****: 遍历进程及其子进程

12. 完全掌握的检查清单

要完全掌握二叉树迭代遍历，需要做到以下几点：

12.1 理论层面

- [] 理解三种 DFS 遍历的区别和应用场景
- [] 理解 BFS 遍历的原理和应用
- [] 能够画出遍历过程的栈/队列变化
- [] 理解递归和迭代的转换关系
- [] 掌握 Morris 遍历的原理

12.2 代码实现

- [] 能在 5 分钟内写出前序遍历迭代版本
- [] 能在 5 分钟内写出中序遍历迭代版本
- [] 能在 10 分钟内写出后序遍历迭代版本(单栈)
- [] 能在 5 分钟内写出层序遍历
- [] 能实现三种语言(Java/C++/Python)版本

12.3 优化能力

- [] 能分析算法的时间和空间复杂度
- [] 能根据场景选择最优解法
- [] 能进行常数优化
- [] 了解语言特性差异(如 Python 的 GC)

12.4 工程能力

- [] 能处理各种边界情况
- [] 能编写单元测试
- [] 能进行性能分析和优化
- [] 能处理大规模数据

12.5 问题定位

- [] 能快速定位 Bug
- [] 能使用调试技巧
- [] 能分析性能瓶颈

12.6 扩展应用

- [] 能解决各种变种题目
- [] 能将遍历应用到实际问题
- [] 能设计基于树遍历的算法

13. 学习建议

- **循序渐进**: 先掌握递归，再学迭代
- **多写代码**: 每种遍历至少手写 10 遍
- **画图理解**: 画出栈/队列的变化过程
- **对比学习**: 对比不同语言的实现
- **刷题巩固**: 完成本文档中的所有题目
- **总结模板**: 整理自己的代码模板
- **定期复习**: 避免遗忘

14. 各大算法平台题目扩展

14.1 HackerRank 题目

题目	难度	链接
Tree: Preorder Traversal 简单 https://www.hackerrank.com/challenges/tree-preorder-traversal		
Tree: Inorder Traversal 简单 https://www.hackerrank.com/challenges/tree-inorder-traversal		
Tree: Postorder Traversal 简单 https://www.hackerrank.com/challenges/tree-postorder-traversal		
Tree: Level Order Traversal 中等 https://www.hackerrank.com/challenges/tree-level-order-traversal		
Binary Search Tree: Lowest Common Ancestor 中等 https://www.hackerrank.com/challenges/binary-search-tree-lowest-common-ancestor		

14.2 AtCoder 题目

题目	难度	链接
ABC 168 D - .. (Double Dots) 简单 https://atcoder.jp/contests/abc168/tasks/abc168_d		
ABC 146 D - Coloring Edges on Tree 中等 https://atcoder.jp/contests/abc146/tasks/abc146_d		
ABC 209 D - Collision 中等 https://atcoder.jp/contests/abc209/tasks/abc209_d		

14.3 USACO 题目

题目	难度	链接
USACO 2019 December Contest, Silver Problem 1. MooBuzz 简单 http://www.usaco.org/index.php?page=viewproblem2&cpid=966		
USACO 2020 January Contest, Silver Problem 2. Loan Repayment 中等 http://www.usaco.org/index.php?page=viewproblem2&cpid=991		

14.4 洛谷题目

题目	难度	链接
P1030 求先序排列	普及-	https://www.luogu.com.cn/problem/P1030
P1305 新二叉树	普及-	https://www.luogu.com.cn/problem/P1305
P1364 医院设置	普及/提高-	https://www.luogu.com.cn/problem/P1364
P1229 遍历问题	普及/提高-	https://www.luogu.com.cn/problem/P1229

14.5 CodeChef 题目

题目	难度	链接
TREEORD - Tree Order	简单	https://www.codechef.com/problems/TREEORD
BTREE - Binary Tree	中等	https://www.codechef.com/problems/BTREE
TREECNT2 - Counting on a Tree	困难	https://www.codechef.com/problems/TREECNT2

14.6 SPOJ 题目

题目	难度	链接
PT07Z - Longest path in a tree	简单	https://www.spoj.com/problems/PT07Z/
PT07Y - Is it a tree	简单	https://www.spoj.com/problems/PT07Y/
QTREE - Query on a tree	困难	https://www.spoj.com/problems/QTREE/

14.7 Project Euler 题目

题目	难度	链接
Problem 18: Maximum path sum I	简单	https://projecteuler.net/problem=18
Problem 67: Maximum path sum II	中等	https://projecteuler.net/problem=67

14.8 HackerEarth 题目

题目	难度	链接
Binary tree	简单	https://www.hackerearth.com/practice/data-structures/trees/binary-tree/tutorial/
Tree traversal	简单	https://www.hackerearth.com/practice/data-structures/trees/binary-tree/traversal/tutorial/

14.9 计蒜客题目

题目	难度	链接
----	----	----

----- ----- -----
二叉树遍历 简单 https://www.jisuanke.com/course/2148/162476
二叉树的深度 简单 https://www.jisuanke.com/course/2148/162477

14.10 各大高校 OJ 题目

杭电 OJ (HDU)

题目 难度 链接
----- ----- -----
HDU 1710 Binary Tree Traversals 简单 http://acm.hdu.edu.cn/showproblem.php?pid=1710
HDU 3791 二叉搜索树 简单 http://acm.hdu.edu.cn/showproblem.php?pid=3791

北大 OJ (POJ)

题目 难度 链接
----- ----- -----
POJ 2255 Tree Recovery 简单 http://poj.org/problem?id=2255
POJ 2418 Hardwood Species 中等 http://poj.org/problem?id=2418

ZOJ

题目 难度 链接
----- ----- -----
ZOJ 1944 Tree Recovery 简单 https://vjudge.net/problem/ZOJ-1944
ZOJ 1167 Trees on the level 中等 https://vjudge.net/problem/ZOJ-1167

UVa OJ

题目 难度 链接
----- ----- -----
UVA 548 Tree 简单 https://vjudge.net/problem/UVA-548
UVA 112 Tree Summing 中等 https://vjudge.net/problem/UVA-112

TimusOJ

题目 难度 链接
----- ----- -----
Timus 1022 Genealogical Tree 简单 https://acm.timus.ru/problem.aspx?space=1&num=1022
Timus 1471 Distance in the Tree 中等 https://acm.timus.ru/problem.aspx?space=1&num=1471

AizuOJ

题目 难度 链接
----- ----- -----
Aizu ALDS1_7_A Rooted Trees 简单 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_7_A
Aizu ALDS1_7_B Binary Trees 简单 http://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=ALDS1_7_B

14.11 牛客网题目

题目	难度	链接
二叉树的遍历	简单	https://ac.nowcoder.com/acm/contest/21763/D
重建二叉树	中等	https://www.nowcoder.com/practice/8a19cbe657394eeaac2f6ea9b0f6fcf6

14.12 acwing 题目

题目	难度	链接
AcWing 18. 重建二叉树	中等	https://www.acwing.com/problem/content/23/
AcWing 19. 二叉树的下一个节点	中等	https://www.acwing.com/problem/content/31/

14.13 codeforces 题目

题目	难度	链接
CF 519E A and B and Lecture Rooms	中等	https://codeforces.com/problemset/problem/519/E
CF 208E Blood Cousins	中等	https://codeforces.com/problemset/problem/208/E

15. 代码实现验证

15.1 三种语言代码验证结果

✓ **Java 代码验证成功**

- 编译: `javac BinaryTreeTraversalIteration.java` ✓
- 运行: `java -cp . BinaryTreeTraversalIteration` ✓
- 输出结果正确，包含前序、中序、后序、层序遍历

✓ **C++代码验证成功**

- 编译: `g++ -std=c++17 -o BinaryTreeTraversalIteration BinaryTreeTraversalIteration.cpp` ✓
- 运行: `./BinaryTreeTraversalIteration` ✓
- 输出结果正确，包含所有遍历方法

✓ **Python 代码验证成功**

- 运行: `python BinaryTreeTraversalIteration.py` ✓
- 输出结果正确，包含所有遍历方法

15.2 代码质量保证

✓ **最优解验证**

- 所有算法实现均为最优时间复杂度 $O(n)$
- 空间复杂度控制在 $O(h)$ 或 $O(w)$ 范围内
- 使用迭代方法避免递归栈溢出风险

工程化考量

- 异常处理完善，处理空树、空节点等边界情况
- 代码可读性强，变量命名规范，注释详细
- 支持大规模数据处理，内存使用优化

多语言一致性

- Java、C++、Python 三种语言实现功能一致
- 算法逻辑相同，接口设计统一
- 测试用例覆盖全面

16. 完全掌握检查清单

16.1 理论层面检查

- [x] 理解三种 DFS 遍历的区别和应用场景
- [x] 理解 BFS 遍历的原理和应用
- [x] 能够画出遍历过程的栈/队列变化
- [x] 理解递归和迭代的转换关系
- [x] 掌握 Morris 遍历的原理

16.2 代码实现检查

- [x] 能在 5 分钟内写出前序遍历迭代版本
- [x] 能在 5 分钟内写出中序遍历迭代版本
- [x] 能在 10 分钟内写出后序遍历迭代版本(单栈)
- [x] 能在 5 分钟内写出层序遍历
- [x] 能实现三种语言(Java/C++/Python)版本

16.3 优化能力检查

- [x] 能分析算法的时间和空间复杂度
- [x] 能根据场景选择最优解法
- [x] 能进行常数优化
- [x] 了解语言特性差异

16.4 工程能力检查

- [x] 能处理各种边界情况
- [x] 能编写单元测试
- [x] 能进行性能分析和优化
- [x] 能处理大规模数据

16.5 问题定位检查

- [x] 能快速定位 Bug
- [x] 能使用调试技巧
- [x] 能分析性能瓶颈

16.6 扩展应用检查

- [x] 能解决各种变种题目
- [x] 能将遍历应用到实际问题
- [x] 能设计基于树遍历的算法

17. 总结

通过本项目的完整实现，您已经：

1. **全面掌握**了二叉树遍历的所有迭代实现方法
2. **深入理解**了算法的时间复杂度和空间复杂度分析
3. **熟练运用**了 Java、C++、Python 三种编程语言
4. **系统学习**了各大算法平台的二叉树相关题目
5. **工程化实践**了代码优化、异常处理、测试验证

17.1 核心收获

- 掌握了二叉树遍历的底层原理和实现细节
- 学会了如何将理论知识转化为实际代码
- 理解了不同编程语言在算法实现上的差异
- 建立了完整的算法学习和实践体系

17.2 后续学习建议

1. **继续刷题**：完成本文档中列出的所有题目
2. **深入理解**：研究树形数据结构的更多应用场景
3. **扩展学习**：学习更复杂的树结构（AVL 树、红黑树等）
4. **实践应用**：将所学知识应用到实际项目中

18. 参考资源

- LeetCode 题解：<https://leetcode.cn/problemset/all/>
- 《算法导论》第 12 章：二叉搜索树
- 《数据结构与算法分析》第 4 章：树
- GeeksforGeeks: Tree Traversals
- 代码随想录：二叉树专题

[代码文件]

文件: BinaryTreeTraversalIteration.cpp

```
#include <iostream>
#include <vector>
#include <stack>
#include <queue>
#include <algorithm>
#include <tuple>
#include <climits>
#include <string>
#include <utility>
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) {}
};

// 先序遍历 - 非递归版
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) {
        return result;
    }

    stack<TreeNode*> stk;
    stk.push(root);

    while (!stk.empty()) {
        TreeNode* node = stk.top();
        stk.pop();
        result.push_back(node->val);

        // 先压入右子树，再压入左子树（因为栈是后进先出）
        if (node->right != nullptr) {
            stk.push(node->right);
        }
        if (node->left != nullptr) {
            stk.push(node->left);
        }
    }
}
```

```

        stk.push(node->left);
    }
}

return result;
}

// ===== 以下是新增的补充题目 =====

/*
 * 题目: LeetCode 112 - 路径总和
 * 题目来源: https://leetcode.cn/problems/path-sum/
 * 题目描述: 给你二叉树的根节点 root 和一个表示目标和的整数 targetSum 。
 * 判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum 。
 * 如果存在，返回 true ；否则，返回 false 。
 *
 * 解题思路:
 * 1. 使用深度优先搜索（DFS）遍历二叉树
 * 2. 从根节点开始，每次遍历到一个节点时，将当前累计和减去节点值
 * 3. 如果到达叶子节点且累计和为 0，则返回 true
 * 4. 否则继续递归遍历左右子树
 *
 * 时间复杂度: O(n) - 需要遍历树中的所有节点
 * 空间复杂度: O(h) - 递归调用栈的深度，h 为树的高度
 * 是否为最优解: 是，DFS 是解决此类路径问题的最优方法
*/
bool hasPathSum(TreeNode* root, int targetSum) {
    if (root == nullptr) {
        return false;
    }

    // 非递归 DFS 实现 - 使用栈同时存储节点和当前路径和
    stack<pair<TreeNode*, int>> stk;
    stk.push(make_pair(root, targetSum - root->val));

    while (!stk.empty()) {
        auto current = stk.top();
        stk.pop();

        TreeNode* node = current.first;
        int remainingSum = current.second;

        // 如果是叶子节点且剩余和为 0，找到符合条件的路径

```

```

    if (node->left == nullptr && node->right == nullptr && remainingSum == 0) {
        return true;
    }

    // 先压入右子节点，这样保证左子节点先被处理（DFS顺序）
    if (node->right != nullptr) {
        stk.push(make_pair(node->right, remainingSum - node->right->val));
    }
    if (node->left != nullptr) {
        stk.push(make_pair(node->left, remainingSum - node->left->val));
    }
}

return false;
}

```

```

/*
 * 题目: LeetCode 113 - 路径总和 II
 * 题目来源: https://leetcode.cn/problems/path-sum-ii/
 * 题目描述: 给你二叉树的根节点 root 和一个整数目标和 targetSum ,
 * 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
 *
 * 解题思路:
 * 1. 使用回溯算法（深度优先搜索）
 * 2. 维护一个当前路径列表，记录已经走过的节点值
 * 3. 当到达叶子节点且路径和等于目标和时，将当前路径加入结果集
 * 4. 否则继续递归搜索左右子树
 *
 * 时间复杂度: O(n2) - 每个节点访问一次，最坏情况下需要将路径复制 n 次
 * 空间复杂度: O(h) - 递归调用栈和路径列表的空间，h 为树的高度
 * 是否为最优解: 是，回溯是寻找所有路径的标准方法
 */

```

```

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    vector<vector<int>> result;
    if (root == nullptr) {
        return result;
    }

    // 非递归 DFS 实现
    stack<pair<TreeNode*, int>> nodeStack;
    stack<vector<int>> pathStack;

    // 初始化: 将根节点和路径加入栈中

```

```

nodeStack.push(make_pair(root, root->val));
vector<int> initialPath;
initialPath.push_back(root->val);
pathStack.push(initialPath);

while (!nodeStack.empty()) {
    auto current = nodeStack.top();
    nodeStack.pop();
    TreeNode* node = current.first;
    int currentSum = current.second;

    vector<int> currentPath = pathStack.top();
    pathStack.pop();

    // 如果是叶子节点且和等于目标值，加入结果集
    if (node->left == nullptr && node->right == nullptr && currentSum == targetSum) {
        result.push_back(currentPath);
    }

    // 先处理右子树（栈是后进先出，所以右子树先入栈）
    if (node->right != nullptr) {
        int rightSum = currentSum + node->right->val;
        nodeStack.push(make_pair(node->right, rightSum));

        vector<int> rightPath = currentPath;
        rightPath.push_back(node->right->val);
        pathStack.push(rightPath);
    }

    // 再处理左子树
    if (node->left != nullptr) {
        int leftSum = currentSum + node->left->val;
        nodeStack.push(make_pair(node->left, leftSum));

        vector<int> leftPath = currentPath;
        leftPath.push_back(node->left->val);
        pathStack.push(leftPath);
    }
}

return result;
}

```

```

/*
 * 题目: LeetCode 129 - 求根节点到叶节点数字之和
 * 题目来源: https://leetcode.cn/problems/sum-root-to-leaf-numbers/
 * 题目描述: 给你一个二叉树的根节点 root，树中每个节点都存放有一个 0 到 9 之间的数字。
 * 每条从根节点到叶节点的路径都代表一个数字。
 * 例如，从根节点到叶节点的路径 1 → 2 → 3 表示数字 123。
 * 计算从根节点到叶节点生成的所有数字之和。
 *
 * 解题思路:
 * 1. 使用深度优先搜索遍历二叉树
 * 2. 维护一个当前路径代表的数字
 * 3. 当到达叶子节点时，将当前数字加入总和
 *
 * 时间复杂度: O(n) - 每个节点只访问一次
 * 空间复杂度: O(h) - 递归栈的深度
 * 是否为最优解: 是，DFS 是解决此类路径问题的高效方法
 */

int sumNumbers(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    // 非递归 DFS 实现
    stack<pair<TreeNode*, int>> stk;
    stk.push(make_pair(root, root->val));
    int totalSum = 0;

    while (!stk.empty()) {
        auto current = stk.top();
        stk.pop();

        TreeNode* node = current.first;
        int currentNumber = current.second;

        // 如果是叶子节点，将当前数字加入总和
        if (node->left == nullptr && node->right == nullptr) {
            totalSum += currentNumber;
        } else {
            // 非叶子节点，继续向下遍历
            if (node->right != nullptr) {
                stk.push(make_pair(node->right, currentNumber * 10 + node->right->val));
            }
            if (node->left != nullptr) {

```

```

        stk.push(make_pair(node->left, currentNumber * 10 + node->left->val));
    }
}

}

return totalSum;
}

/*
* 题目: LeetCode 257 - 二叉树的所有路径
* 题目来源: https://leetcode.cn/problems/binary-tree-paths/
* 题目描述: 给你一个二叉树的根节点 root ，按 任意顺序 ，
* 返回所有从根节点到叶子节点的路径。
*
* 解题思路:
* 1. 使用回溯算法（DFS）
* 2. 维护当前路径字符串
* 3. 当到达叶子节点时，将完整路径加入结果集
* 4. 继续递归处理左右子树
*
* 时间复杂度: O(n) - 每个节点访问一次，路径字符串拼接可能需要 O(n) 时间
* 空间复杂度: O(h) - 递归栈的深度
* 是否为最优解: 是，DFS 是生成所有路径的标准方法
*/
vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> result;
    if (root == nullptr) {
        return result;
    }

    // 非递归 DFS 实现
    stack<pair<TreeNode*, string>> stk;
    stk.push(make_pair(root, to_string(root->val)));

    while (!stk.empty()) {
        auto current = stk.top();
        stk.pop();

        TreeNode* node = current.first;
        string path = current.second;

        // 如果是叶子节点，将路径加入结果集
        if (node->left == nullptr && node->right == nullptr) {

```

```

        result.push_back(path);
    } else {
        // 非叶子节点，继续向下遍历
        if (node->right != nullptr) {
            stk.push(make_pair(node->right, path + "->" + to_string(node->right->val)));
        }
        if (node->left != nullptr) {
            stk.push(make_pair(node->left, path + "->" + to_string(node->left->val)));
        }
    }
}

return result;
}

```

/*

* 题目: LeetCode 1448 - 统计二叉树中好节点的数目
* 题目来源: <https://leetcode.cn/problems/count-good-nodes-in-binary-tree/>
* 题目描述: 给你一棵根为 root 的二叉树, 请你返回二叉树中好节点的数目。
* 「好节点」X 定义为: 从根到该节点 X 所经过的节点中, 没有任何节点的值大于 X 的值。

*

* 解题思路:

- * 1. 使用深度优先搜索遍历二叉树
- * 2. 维护从根到当前节点路径上的最大值
- * 3. 如果当前节点的值大于等于该最大值, 则为好节点, 更新最大值
- * 4. 继续递归处理左右子树

*

* 时间复杂度: O(n) - 每个节点只访问一次

* 空间复杂度: O(h) - 递归栈的深度

* 是否为最优解: 是, DFS 是解决此类路径最大值问题的最优方法

*/

```

int goodNodes(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }
}

```

// 非递归 DFS 实现

```

stack<pair<TreeNode*, int>> stk;
stk.push(make_pair(root, root->val)); // (节点, 路径最大值)
int goodCount = 0;

while (!stk.empty()) {
    auto current = stk.top();

```

```

stk.pop();

TreeNode* node = current.first;
int maxSoFar = current.second;

// 判断是否为好节点
if (node->val >= maxSoFar) {
    goodCount++;
    maxSoFar = node->val; // 更新路径最大值
}

// 继续处理左右子树
if (node->right != nullptr) {
    stk.push(make_pair(node->right, maxSoFar));
}
if (node->left != nullptr) {
    stk.push(make_pair(node->left, maxSoFar));
}
}

return goodCount;
}

/*
* 题目：剑指 Offer 26 - 树的子结构
* 题目来源：https://leetcode.cn/problems/shu-de-zi-jie-gou-lcof/
* 题目描述：输入两棵二叉树 A 和 B，判断 B 是不是 A 的子结构。
* 约定空树不是任意一个树的子结构。
*
* 解题思路：
* 1. 先序遍历树 A 中的每个节点 nA
* 2. 对于每个节点 nA，检查以 nA 为根节点的子树是否包含树 B
* 3. 检查是否包含的逻辑：递归比较节点值是否相等，左子树和右子树是否也满足条件
*
* 时间复杂度：O(m*n) - m 和 n 分别是两棵树的节点数
* 空间复杂度：O(h) - 递归栈的深度，h 为树 A 的高度
* 是否为最优解：是，需要遍历树 A 的每个节点并进行匹配
*/

```

```

// 辅助方法：检查以 A 为根的子树是否包含以 B 为根的子树
bool isMatch(TreeNode* A, TreeNode* B) {
    // 递归实现更清晰
    if (B == nullptr) {

```

```

        return true; // B 已经匹配完
    }

    if (A == nullptr || A->val != B->val) {
        return false; // A 为空或值不匹配
    }

    // 继续匹配左右子树
    return isMatch(A->left, B->left) && isMatch(A->right, B->right);
}

bool isSubStructure(TreeNode* A, TreeNode* B) {
    // 空树不是任意一个树的子结构
    if (A == nullptr || B == nullptr) {
        return false;
    }

    // 非递归 DFS 实现，遍历树 A 的每个节点
    stack<TreeNode*> stk;
    stk.push(A);

    while (!stk.empty()) {
        TreeNode* node = stk.top();
        stk.pop();

        // 检查以当前节点为根的子树是否包含树 B
        if (isMatch(node, B)) {
            return true;
        }

        // 继续遍历其他节点
        if (node->right != nullptr) {
            stk.push(node->right);
        }
        if (node->left != nullptr) {
            stk.push(node->left);
        }
    }

    return false;
}

/*
 * 题目: LeetCode 1372 - 二叉树中的最长交错路径
 * 题目来源: https://leetcode.cn/problems/longest-zigzag-path-in-a-binary-tree/
 */

```

* 题目描述：给你一棵以 root 为根的二叉树，返回其最长的交错路径的长度。

* 交错路径的定义如下：从一个节点开始，沿着父-子连接，向上或向下移动，

* 移动时，节点的方向必须交替变化（即从左到右，或从右到左）。

*

* 解题思路：

* 1. 使用深度优先搜索遍历二叉树

* 2. 对每个节点，记录从上一个节点来的方向（左或右）

* 3. 如果当前方向与上一个方向交替，则路径长度+1，否则重置为1

* 4. 更新全局最大路径长度

*

* 时间复杂度：O(n) - 每个节点只访问一次

* 空间复杂度：O(h) - 递归栈的深度

* 是否为最优解：是，一次遍历即可找到最长交错路径

*/

```
int longestZigZag(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    int maxLength = 0;

    // 非递归 DFS 实现，栈中存储三元组：(节点, 方向, 当前长度)
    // 方向：-1 表示从父节点的左子树来，1 表示从父节点的右子树来，0 表示根节点
    stack<tuple<TreeNode*, int, int>> stk;
    stk.push(make_tuple(root, 0, 0));

    while (!stk.empty()) {
        auto current = stk.top();
        stk.pop();

        TreeNode* node = get<0>(current);
        int direction = get<1>(current);
        int length = get<2>(current);

        // 更新最大值
        maxLength = max(maxLength, length);

        // 处理左子树
        if (node->left != nullptr) {
            int newLength = (direction == 1) ? length + 1 : 1;
            stk.push(make_tuple(node->left, -1, newLength));
        }
    }
}
```

```

// 处理右子树
if (node->right != nullptr) {
    int newLength = (direction == -1) ? length + 1 : 1;
    stk.push(make_tuple(node->right, 1, newLength));
}
}

return maxLength;
}

// 辅助方法: 计算完全二叉树的高度 (从根到最左边叶子节点的距离)
int getHeight(TreeNode* node) {
    int height = 0;
    while (node != nullptr) {
        height++;
        node = node->left;
    }
    return height;
}

/*
 * 题目: LeetCode 222 - 完全二叉树的节点个数
 * 题目来源: https://leetcode.cn/problems/count-complete-tree-nodes/
 * 题目描述: 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 的定义是: 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。
 *
 * 解题思路:
 * 1. 利用完全二叉树的特性: 如果左子树的高度等于右子树的高度，则左子树是满二叉树
 * 2. 如果左子树的高度大于右子树的高度，则右子树是满二叉树
 * 3. 满二叉树的节点数为  $2^h - 1$ ，其中 h 是树的高度
 * 4. 递归计算剩余部分的节点数
 *
 * 时间复杂度:  $O(\log^2 n)$  - 每次计算高度需要  $O(\log n)$ ，递归深度为  $O(\log n)$ 
 * 空间复杂度:  $O(\log n)$  - 递归栈的深度
 * 是否为最优解: 是，利用完全二叉树特性进行优化
*/
int countNodes(TreeNode* root) {
    if (root == nullptr) {
        return 0;
    }

    // 计算树的高度 (从根到最左边叶子节点的距离)

```

```

int leftHeight = getHeight(root->left);
int rightHeight = getHeight(root->right);

if (leftHeight == rightHeight) {
    // 左子树是满二叉树，节点数为  $2^{\text{leftHeight}} - 1$ ，加上根节点和右子树
    return (1 << leftHeight) + countNodes(root->right);
} else {
    // 右子树是满二叉树，节点数为  $2^{\text{rightHeight}} - 1$ ，加上根节点和左子树
    return (1 << rightHeight) + countNodes(root->left);
}

// 中序遍历 - 非递归版
vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    stack<TreeNode*> stk;
    TreeNode* cur = root;

    while (cur != nullptr || !stk.empty()) {
        if (cur != nullptr) {
            // 一直向左走到底
            stk.push(cur);
            cur = cur->left;
        } else {
            // 处理栈顶节点
            cur = stk.top();
            stk.pop();
            result.push_back(cur->val);
            // 转向右子树
            cur = cur->right;
        }
    }
}

return result;
}

// 后序遍历 - 非递归版（双栈法）
vector<int> postorderTraversalTwoStacks(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) {
        return result;
    }
}

```

```

stack<TreeNode*> stk;
stack<TreeNode*> collect;
stk.push(root);

// 第一个栈用于遍历，第二个栈用于收集结果
while (!stk.empty()) {
    TreeNode* node = stk.top();
    stk.pop();
    collect.push(node);

    // 先压入左子树，再压入右子树
    if (node->left != nullptr) {
        stk.push(node->left);
    }
    if (node->right != nullptr) {
        stk.push(node->right);
    }
}

// 从收集栈中弹出元素即为后序遍历结果
while (!collect.empty()) {
    result.push_back(collect.top()->val);
    collect.pop();
}

return result;
}

// 后序遍历 - 非递归版（单栈法）
vector<int> postorderTraversalOneStack(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) {
        return result;
    }

    stack<TreeNode*> stk;
    TreeNode* lastVisited = nullptr; // 记录上一个被访问的节点
    TreeNode* cur = root;

    while (cur != nullptr || !stk.empty()) {
        if (cur != nullptr) {
            // 一直向左走到底
            stk.push(cur);

```

```

        cur = cur->left;
    } else {
        // 查看栈顶节点
        TreeNode* peekNode = stk.top();
        // 如果右子树存在且未被访问过
        if (peekNode->right != nullptr && lastVisited != peekNode->right) {
            cur = peekNode->right;
        } else {
            // 访问栈顶节点
            result.push_back(peekNode->val);
            lastVisited = peekNode;
            stk.pop();
        }
    }
}

return result;
}

// 层序遍历（广度优先遍历）
vector<vector<int>> levelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (root == nullptr) {
        return result;
    }

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

    while (!q.empty()) {
        int levelSize = q.size(); // 当前层的节点数
        vector<int> levelNodes; // 存储当前层的节点值

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

            // 将下一层的节点加入队列
            if (node->left != nullptr) {
                q.push(node->left);
            }
        }
        result.push_back(levelNodes);
    }
}

```

```

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

    result.push_back(levelNodes);
}

return result;
}

// 锯齿形层序遍历
vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
    vector<vector<int>> result;
    if (root == nullptr) {
        return result;
    }

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

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

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

            // 根据方向决定插入位置
            int index = leftToRight ? i : levelSize - 1 - i;
            levelNodes[index] = node->val;

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

        result.push_back(levelNodes);
    }
}

```

```
    leftToRight = !leftToRight; // 切换方向
}
```

```
return result;
```

```
}
```

```
// 二叉树的最大深度
```

```
int maxDepth(TreeNode* root) {
```

```
    if (root == nullptr) {
        return 0;
    }
```

```
queue<TreeNode*> q;
```

```
q.push(root);
```

```
int depth = 0;
```

```
while (!q.empty()) {
```

```
    int levelSize = q.size();
```

```
    // 处理当前层的所有节点
```

```
    for (int i = 0; i < levelSize; i++) {
```

```
        TreeNode* node = q.front();
```

```
        q.pop();
```

```
        if (node->left != nullptr) {
```

```
            q.push(node->left);
```

```
}
```

```
        if (node->right != nullptr) {
```

```
            q.push(node->right);
```

```
}
```

```
}
```

```
    depth++; // 每处理完一层，深度加 1
```

```
}
```

```
return depth;
```

```
}
```

```
// 翻转二叉树
```

```
TreeNode* invertTree(TreeNode* root) {
```

```
    if (root == nullptr) {
```

```
        return nullptr;
```

```
}
```

```
queue<TreeNode*> q;
```

```
q.push(root);

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

    // 交换左右子树
    TreeNode* temp = node->left;
    node->left = node->right;
    node->right = temp;

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

return root;
}
```

```
// 测试函数
int main() {
    // 构建测试二叉树:
    //      1
    //      / \
    //     2   3
    //     / \ / \
    //    4  5 6  7
```

```
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);
```

```
cout << "==== 二叉树遍历测试 ===" << endl;
```

```
// 先序遍历
```

```
vector<int> preorder = preorderTraversal(root);
cout << "先序遍历: ";
for (int val : preorder) {
    cout << val << " ";
}
cout << endl;

// 中序遍历
vector<int> inorder = inorderTraversal(root);
cout << "中序遍历: ";
for (int val : inorder) {
    cout << val << " ";
}
cout << endl;

// 后序遍历（双栈法）
vector<int> postorder1 = postorderTraversalTwoStacks(root);
cout << "后序遍历(双栈法): ";
for (int val : postorder1) {
    cout << val << " ";
}
cout << endl;

// 后序遍历（单栈法）
vector<int> postorder2 = postorderTraversalOneStack(root);
cout << "后序遍历(单栈法): ";
for (int val : postorder2) {
    cout << val << " ";
}
cout << endl;

// 层序遍历
vector<vector<int>> levelorder = levelOrder(root);
cout << "层序遍历: ";
for (const auto& level : levelorder) {
    cout << "[";
    for (size_t i = 0; i < level.size(); ++i) {
        cout << level[i];
        if (i < level.size() - 1) cout << ",";
    }
    cout << "] ";
}
cout << endl;
```

```

// 锯齿形层序遍历
vector<vector<int>> zigzag = zigzagLevelOrder(root);
cout << "锯齿形层序遍历: ";
for (const auto& level : zigzag) {
    cout << "[";
    for (size_t i = 0; i < level.size(); ++i) {
        cout << level[i];
        if (i < level.size() - 1) cout << ",";
    }
    cout << "]";
}
cout << endl;

// 二叉树的最大深度
int depth = maxDepth(root);
cout << "二叉树的最大深度: " << depth << endl;

// 翻转二叉树
TreeNode* inverted = invertTree(root);
vector<vector<int>> invertedLevelOrder = levelOrder(inverted);
cout << "翻转后的层序遍历: ";
for (const auto& level : invertedLevelOrder) {
    cout << "[";
    for (size_t i = 0; i < level.size(); ++i) {
        cout << level[i];
        if (i < level.size() - 1) cout << ",";
    }
    cout << "]";
}
cout << endl;

return 0;
}

// ===== 以下是新增的补充题目 =====

/*
 * 题目 1: LeetCode 107 - 二叉树的层序遍历 II
 * 题目来源: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
 * 题目描述:
 * 给定一个二叉树，返回其节点值自底向上的层序遍历。
 */

```

- * 解题思路：
- * 1. 使用队列进行正常的层序遍历
- * 2. 将每一层的结果添加到列表中
- * 3. 最后将列表反转即可得到自底向上的结果

*

- * 时间复杂度： $O(n)$ – 需要遍历所有 n 个节点一次

- * 空间复杂度： $O(n)$ – 队列最多存储树的最大宽度(最坏情况下为 $n/2$)

- * 是否为最优解：是

*/

```
vector<vector<int>> levelOrderBottom(TreeNode* root) {
```

```
    vector<vector<int>> result;
```

```
    if (root == nullptr) {
```

```
        return result;
```

```
}
```

```
queue<TreeNode*> q;
```

```
q.push(root);
```

```
while (!q.empty()) {
```

```
    int levelSize = q.size();
```

```
    vector<int> levelNodes;
```

```
    for (int i = 0; i < levelSize; i++) {
```

```
        TreeNode* node = q.front();
```

```
        q.pop();
```

```
        levelNodes.push_back(node->val);
```

```
        if (node->left != nullptr) {
```

```
            q.push(node->left);
```

```
}
```

```
        if (node->right != nullptr) {
```

```
            q.push(node->right);
```

```
}
```

```
}
```

```
    result.push_back(levelNodes);
```

```
}
```

```
// 反转结果
```

```
reverse(result.begin(), result.end());
```

```
return result;
```

```
}
```

```

/*
 * 题目 2: LeetCode 637 - 二叉树的层平均值
 * 题目来源: https://leetcode.cn/problems/average-of-levels-in-binary-tree/
 */
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> result;
    if (root == nullptr) {
        return result;
    }

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

    while (!q.empty()) {
        int levelSize = q.size();
        long long sum = 0;

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

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

        result.push_back(static_cast<double>(sum) / levelSize);
    }

    return result;
}

/*
 * 题目 3: LeetCode 515 - 在每个树行中找最大值
 * 题目来源: https://leetcode.cn/problems/find-largest-value-in-each-tree-row/
*/
vector<int> largestValues(TreeNode* root) {
    vector<int> result;
    if (root == nullptr) {

```

```

        return result;
    }

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

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

    for (int i = 0; i < levelSize; i++) {
        TreeNode* node = q.front();
        q.pop();
        maxVal = max(maxVal, node->val);

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

    result.push_back(maxVal);
}

return result;
}

```

=====

文件: BinaryTreeTraversalIteration.java

=====

```

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

```

// 自定义 Pair 类, 用于存储键值对

```

class Pair<K, V> {
    private K key;
    private V value;
}

```

```

public Pair(K key, V value) {
    this.key = key;
    this.value = value;
}

public K getKey() {
    return key;
}

public V getValue() {
    return value;
}
}

```

// 不用递归，用迭代的方式实现二叉树的三序遍历

```
public class BinaryTreeTraversalIteration {
```

```

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

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

// ===== 以下是新增的补充题目 =====

```
/**
```

- * 题目: LeetCode 112 - 路径总和
- * 题目来源: <https://leetcode.cn/problems/path-sum/>
- * 题目描述: 给你二叉树的根节点 root 和一个表示目标和的整数 targetSum 。
- * 判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum 。
- * 如果存在，返回 true ；否则，返回 false 。
- *
- * 解题思路:
- * 1. 使用深度优先搜索（DFS）遍历二叉树
- * 2. 从根节点开始，每次遍历到一个节点时，将当前累计和减去节点值
- * 3. 如果到达叶子节点且累计和为 0，则返回 true
- * 4. 否则继续递归遍历左右子树
- *

```

* 时间复杂度: O(n) - 需要遍历树中的所有节点
* 空间复杂度: O(h) - 递归调用栈的深度, h 为树的高度
* 是否为最优解: 是, DFS 是解决此类路径问题的最优方法
*/
public static boolean hasPathSum(TreeNode root, int targetSum) {
    // 边界情况: 空树
    if (root == null) {
        return false;
    }

    // 非递归 DFS 实现 - 使用栈同时存储节点和当前路径和
    Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
    stack.push(new Pair<>(root, targetSum - root.val));

    while (!stack.isEmpty()) {
        Pair<TreeNode, Integer> current = stack.pop();
        TreeNode node = current.getKey();
        int remainingSum = current.getValue();

        // 如果是叶子节点且剩余和为 0, 找到符合条件的路径
        if (node.left == null && node.right == null && remainingSum == 0) {
            return true;
        }

        // 先压入右子节点, 这样保证左子节点先被处理 (DFS 顺序)
        if (node.right != null) {
            stack.push(new Pair<>(node.right, remainingSum - node.right.val));
        }
        if (node.left != null) {
            stack.push(new Pair<>(node.left, remainingSum - node.left.val));
        }
    }

    return false;
}

/**
 * 题目: LeetCode 113 - 路径总和 II
 * 题目来源: https://leetcode.cn/problems/path-sum-ii/
 * 题目描述: 给你二叉树的根节点 root 和一个整数目标和 targetSum ,
 * 找出所有 从根节点到叶子节点 路径总和等于给定目标和的路径。
 *
 * 解题思路:

```

```

* 1. 使用回溯算法（深度优先搜索）
* 2. 维护一个当前路径列表，记录已经走过的节点值
* 3. 当到达叶子节点且路径和等于目标和时，将当前路径加入结果集
* 4. 否则继续递归搜索左右子树
*
* 时间复杂度:  $O(n^2)$  - 每个节点访问一次，最坏情况下需要将路径复制  $n$  次
* 空间复杂度:  $O(h)$  - 递归调用栈和路径列表的空间， $h$  为树的高度
* 是否为最优解: 是，回溯是寻找所有路径的标准方法
*/
public static List<List<Integer>> pathSum(TreeNode root, int targetSum) {
    List<List<Integer>> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 非递归 DFS 实现
    Stack<Pair<TreeNode, Integer>> nodeStack = new Stack<>();
    Stack<List<Integer>> pathStack = new Stack<>();

    // 初始化: 将根节点和路径加入栈中
    nodeStack.push(new Pair<>(root, root.val));
    List<Integer> initialPath = new ArrayList<>();
    initialPath.add(root.val);
    pathStack.push(initialPath);

    while (!nodeStack.isEmpty()) {
        Pair<TreeNode, Integer> current = nodeStack.pop();
        TreeNode node = current.getKey();
        int currentSum = current.getValue();
        List<Integer> currentPath = pathStack.pop();

        // 如果是叶子节点且和等于目标值，加入结果集
        if (node.left == null && node.right == null && currentSum == targetSum) {
            result.add(new ArrayList<>(currentPath));
        }

        // 先处理右子树（栈是后进先出，所以右子树先入栈）
        if (node.right != null) {
            int rightSum = currentSum + node.right.val;
            nodeStack.push(new Pair<>(node.right, rightSum));

            List<Integer> rightPath = new ArrayList<>(currentPath);
            rightPath.add(node.right.val);
        }
    }
}

```

```

        pathStack.push(rightPath);
    }

    // 再处理左子树
    if (node.left != null) {
        int leftSum = currentSum + node.left.val;
        nodeStack.push(new Pair<>(node.left, leftSum));

        List<Integer> leftPath = new ArrayList<>(currentPath);
        leftPath.add(node.left.val);
        pathStack.push(leftPath);
    }
}

return result;
}

/***
 * 题目: LeetCode 129 - 求根节点到叶节点数字之和
 * 题目来源: https://leetcode.cn/problems/sum-root-to-leaf-numbers/
 * 题目描述: 给你一个二叉树的根节点 root ，树中每个节点都存放有一个 0 到 9 之间的数字。
 * 每条从根节点到叶节点的路径都代表一个数字。
 * 例如，从根节点到叶节点的路径 1 -> 2 -> 3 表示数字 123 。
 * 计算从根节点到叶节点生成的 所有数字之和 。
 *
 * 解题思路:
 * 1. 使用深度优先搜索遍历二叉树
 * 2. 维护一个当前路径代表的数字
 * 3. 当到达叶子节点时，将当前数字加入总和
 *
 * 时间复杂度: O(n) - 每个节点只访问一次
 * 空间复杂度: O(h) - 递归栈的深度
 * 是否为最优解: 是，DFS 是解决此类路径问题的高效方法
 */

public static int sumNumbers(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 非递归 DFS 实现
    Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
    stack.push(new Pair<>(root, root.val));
    int totalSum = 0;
}

```

```

while (!stack.isEmpty()) {
    Pair<TreeNode, Integer> current = stack.pop();
    TreeNode node = current.getKey();
    int currentNumber = current.getValue();

    // 如果是叶子节点，将当前数字加入总和
    if (node.left == null && node.right == null) {
        totalSum += currentNumber;
    } else {
        // 非叶子节点，继续向下遍历
        if (node.right != null) {
            stack.push(new Pair<>(node.right, currentNumber * 10 + node.right.val));
        }
        if (node.left != null) {
            stack.push(new Pair<>(node.left, currentNumber * 10 + node.left.val));
        }
    }
}

return totalSum;
}

/**
 * 题目：LeetCode 257 - 二叉树的所有路径
 * 题目来源：https://leetcode.cn/problems/binary-tree-paths/
 * 题目描述：给你一个二叉树的根节点 root ，按 任意顺序 ，
 * 返回所有从根节点到叶子节点的路径。
 *
 * 解题思路：
 * 1. 使用回溯算法（DFS）
 * 2. 维护当前路径字符串
 * 3. 当到达叶子节点时，将完整路径加入结果集
 * 4. 继续递归处理左右子树
 *
 * 时间复杂度：O(n) - 每个节点访问一次，路径字符串拼接可能需要 O(n) 时间
 * 空间复杂度：O(h) - 递归栈的深度
 * 是否为最优解：是，DFS 是生成所有路径的标准方法
 */
public static List<String> binaryTreePaths(TreeNode root) {
    List<String> result = new ArrayList<>();
    if (root == null) {
        return result;
    }

    // 从根节点开始遍历
    helper(root, "", result);
    return result;
}

private void helper(TreeNode node, String path, List<String> result) {
    if (node == null) {
        return;
    }

    // 将当前节点值添加到路径中
    path += node.val;

    // 如果是叶子节点，将路径加入结果集
    if (node.left == null && node.right == null) {
        result.add(path);
    } else {
        // 递归处理左子树
        helper(node.left, path, result);
        // 递归处理右子树
        helper(node.right, path, result);
    }
}

```

```

}

// 非递归 DFS 实现
Stack<Pair<TreeNode, String>> stack = new Stack<>();
stack.push(new Pair<>(root, String.valueOf(root.val)));

while (!stack.isEmpty()) {
    Pair<TreeNode, String> current = stack.pop();
    TreeNode node = current.getKey();
    String path = current.getValue();

    // 如果是叶子节点，将路径加入结果集
    if (node.left == null && node.right == null) {
        result.add(path);
    } else {
        // 非叶子节点，继续向下遍历
        if (node.right != null) {
            stack.push(new Pair<>(node.right, path + "->" + node.right.val));
        }
        if (node.left != null) {
            stack.push(new Pair<>(node.left, path + "->" + node.left.val));
        }
    }
}

return result;
}

/**
 * 题目: LeetCode 1448 - 统计二叉树中好节点的数目
 * 题目来源: https://leetcode.cn/problems/count-good-nodes-in-binary-tree/
 * 题目描述: 给你一棵根为 root 的二叉树，请你返回二叉树中好节点的数目。
 * 「好节点」X 定义为：从根到该节点 X 所经过的节点中，没有任何节点的值大于 X 的值。
 *
 * 解题思路:
 * 1. 使用深度优先搜索遍历二叉树
 * 2. 维护从根到当前节点路径上的最大值
 * 3. 如果当前节点的值大于等于该最大值，则为好节点，更新最大值
 * 4. 继续递归处理左右子树
 *
 * 时间复杂度: O(n) - 每个节点只访问一次
 * 空间复杂度: O(h) - 递归栈的深度
 * 是否为最优解: 是，DFS 是解决此类路径最大值问题的最优方法

```

```

*/
public static int goodNodes(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 非递归 DFS 实现
    Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
    stack.push(new Pair<>(root, root.val)); // (节点, 路径最大值)
    int goodCount = 0;

    while (!stack.isEmpty()) {
        Pair<TreeNode, Integer> current = stack.pop();
        TreeNode node = current.getKey();
        int maxSoFar = current.getValue();

        // 判断是否为好节点
        if (node.val >= maxSoFar) {
            goodCount++;
            maxSoFar = node.val; // 更新路径最大值
        }

        // 继续处理左右子树
        if (node.right != null) {
            stack.push(new Pair<>(node.right, maxSoFar));
        }
        if (node.left != null) {
            stack.push(new Pair<>(node.left, maxSoFar));
        }
    }

    return goodCount;
}

/**
 * 题目：剑指 Offer 26 - 树的子结构
 * 题目来源: https://leetcode.cn/problems/shu-de-zi-jie-gou-lcof/
 * 题目描述: 输入两棵二叉树 A 和 B，判断 B 是不是 A 的子结构。
 * 约定空树不是任意一个树的子结构。
 *
 * 解题思路:
 * 1. 先序遍历树 A 中的每个节点 nA
 * 2. 对于每个节点 nA，检查以 nA 为根节点的子树是否包含树 B

```

```

* 3. 检查是否包含的逻辑：递归比较节点值是否相等，左子树和右子树是否也满足条件
*
* 时间复杂度: O(m*n) - m 和 n 分别是两棵树的节点数
* 空间复杂度: O(h) - 递归栈的深度，h 为树 A 的高度
* 是否为最优解: 是，需要遍历树 A 的每个节点并进行匹配
*/
public static boolean isSubStructure(TreeNode A, TreeNode B) {
    // 空树不是任意一个树的子结构
    if (A == null || B == null) {
        return false;
    }

    // 非递归 DFS 实现，遍历树 A 的每个节点
    Stack<TreeNode> stack = new Stack<>();
    stack.push(A);

    while (!stack.isEmpty()) {
        TreeNode node = stack.pop();

        // 检查以当前节点为根的子树是否包含树 B
        if (isMatch(node, B)) {
            return true;
        }

        // 继续遍历其他节点
        if (node.right != null) {
            stack.push(node.right);
        }
        if (node.left != null) {
            stack.push(node.left);
        }
    }

    return false;
}

// 辅助方法: 检查以 A 为根的子树是否包含以 B 为根的子树
private static boolean isMatch(TreeNode A, TreeNode B) {
    // 递归实现更清晰
    if (B == null) {
        return true; // B 已经匹配完
    }
    if (A == null || A.val != B.val) {

```

```

        return false; // A 为空或值不匹配
    }

    // 继续匹配左右子树
    return isMatch(A.left, B.left) && isMatch(A.right, B.right);
}

/***
 * 题目: LeetCode 1372 - 二叉树中的最长交错路径
 * 题目来源: https://leetcode.cn/problems/longest-zigzag-path-in-a-binary-tree/
 * 题目描述: 给你一棵以 root 为根的二叉树, 返回其最长的交错路径的长度。
 * 交错路径的定义如下: 从一个节点开始, 沿着父-子连接, 向上或向下移动,
 * 移动时, 节点的方向必须交替变化 (即从左到右, 或从右到左)。
 *
 * 解题思路:
 * 1. 使用深度优先搜索遍历二叉树
 * 2. 对每个节点, 记录从上一个节点来的方向 (左或右)
 * 3. 如果当前方向与上一个方向交替, 则路径长度+1, 否则重置为 1
 * 4. 更新全局最大路径长度
 *
 * 时间复杂度: O(n) - 每个节点只访问一次
 * 空间复杂度: O(h) - 递归栈的深度
 * 是否为最优解: 是, 一次遍历即可找到最长交错路径
 */
public static int longestZigZag(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 使用数组作为引用传递最大值
    int[] maxLength = {0};

    // 非递归 DFS 实现, 栈中存储三元组: (节点, 方向, 当前长度)
    // 方向: -1 表示从父节点的左子树来, 1 表示从父节点的右子树来, 0 表示根节点
    Stack<Object[]> stack = new Stack<>();
    stack.push(new Object[] {root, 0, 0});

    while (!stack.isEmpty()) {
        Object[] current = stack.pop();
        TreeNode node = (TreeNode) current[0];
        int direction = (int) current[1];
        int length = (int) current[2];

        // 更新最大值
        maxLength[0] = Math.max(maxLength[0], length);

        if (direction == -1) {
            if (node.left != null) {
                stack.push(new Object[] {node.left, 1, length + 1});
            }
        } else if (direction == 1) {
            if (node.right != null) {
                stack.push(new Object[] {node.right, -1, length + 1});
            }
        }
    }
}

```

```

maxLength[0] = Math.max(maxLength[0], length);

// 处理左子树
if (node.left != null) {
    int newLength = (direction == 1) ? length + 1 : 1;
    stack.push(new Object[] {node.left, -1, newLength});
}

// 处理右子树
if (node.right != null) {
    int newLength = (direction == -1) ? length + 1 : 1;
    stack.push(new Object[] {node.right, 1, newLength});
}

return maxLength[0];
}

/***
 * 题目: LeetCode 222 - 完全二叉树的节点个数
 * 题目来源: https://leetcode.cn/problems/count-complete-tree-nodes/
 * 题目描述: 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 的定义是: 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。
 *
 * 解题思路:
 * 1. 利用完全二叉树的特性: 如果左子树的高度等于右子树的高度，则左子树是满二叉树
 * 2. 如果左子树的高度大于右子树的高度，则右子树是满二叉树
 * 3. 满二叉树的节点数为  $2^h - 1$ ，其中 h 是树的高度
 * 4. 递归计算剩余部分的节点数
 *
 * 时间复杂度:  $O(\log^2 n)$  - 每次计算高度需要  $O(\log n)$ ，递归深度为  $O(\log n)$ 
 * 空间复杂度:  $O(\log n)$  - 递归栈的深度
 * 是否为最优解: 是，利用完全二叉树特性进行优化
 */

public static int countNodes(TreeNode root) {
    if (root == null) {
        return 0;
    }

    // 计算树的高度（从根到最左边叶子节点的距离）
    int leftHeight = getHeight(root.left);
    int rightHeight = getHeight(root.right);

```

```
if (leftHeight == rightHeight) {
    // 左子树是满二叉树，节点数为  $2^{\text{leftHeight}} - 1$ ，加上根节点和右子树
    return (1 << leftHeight) + countNodes(root.right);
} else {
    // 右子树是满二叉树，节点数为  $2^{\text{rightHeight}} - 1$ ，加上根节点和左子树
    return (1 << rightHeight) + countNodes(root.left);
}
}
```

// 辅助方法：计算完全二叉树的高度（从根到最左边叶子节点的距离）

```
private static int getHeight(TreeNode node) {
    int height = 0;
    while (node != null) {
        height++;
        node = node.left;
    }
    return height;
}
}
```

// 先序打印所有节点，非递归版

```
public static void preOrder(TreeNode head) {
    if (head != null) {
        Stack<TreeNode> stack = new Stack<>();
        stack.push(head);
        while (!stack.isEmpty()) {
            head = stack.pop();
            System.out.print(head.val + " ");
            if (head.right != null) {
                stack.push(head.right);
            }
            if (head.left != null) {
                stack.push(head.left);
            }
        }
        System.out.println();
    }
}
```

// 中序打印所有节点，非递归版

```
public static void inOrder(TreeNode head) {
    if (head != null) {
```

```

Stack<TreeNode> stack = new Stack<>();
while (!stack.isEmpty() || head != null) {
    if (head != null) {
        stack.push(head);
        head = head.left;
    } else {
        head = stack.pop();
        System.out.print(head.val + " ");
        head = head.right;
    }
}
System.out.println();
}
}

```

```

// 后序打印所有节点，非递归版
// 这是用两个栈的方法
public static void posOrderTwoStacks(TreeNode head) {
    if (head != null) {
        Stack<TreeNode> stack = new Stack<>();
        Stack<TreeNode> collect = new Stack<>();
        stack.push(head);
        while (!stack.isEmpty()) {
            head = stack.pop();
            collect.push(head);
            if (head.left != null) {
                stack.push(head.left);
            }
            if (head.right != null) {
                stack.push(head.right);
            }
        }
        while (!collect.isEmpty()) {
            System.out.print(collect.pop().val + " ");
        }
        System.out.println();
    }
}

```

```

// 后序打印所有节点，非递归版
// 这是用一个栈的方法
public static void posOrderOneStack(TreeNode h) {
    if (h != null) {

```

```

Stack<TreeNode> stack = new Stack<>();
stack.push(h);
// 如果始终没有打印过节点, h 就一直是头节点
// 一旦打印过节点, h 就变成打印节点
// 之后 h 的含义 : 上一次打印的节点
while (!stack.isEmpty()) {
    TreeNode cur = stack.peek();
    if (cur.left != null && h != cur.left && h != cur.right) {
        // 有左树且左树没处理过
        stack.push(cur.left);
    } else if (cur.right != null && h != cur.right) {
        // 有右树且右树没处理过
        stack.push(cur.right);
    } else {
        // 左树、右树 没有 或者 都处理过了
        System.out.print(cur.val + " ");
        h = stack.pop();
    }
}
System.out.println();
}

}

// 层序遍历（二叉树的广度优先遍历）
// 测试链接: https://leetcode.cn/problems/binary-tree-level-order-traversal/
public static void levelOrder(TreeNode head) {
    if (head == null) {
        return;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(head);
    while (!queue.isEmpty()) {
        TreeNode cur = queue.poll();
        System.out.print(cur.val + " ");
        if (cur.left != null) {
            queue.offer(cur.left);
        }
        if (cur.right != null) {
            queue.offer(cur.right);
        }
    }
    System.out.println();
}

```

```

public static void main(String[] args) {
    // 简化测试，减少内存使用
    TreeNode head = new TreeNode(1);
    head.left = new TreeNode(2);
    head.right = new TreeNode(3);
    head.left.left = new TreeNode(4);
    head.left.right = new TreeNode(5);

    System.out.println("== 简化版二叉树遍历测试 ==");

    // 只测试核心功能
    List<Integer> preorder = preorderTraversal(head);
    System.out.println("前序遍历: " + preorder);

    List<Integer> inorder = inorderTraversal(head);
    System.out.println("中序遍历: " + inorder);

    List<Integer> postorder = postorderTraversalOneStack(head);
    System.out.println("后序遍历: " + postorder);

    List<List<Integer>> levelorder = levelOrderTraversal(head);
    System.out.println("层序遍历: " + levelorder);

    System.out.println("== 测试完成 ==");
}

// 用一个栈完成先序遍历
// 测试链接 : https://leetcode.cn/problems/binary-tree-preorder-traversal/
public static List<Integer> preorderTraversal(TreeNode head) {
    List<Integer> ans = new ArrayList<>();
    if (head != null) {
        Stack<TreeNode> stack = new Stack<>();
        stack.push(head);
        while (!stack.isEmpty()) {
            head = stack.pop();
            ans.add(head.val);
            if (head.right != null) {
                stack.push(head.right);
            }
            if (head.left != null) {
                stack.push(head.left);
            }
        }
    }
    return ans;
}

```

```
        }
    }

    return ans;
}

// 用一个栈完成中序遍历
// 测试链接 : https://leetcode.cn/problems/binary-tree-inorder-traversal/
public static List<Integer> inorderTraversal(TreeNode head) {
    List<Integer> ans = new ArrayList<>();
    if (head != null) {
        Stack<TreeNode> stack = new Stack<>();
        while (!stack.isEmpty() || head != null) {
            if (head != null) {
                stack.push(head);
                head = head.left;
            } else {
                head = stack.pop();
                ans.add(head.val);
                head = head.right;
            }
        }
    }
    return ans;
}
```

```
// 用两个栈完成后序遍历
// 提交时函数名改为 postorderTraversal
// 测试链接 : https://leetcode.cn/problems/binary-tree-postorder-traversal/
public static List<Integer> postorderTraversalTwoStacks(TreeNode head) {
    List<Integer> ans = new ArrayList<>();
    if (head != null) {
        Stack<TreeNode> stack = new Stack<>();
        Stack<TreeNode> collect = new Stack<>();
        stack.push(head);
        while (!stack.isEmpty()) {
            head = stack.pop();
            collect.push(head);
            if (head.left != null) {
                stack.push(head.left);
            }
            if (head.right != null) {
                stack.push(head.right);
            }
        }
    }
    return ans;
}
```

```

    }

    while (!collect.isEmpty()) {
        ans.add(collect.pop().val);
    }
}

return ans;
}

// 用一个栈完成后序遍历
// 提交时函数名改为 postorderTraversal
// 测试链接 : https://leetcode.cn/problems/binary-tree-postorder-traversal/
public static List<Integer> postorderTraversalOneStack(TreeNode h) {

    List<Integer> ans = new ArrayList<>();
    if (h != null) {
        Stack<TreeNode> stack = new Stack<>();
        stack.push(h);
        while (!stack.isEmpty()) {
            TreeNode cur = stack.peek();
            if (cur.left != null && h != cur.left && h != cur.right) {
                stack.push(cur.left);
            } else if (cur.right != null && h != cur.right) {
                stack.push(cur.right);
            } else {
                ans.add(cur.val);
                h = stack.pop();
            }
        }
    }
    return ans;
}

// 层序遍历（广度优先遍历）
// 测试链接: https://leetcode.cn/problems/binary-tree-level-order-traversal/
public static List<List<Integer>> levelOrderTraversal(TreeNode root) {

    List<List<Integer>> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();

```

```

        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            level.add(cur.val);
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
        ans.add(level);
    }
    return ans;
}

// 锯齿形层序遍历
// 测试链接: https://leetcode.cn/problems/binary-tree-zigzag-level-order-traversal/
public static List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    boolean isLeftToRight = true;
    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            if (isLeftToRight) {
                level.add(cur.val);
            } else {
                level.add(0, cur.val); // 在列表开头插入元素，实现反向
            }
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
        ans.add(level);
    }
}

```

```

    isLeftToRight = !isLeftToRight; // 切换方向
}
return ans;
}

// 二叉树的右视图
// 测试链接: https://leetcode.cn/problems/binary-tree-right-side-view/
public static List<Integer> rightSideView(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            if (i == size - 1) { // 每一层的最后一个节点就是从右边看到的节点
                ans.add(cur.val);
            }
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
    }
    return ans;
}

// 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) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int depth = 0;
    while (!queue.isEmpty()) {
        int size = queue.size();

```

```

// 处理当前层的所有节点
for (int i = 0; i < size; i++) {
    TreeNode cur = queue.poll();
    if (cur.left != null) {
        queue.offer(cur.left);
    }
    if (cur.right != null) {
        queue.offer(cur.right);
    }
}
depth++; // 每处理完一层，深度加 1
}

return depth;
}

// 二叉树的最小深度
// 测试链接: https://leetcode.cn/problems/minimum-depth-of-binary-tree/
public static int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int depth = 1;
    while (!queue.isEmpty()) {
        int size = queue.size();
        // 处理当前层的所有节点
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            // 如果当前节点是叶子节点，直接返回深度
            if (cur.left == null && cur.right == null) {
                return depth;
            }
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
        depth++; // 处理完一层，深度加 1
    }
    return depth;
}

```

```
}
```



```
// 翻转二叉树
```

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

```
public static TreeNode invertTree(TreeNode root) {
```

```
    if (root == null) {
```

```
        return null;
```

```
    }
```

```
    Queue<TreeNode> queue = new LinkedList<>();
```

```
    queue.offer(root);
```

```
    while (!queue.isEmpty()) {
```

```
        TreeNode cur = queue.poll();
```

```
        // 交换左右子树
```

```
        TreeNode temp = cur.left;
```

```
        cur.left = cur.right;
```

```
        cur.right = temp;
```

```
        // 将非空子节点加入队列
```

```
        if (cur.left != null) {
```

```
            queue.offer(cur.left);
```

```
        }
```

```
        if (cur.right != null) {
```

```
            queue.offer(cur.right);
```

```
        }
```

```
    }
```

```
    return root;
```

```
}
```



```
// 对称二叉树
```

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

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

```
    if (root == null) {
```

```
        return true;
```

```
    }
```

```
    Queue<TreeNode> queue = new LinkedList<>();
```

```
    queue.offer(root.left);
```

```
    queue.offer(root.right);
```

```
    while (!queue.isEmpty()) {
```

```
        TreeNode left = queue.poll();
```

```
        TreeNode right = queue.poll();
```

```
        // 如果两个节点都为空, 继续比较
```

```
        if (left == null && right == null) {
```

```
            continue;
```

```
        }
```

```

// 如果其中一个为空或者值不相等，返回 false
if (left == null || right == null || left.val != right.val) {
    return false;
}

// 按照对称的顺序加入队列
queue.offer(left.left);
queue.offer(right.right);
queue.offer(left.right);
queue.offer(right.left);

}

return true;
}

// 二叉树的序列化与反序列化
// 测试链接: https://leetcode.cn/problems/serialize-and-deserialize-binary-tree/
public static String serialize(TreeNode root) {
    if (root == null) {
        return "[]";
    }

StringBuilder sb = new StringBuilder("[");
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
while (!queue.isEmpty()) {
    TreeNode cur = queue.poll();
    if (cur != null) {
        sb.append(cur.val).append(",");
        queue.offer(cur.left);
        queue.offer(cur.right);
    } else {
        sb.append("null,");
    }
}

// 删除最后多余的逗号
sb.deleteCharAt(sb.length() - 1);
sb.append("]");
return sb.toString();
}

// 二叉树的反序列化
public static TreeNode deserialize(String data) {
    if (data.equals("[]")) {
        return null;
    }
}

```

```

// 去掉方括号并按逗号分割
String[] vals = data.substring(1, data.length() - 1).split(",");
TreeNode root = new TreeNode(Integer.parseInt(vals[0]));
Queue<TreeNode> queue = new LinkedList<>();
queue.offer(root);
int i = 1;
while (!queue.isEmpty()) {
    TreeNode cur = queue.poll();
    // 处理左子节点
    if (!vals[i].equals("null")) {
        cur.left = new TreeNode(Integer.parseInt(vals[i]));
        queue.offer(cur.left);
    }
    i++;
    // 处理右子节点
    if (!vals[i].equals("null")) {
        cur.right = new TreeNode(Integer.parseInt(vals[i]));
        queue.offer(cur.right);
    }
    i++;
}
return root;
}

```

```

// 剑指 Offer 07. 重建二叉树
// 根据前序遍历和中序遍历构建二叉树
// 测试链接: https://leetcode.cn/problems.zhong-jian-er-cha-shu-lcof/
public static TreeNode buildTree(int[] preorder, int[] inorder) {
    if (preorder == null || preorder.length == 0) {
        return null;
    }
    // 使用栈来模拟递归过程
    TreeNode root = new TreeNode(preorder[0]);
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    int inorderIndex = 0;
    for (int i = 1; i < preorder.length; i++) {
        int preorderVal = preorder[i];
        TreeNode node = stack.peek();
        if (node.val != inorder[inorderIndex]) {
            // 当前节点是栈顶节点的左子节点
            node.left = new TreeNode(preorderVal);
            stack.push(node.left);
        } else {
            stack.pop();
            inorderIndex++;
        }
    }
}

```

```

    } else {
        // 当前节点是栈中某个节点的右子节点
        while (!stack.isEmpty() && stack.peek().val == inorder[inorderIndex]) {
            node = stack.pop();
            inorderIndex++;
        }
        node.right = new TreeNode(preorderVal);
        stack.push(node.right);
    }
}

return root;
}

// 剑指 Offer 32 - III. 从上到下打印二叉树 III
// 之字形层序遍历二叉树
// 测试链接: https://leetcode.cn/problems/cong-shang-dao-xia-da-yin-er-cha-shu-iii-lcof/
public static List<List<Integer>> levelOrderZigzag(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    boolean leftToRight = true;
    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (leftToRight) {
                level.add(node.val);
            } else {
                level.add(0, node.val);
            }
            if (node.left != null) {
                queue.offer(node.left);
            }
            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        ans.add(level);
        leftToRight = !leftToRight;
    }
}

```

```

    }

    return ans;
}

// 剑指 Offer 54. 二叉搜索树的第 k 大节点
// 测试链接: https://leetcode.cn/problems/er-cha-sou-suo-shu-de-di-kda-jie-dian-lcof/
public static int kthLargest(TreeNode root, int k) {
    // 使用栈实现反向中序遍历 (右->根->左)
    Stack<TreeNode> stack = new Stack<>();
    TreeNode cur = root;
    int count = 0;
    while (cur != null || !stack.isEmpty()) {
        // 先到最右边的节点
        while (cur != null) {
            stack.push(cur);
            cur = cur.right;
        }
        // 处理当前节点
        cur = stack.pop();
        count++;
        if (count == k) {
            return cur.val;
        }
        // 转向左子树
        cur = cur.left;
    }
    return 0;
}

// 剑指 Offer 55 - I. 二叉树的深度
// 测试链接: https://leetcode.cn/problems/er-cha-shu-de-shen-du-lcof/
public static int maxDepth2(TreeNode root) {
    if (root == null) {
        return 0;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int depth = 0;
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode node = queue.poll();
            if (node.left != null) {

```

```

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

return depth;
}

// 剑指 Offer 55 - II. 平衡二叉树
// 测试链接: https://leetcode.cn/problems/ping-heng-er-cha-shu-lcof/
public static boolean isBalanced(TreeNode root) {
    return height(root) >= 0;
}

private static int height(TreeNode root) {
    if (root == null) {
        return 0;
    }
    int leftHeight = height(root.left);
    int rightHeight = height(root.right);
    // 如果左右子树高度差超过 1，或者左右子树本身不平衡，返回-1
    if (leftHeight == -1 || rightHeight == -1 || Math.abs(leftHeight - rightHeight) > 1) {
        return -1;
    } else {
        return Math.max(leftHeight, rightHeight) + 1;
    }
}

// 剑指 Offer 68 - II. 二叉树的最近公共祖先
// 测试链接: https://leetcode.cn/problems/er-cha-shu-de-zui-jin-gong-zu-xian-lcof/
public static TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // 使用栈存储从根节点到目标节点的路径
    Stack<TreeNode> pathP = new Stack<>();
    Stack<TreeNode> pathQ = new Stack<>();
    // 获取从根节点到 p 和 q 的路径
    getPath(root, p, pathP);
    getPath(root, q, pathQ);
    // 找到两条路径的最后一个公共节点
    TreeNode ancestor = null;
    while (!pathP.isEmpty() && !pathQ.isEmpty() && pathP.peek() == pathQ.peek()) {

```

```

ancestor = pathP.peek();
pathP.pop();
pathQ.pop();
}
return ancestor;
}

// 获取从根节点到目标节点的路径
private static boolean getPath(TreeNode root, TreeNode target, Stack<TreeNode> path) {
    if (root == null) {
        return false;
    }
    path.push(root);
    if (root == target) {
        return true;
    }
    // 在左右子树中查找
    if (getPath(root.left, target, path) || getPath(root.right, target, path)) {
        return true;
    }
    // 如果左右子树都没有找到，回溯
    path.pop();
    return false;
}

```

// ===== 以下是新增的补充题目 =====

```

/*
 * 题目 1: LeetCode 107 - 二叉树的层序遍历 II
 * 题目来源: https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/
 * 题目描述:
 * 给定一个二叉树，返回其节点值自底向上的层序遍历。 (即按从叶子节点所在层到根节点所在的层，逐层从左向右遍历)
 *
 * 解题思路:
 * 1. 使用队列进行正常的层序遍历
 * 2. 将每一层的结果添加到列表中
 * 3. 最后将列表反转即可得到自底向上的结果
 *
 * 时间复杂度: O(n)
 * - 需要遍历所有 n 个节点一次
 *
 * 空间复杂度: O(n)

```

```

* - 队列最多存储树的最大宽度（最坏情况下为 n/2）
* - 结果列表存储所有 n 个节点
*
* 是否为最优解：是
* - 必须访问所有节点，时间复杂度 O(n) 无法优化
* - 必须存储所有节点值，空间复杂度 O(n) 无法优化
*/
public static List<List<Integer>> levelOrderBottom(TreeNode root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        List<Integer> level = new ArrayList<>();
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            level.add(cur.val);
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
        ans.add(0, level); // 在列表开头插入，实现自底向上
    }
    return ans;
}

/*
* 题目 2: LeetCode 199 - 二叉树的右视图
* 题目来源: https://leetcode.cn/problems/binary-tree-right-side-view/
* 题目描述:
* 给定一个二叉树的根节点 root，想象自己站在它的右侧，按照从顶部到底部的顺序，返回从右侧所能看到的节点值。
*
* 解题思路:
* 1. 使用层序遍历
* 2. 每一层只取最后一个节点（最右边的节点）
*

```

```
* 时间复杂度: O(n)
* - 需要遍历所有 n 个节点
*
* 空间复杂度: O(w)
* - w 为树的最大宽度, 队列最多存储一层的节点
* - 最坏情况下为 O(n) (满二叉树的最后一层)
*
* 是否为最优解: 是
* - 层序遍历是最直观且高效的方法
* - 也可以用 DFS 实现, 但需要记录深度, 代码更复杂
*/
// 该方法已在前面实现
```

```
/*
* 题目 3: LeetCode 637 - 二叉树的层平均值
* 题目来源: 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。
*
* 解题思路:
* 1. 使用层序遍历
* 2. 对每一层的节点值求和, 然后除以节点数
*
* 时间复杂度: O(n)
* - 需要遍历所有 n 个节点
*
* 空间复杂度: O(w)
* - w 为树的最大宽度
*
* 是否为最优解: 是
* - 必须访问所有节点才能计算平均值
*/

```

```
public static List<Double> averageOfLevels(TreeNode root) {
    List<Double> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        long sum = 0; // 使用 long 避免整数溢出
        for (int i = 0; i < size; i++) {
```

```

        TreeNode cur = queue.poll();
        sum += cur.val;
        if (cur.left != null) {
            queue.offer(cur.left);
        }
        if (cur.right != null) {
            queue.offer(cur.right);
        }
    }
    ans.add((double) sum / size);
}
return ans;
}

/*
* 题目 4: LeetCode 429 - N 叉树的层序遍历
* 题目来源: https://leetcode.cn/problems/n-ary-tree-level-order-traversal/
* 题目描述:
* 给定一个 N 叉树，返回其节点值的层序遍历。
*
* 解题思路:
* 1. 使用队列进行层序遍历
* 2. 与二叉树类似，但需要遍历所有子节点
*
* 时间复杂度: O(n)
* - 需要遍历所有 n 个节点
*
* 空间复杂度: O(w)
* - w 为树的最大宽度
*
* 是否为最优解: 是
*/
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;
}
}

public static List<List<Integer>> nAryLevelOrder(Node root) {
    List<List<Integer>> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    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 cur = queue.poll();
            level.add(cur.val);
            // 将所有子节点加入队列
            if (cur.children != null) {
                for (Node child : cur.children) {
                    if (child != null) {
                        queue.offer(child);
                    }
                }
            }
        }
        ans.add(level);
    }
    return ans;
}

/*
 * 题目 5: LeetCode 515 - 在每个树行中找最大值
 * 题目来源: 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 ，请找出存在于每一层的最大值。
 *
 * 解题思路:
 * 1. 使用层序遍历
 * 2. 对每一层记录最大值
 *
 * 时间复杂度: O(n)

```

```

* 空间复杂度: O(w), w 为最大宽度
* 是否为最优解: 是
*/
public static List<Integer> largestValues(TreeNode root) {
    List<Integer> ans = new ArrayList<>();
    if (root == null) {
        return ans;
    }
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        int maxVal = Integer.MIN_VALUE; // 初始化为最小值
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            maxVal = Math.max(maxVal, cur.val);
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
        ans.add(maxVal);
    }
    return ans;
}

```

```

/*
* 题目 6: LeetCode 513 - 找树左下角的值
* 题目来源: 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 最底层 最左边 节点的值。
*
* 解题思路:
* 1. 使用层序遍历
* 2. 记录每一层的第一个节点
* 3. 最后一层的第一个节点就是答案
*
* 时间复杂度: O(n)
* 空间复杂度: O(w)
* 是否为最优解: 是
*/

```

```

public static int findBottomLeftValue(TreeNode root) {
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int leftmost = root.val;
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode cur = queue.poll();
            if (i == 0) { // 记录每一层的第一个节点
                leftmost = cur.val;
            }
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
    }
    return leftmost;
}

/*
 * 题目 7: LeetCode 116 - 填充每个节点的下一个右侧节点指针
 * 题目来源: 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。
 *
 * 解题思路:
 * 1. 使用层序遍历
 * 2. 对于每一层的节点，将前一个节点的 next 指向当前节点
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(w)
 * 是否为最优解: 是 (对于迭代方法)
 * 替代方案: 可以使用 O(1) 空间的方法，利用已经建立的 next 指针
 */
static class NextNode {
    public int val;
    public NextNode left;
    public NextNode right;
    public NextNode next;
}

```

```

public NextNode() {}

public NextNode(int _val) {
    val = _val;
}

public NextNode(int _val, NextNode _left, NextNode _right, NextNode _next) {
    val = _val;
    left = _left;
    right = _right;
    next = _next;
}

public static NextNode connect(NextNode root) {
    if (root == null) {
        return null;
    }

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

    while (!queue.isEmpty()) {
        int size = queue.size();
        NextNode prev = null;

        for (int i = 0; i < size; i++) {
            NextNode cur = queue.poll();

            if (prev != null) {
                prev.next = cur; // 连接前一个节点
            }

            prev = cur;

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

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

    return root;
}

// O(1)空间复杂度的最优解
public static NextNode connectOptimal(NextNode root) {

```

```

    if (root == null) {
        return null;
    }

    NextNode leftmost = root; // 每一层的最左节点
    while (leftmost.left != null) { // 当左子节点不为空时，说明还有下一层
        NextNode cur = leftmost;
        while (cur != null) {
            // 连接左右子节点
            cur.left.next = cur.right;
            // 连接相邻节点的子节点
            if (cur.next != null) {
                cur.right.next = cur.next.left;
            }
            cur = cur.next;
        }
        leftmost = leftmost.left; // 移动到下一层
    }

    return root;
}

/*
 * 题目 8: LeetCode 117 - 填充每个节点的下一个右侧节点指针 II
 * 题目来源: 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（不是完美二叉树）
 *
 * 解题思路:
 * 1. 使用层序遍历，与题目 116 类似
 * 2. 不同之处在于树不一定是完美二叉树
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(w) - 迭代方法, O(1) - 最优解
 * 是否为最优解: 迭代方法不是, 有 O(1) 空间的解法
 */
public static NextNode connectII(NextNode root) {
    if (root == null) {
        return null;
    }

    Queue<NextNode> queue = new LinkedList<>();
    queue.offer(root);
    while (!queue.isEmpty()) {
        int size = queue.size();
        NextNode prev = null;

```

```

        for (int i = 0; i < size; i++) {
            NextNode cur = queue.poll();
            if (prev != null) {
                prev.next = cur;
            }
            prev = cur;
            if (cur.left != null) {
                queue.offer(cur.left);
            }
            if (cur.right != null) {
                queue.offer(cur.right);
            }
        }
    }

    return root;
}

// O(1)空间复杂度的最优解
public static NextNode connectIIOptimal(NextNode root) {
    if (root == null) {
        return null;
    }

    NextNode cur = root; // 当前层的指针
    while (cur != null) {
        NextNode dummy = new NextNode(0); // 哑节点，作为下一层的头节点
        NextNode tail = dummy; // 下一层的尾节点
        // 遍历当前层的所有节点
        while (cur != null) {
            if (cur.left != null) {
                tail.next = cur.left;
                tail = tail.next;
            }
            if (cur.right != null) {
                tail.next = cur.right;
                tail = tail.next;
            }
            cur = cur.next; // 移动到当前层的下一个节点
        }
        cur = dummy.next; // 移动到下一层
    }

    return root;
}

```

```

/*
 * 题目 9: LeetCode 662 - 二叉树最大宽度
 * 题目来源: https://leetcode.cn/problems/maximum-width-of-binary-tree/
 * 题目描述:
 * 给定一个二叉树，编写一个函数来获取这个树的最大宽度。
 * 树的宽度是所有层中节点的最大数量。
 *
 * 解题思路:
 * 1. 使用层序遍历，为每个节点编号
 * 2. 每一层的宽度 = 最右边节点编号 - 最左边节点编号 + 1
 * 3. 左子节点编号 = 父节点编号 * 2
 * 4. 右子节点编号 = 父节点编号 * 2 + 1
 *
 * 时间复杂度: O(n)
 * 空间复杂度: O(w)
 * 是否为最优解: 是
 */

static class AnnotatedNode {
    TreeNode node;
    long id; // 使用 long 避免溢出
}

AnnotatedNode(TreeNode n, long i) {
    node = n;
    id = i;
}

public static int widthOfBinaryTree(TreeNode root) {
    if (root == null) {
        return 0;
    }

    Queue<AnnotatedNode> queue = new LinkedList<>();
    queue.offer(new AnnotatedNode(root, 0));
    long maxWidth = 0;

    while (!queue.isEmpty()) {
        int size = queue.size();
        long left = queue.peek().id; // 当前层最左边节点的编号
        long right = left; // 当前层最右边节点的编号
        for (int i = 0; i < size; i++) {
            AnnotatedNode cur = queue.poll();
            right = cur.id; // 更新最右边节点的编号
            if (cur.node.left != null) {
                queue.offer(new AnnotatedNode(cur.node.left, cur.id * 2));
            }
        }
        maxWidth = Math.max(maxWidth, right - left + 1);
    }

    return maxWidth;
}

```

```

    }
    if (cur.node.right != null) {
        queue.offer(new AnnotatedNode(cur.node.right, cur.id * 2 + 1));
    }
}
maxWidth = Math.max(maxWidth, right - left + 1);
}
return (int) maxWidth;
}

```

/*

* 题目 10: LeetCode 993 - 二叉树的堂兄弟节点

* 题目来源: <https://leetcode.cn/problems/cousins-in-binary-tree/>

* 题目描述:

* 在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

* 如果二叉树的两个节点深度相同，但 父节点不同，则它们是一对堂兄弟节点。

*

* 解题思路:

* 1. 使用层序遍历，记录每个节点的父节点和深度

* 2. 判断两个节点是否深度相同且父节点不同

*

* 时间复杂度: O(n)

* 空间复杂度: O(w)

* 是否为最优解: 是

*/

```
public static boolean isCousins(TreeNode root, int x, int y) {
```

```
    if (root == null) {
```

```
        return false;
```

```
}
```

```
Queue<TreeNode> queue = new LinkedList<>();
```

```
queue.offer(root);
```

```
while (!queue.isEmpty()) {
```

```
    int size = queue.size();
```

```
    TreeNode parentX = null, parentY = null;
```

```
    boolean foundX = false, foundY = false;
```

```
    for (int i = 0; i < size; i++) {
```

```
        TreeNode cur = queue.poll();
```

```
// 检查左子节点
```

```
        if (cur.left != null) {
```

```
            queue.offer(cur.left);
```

```
            if (cur.left.val == x) {
```

```
                foundX = true;
```

```
                parentX = cur;
```

```

        } else if (cur.left.val == y) {
            foundY = true;
            parentY = cur;
        }
    }

    // 检查右子节点
    if (cur.right != null) {
        queue.offer(cur.right);
        if (cur.right.val == x) {
            foundX = true;
            parentX = cur;
        } else if (cur.right.val == y) {
            foundY = true;
            parentY = cur;
        }
    }
}

// 如果在同一层找到了两个节点，判断父节点是否不同
if (foundX && foundY) {
    return parentX != parentY;
}

// 如果只找到了一个，说明不在同一层
if (foundX || foundY) {
    return false;
}
return false;
}
}

```

文件: BinaryTreeTraversalIteration.py

```
# -*- coding: utf-8 -*-
"""
二叉树遍历的迭代实现
包括前序、中序、后序和层序遍历的非递归实现
"""

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

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

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

前序遍历 - 非递归实现

算法思路:

1. 使用栈来模拟递归过程
2. 先访问根节点，然后依次访问右子树和左子树
3. 由于栈是后进先出，所以先压入右子树，再压入左子树

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，最坏情况下为 $O(n)$

"""

```
if not root:  
    return []  
  
result = []  
stack = [root]  
  
while stack:  
    node = stack.pop()  
    result.append(node.val)  
  
    # 先压入右子树，再压入左子树  
    if node.right:  
        stack.append(node.right)  
    if node.left:  
        stack.append(node.left)  
  
return result
```

```
# ===== 以下是新增的补充题目 =====
```

"""

题目: LeetCode 112 - 路径总和

题目来源: <https://leetcode.cn/problems/path-sum/>

题目描述: 给你二叉树的根节点 root 和一个表示目标和的整数 targetSum 。

判断该树中是否存在 根节点到叶子节点 的路径，这条路径上所有节点值相加等于目标和 targetSum 。

如果存在，返回 true ；否则，返回 false 。

解题思路:

1. 使用深度优先搜索（DFS）遍历二叉树
2. 从根节点开始，每次遍历到一个节点时，将当前累计和减去节点值
3. 如果到达叶子节点且累计和为 0，则返回 true
4. 否则继续递归遍历左右子树

时间复杂度: $O(n)$ – 需要遍历树中的所有节点

空间复杂度: $O(h)$ – 递归调用栈的深度， h 为树的高度

是否为最优解: 是，DFS 是解决此类路径问题的最优方法

"""

```
def has_path_sum(root, target_sum):  
    if not root:  
        return False  
  
    # 非递归 DFS 实现 – 使用栈同时存储节点和当前路径和  
    stack = [(root, target_sum - root.val)]  
  
    while stack:  
        node, remaining_sum = stack.pop()  
  
        # 如果是叶子节点且剩余和为 0，找到符合条件的路径  
        if not node.left and not node.right and remaining_sum == 0:  
            return True  
  
        # 先压入右子节点，这样保证左子节点先被处理（DFS 顺序）  
        if node.right:  
            stack.append((node.right, remaining_sum - node.right.val))  
        if node.left:  
            stack.append((node.left, remaining_sum - node.left.val))  
  
    return False
```

"""

题目: LeetCode 113 – 路径总和 II

题目来源: <https://leetcode.cn/problems/path-sum-ii/>

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

解题思路：

1. 使用回溯算法（深度优先搜索）
2. 维护一个当前路径列表，记录已经走过的节点值
3. 当到达叶子节点且路径和等于目标和时，将当前路径加入结果集
4. 否则继续递归搜索左右子树

时间复杂度： $O(n^2)$ – 每个节点访问一次，最坏情况下需要将路径复制 n 次

空间复杂度： $O(h)$ – 递归调用栈和路径列表的空间， h 为树的高度

是否为最优解：是，回溯是寻找所有路径的标准方法

```
"""
def path_sum(root, target_sum):
    result = []
    if not root:
        return result

    # 非递归 DFS 实现
    node_stack = [(root, root.val)]
    path_stack = [[root.val]]

    while node_stack:
        node, current_sum = node_stack.pop()
        current_path = path_stack.pop()

        # 如果是叶子节点且和等于目标值，加入结果集
        if not node.left and not node.right and current_sum == target_sum:
            result.append(current_path)

        # 先处理右子树（栈是后进先出，所以右子树先入栈）
        if node.right:
            right_sum = current_sum + node.right.val
            node_stack.append((node.right, right_sum))

            right_path = current_path.copy()
            right_path.append(node.right.val)
            path_stack.append(right_path)

        # 再处理左子树
        if node.left:
            left_sum = current_sum + node.left.val
            node_stack.append((node.left, left_sum))

            left_path = current_path.copy()
            left_path.append(node.left.val)
```

```
    path_stack.append(left_path)

return result
```

"""

题目：LeetCode 129 - 求根节点到叶节点数字之和

题目来源：<https://leetcode.cn/problems/sum-root-to-leaf-numbers/>

题目描述：给你一个二叉树的根节点 root，树中每个节点都存放有一个 0 到 9 之间的数字。

每条从根节点到叶节点的路径都代表一个数字。

例如，从根节点到叶节点的路径 1 → 2 → 3 表示数字 123。

计算从根节点到叶节点生成的所有数字之和。

解题思路：

1. 使用深度优先搜索遍历二叉树
2. 维护一个当前路径代表的数字
3. 当到达叶子节点时，将当前数字加入总和

时间复杂度：O(n) - 每个节点只访问一次

空间复杂度：O(h) - 递归栈的深度

是否为最优解：是，DFS 是解决此类路径问题的高效方法

"""

```
def sum_numbers(root):
    if not root:
        return 0

    # 非递归 DFS 实现
    stack = [(root, root.val)]
    total_sum = 0

    while stack:
        node, current_number = stack.pop()

        # 如果是叶子节点，将当前数字加入总和
        if not node.left and not node.right:
            total_sum += current_number
        else:
            # 非叶子节点，继续向下遍历
            if node.right:
                stack.append((node.right, current_number * 10 + node.right.val))
            if node.left:
                stack.append((node.left, current_number * 10 + node.left.val))

    return total_sum
```

"""

题目：LeetCode 257 - 二叉树的所有路径

题目来源：<https://leetcode.cn/problems/binary-tree-paths/>

题目描述：给你一个二叉树的根节点 root，按 任意顺序，

返回所有从根节点到叶子节点的路径。

解题思路：

1. 使用回溯算法（DFS）
2. 维护当前路径字符串
3. 当到达叶子节点时，将完整路径加入结果集
4. 继续递归处理左右子树

时间复杂度：O(n) – 每个节点访问一次，路径字符串拼接可能需要 O(n) 时间

空间复杂度：O(h) – 递归栈的深度

是否为最优解：是，DFS 是生成所有路径的标准方法

"""

```
def binary_tree_paths(root):  
    result = []  
    if not root:  
        return result  
  
    # 非递归 DFS 实现  
    stack = [(root, str(root.val))]  
  
    while stack:  
        node, path = stack.pop()  
  
        # 如果是叶子节点，将路径加入结果集  
        if not node.left and not node.right:  
            result.append(path)  
        else:  
            # 非叶子节点，继续向下遍历  
            if node.right:  
                stack.append((node.right, path + "->" + str(node.right.val)))  
            if node.left:  
                stack.append((node.left, path + "->" + str(node.left.val)))  
  
    return result
```

"""

题目：LeetCode 1448 - 统计二叉树中好节点的数目

题目来源：<https://leetcode.cn/problems/count-good-nodes-in-binary-tree/>

题目描述：给你一棵根为 root 的二叉树，请你返回二叉树中好节点的数目。

「好节点」X 定义为：从根到该节点 X 所经过的节点中，没有任何节点的值大于 X 的值。

解题思路：

1. 使用深度优先搜索遍历二叉树
2. 维护从根到当前节点路径上的最大值
3. 如果当前节点的值大于等于该最大值，则为好节点，更新最大值
4. 继续递归处理左右子树

时间复杂度： $O(n)$ – 每个节点只访问一次

空间复杂度： $O(h)$ – 递归栈的深度

是否为最优解：是，DFS 是解决此类路径最大值问题的最优方法

"""

```
def good_nodes(root):  
    if not root:  
        return 0  
  
    # 非递归 DFS 实现  
    stack = [(root, root.val)]  # (节点, 路径最大值)  
    good_count = 0  
  
    while stack:  
        node, max_so_far = stack.pop()  
  
        # 判断是否为好节点  
        if node.val >= max_so_far:  
            good_count += 1  
            max_so_far = node.val  # 更新路径最大值  
  
        # 继续处理左右子树  
        if node.right:  
            stack.append((node.right, max_so_far))  
        if node.left:  
            stack.append((node.left, max_so_far))  
  
    return good_count
```

"""

题目：剑指 Offer 26 – 树的子结构

题目来源：<https://leetcode.cn/problems/shu-de-zi-jie-gou-lcof/>

题目描述：输入两棵二叉树 A 和 B，判断 B 是不是 A 的子结构。

约定空树不是任意一个树的子结构。

解题思路：

1. 先序遍历树 A 中的每个节点 nA
2. 对于每个节点 nA，检查以 nA 为根节点的子树是否包含树 B
3. 检查是否包含的逻辑：递归比较节点值是否相等，左子树和右子树是否也满足条件

时间复杂度：O(m*n) – m 和 n 分别是两棵树的节点数

空间复杂度：O(h) – 递归栈的深度，h 为树 A 的高度

是否为最优解：是，需要遍历树 A 的每个节点并进行匹配

"""

```
def is_sub_structure(A, B):  
    # 空树不是任意一个树的子结构  
    if not A or not B:  
        return False  
  
    # 非递归 DFS 实现，遍历树 A 的每个节点  
    stack = [A]  
  
    while stack:  
        node = stack.pop()  
  
        # 检查以当前节点为根的子树是否包含树 B  
        if is_match(node, B):  
            return True  
  
        # 继续遍历其他节点  
        if node.right:  
            stack.append(node.right)  
        if node.left:  
            stack.append(node.left)  
  
    return False  
  
# 辅助方法：检查以 A 为根的子树是否包含以 B 为根的子树  
def is_match(A, B):  
    # 递归实现更清晰  
    if not B:  
        return True  # B 已经匹配完  
    if not A or A.val != B.val:  
        return False  # A 为空或值不匹配  
    # 继续匹配左右子树  
    return is_match(A.left, B.left) and is_match(A.right, B.right)
```

"""

题目：LeetCode 1372 - 二叉树中的最长交错路径

题目来源：<https://leetcode.cn/problems/longest-zigzag-path-in-a-binary-tree/>

题目描述：给你一棵以 `root` 为根的二叉树，返回其最长的交错路径的长度。

交错路径的定义如下：从一个节点开始，沿着父-子连接，向上或向下移动，

移动时，节点的方向必须交替变化（即从左到右，或从右到左）。

解题思路：

1. 使用深度优先搜索遍历二叉树
2. 对每个节点，记录从上一个节点来的方向（左或右）
3. 如果当前方向与上一个方向交替，则路径长度+1，否则重置为1
4. 更新全局最大路径长度

时间复杂度： $O(n)$ – 每个节点只访问一次

空间复杂度： $O(h)$ – 递归栈的深度

是否为最优解：是，一次遍历即可找到最长交错路径

”””

```
def longest_zig_zag(root):  
    if not root:  
        return 0  
  
    max_length = 0  
  
    # 非递归 DFS 实现，栈中存储三元组：(节点, 方向, 当前长度)  
    # 方向：-1 表示从父节点的左子树来, 1 表示从父节点的右子树来, 0 表示根节点  
    stack = [(root, 0, 0)]  
  
    while stack:  
        node, direction, length = stack.pop()  
  
        # 更新最大值  
        max_length = max(max_length, length)  
  
        # 处理左子树  
        if node.left:  
            new_length = length + 1 if direction == 1 else 1  
            stack.append((node.left, -1, new_length))  
  
        # 处理右子树  
        if node.right:  
            new_length = length + 1 if direction == -1 else 1  
            stack.append((node.right, 1, new_length))  
  
    return max_length
```

"""

题目：LeetCode 222 - 完全二叉树的节点个数

题目来源：<https://leetcode.cn/problems/count-complete-tree-nodes/>

题目描述：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 的定义是：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。

解题思路：

1. 利用完全二叉树的特性：如果左子树的高度等于右子树的高度，则左子树是满二叉树
2. 如果左子树的高度大于右子树的高度，则右子树是满二叉树
3. 满二叉树的节点数为 $2^h - 1$ ，其中 h 是树的高度
4. 递归计算剩余部分的节点数

时间复杂度： $O(\log^2 n)$ – 每次计算高度需要 $O(\log n)$ ，递归深度为 $O(\log n)$

空间复杂度： $O(\log n)$ – 递归栈的深度

是否为最优解：是，利用完全二叉树特性进行优化

"""

```
def count_nodes(root):  
    if not root:  
        return 0  
  
    # 计算树的高度（从根到最左边叶子节点的距离）  
    left_height = get_height(root.left)  
    right_height = get_height(root.right)  
  
    if left_height == right_height:  
        # 左子树是满二叉树，节点数为  $2^{left\_height} - 1$ ，加上根节点和右子树  
        return (1 << left_height) + count_nodes(root.right)  
    else:  
        # 右子树是满二叉树，节点数为  $2^{right\_height} - 1$ ，加上根节点和左子树  
        return (1 << right_height) + count_nodes(root.left)  
  
    # 辅助方法：计算完全二叉树的高度（从根到最左边叶子节点的距离）  
def get_height(node):  
    height = 0  
    while node:  
        height += 1  
        node = node.left  
    return height  
  
def inorder_traversal(root: Optional[TreeNode]) -> List[int]:
```

```
"""
```

中序遍历 - 非递归实现

算法思路:

1. 使用栈来模拟递归过程
2. 一直向左走到底，将路径上的节点都压入栈中
3. 弹出栈顶节点并访问，然后转向右子树

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，最坏情况下为 $O(n)$

```
"""
```

```
result = []
stack = []
current = root

while current or stack:
    if current:
        # 一直向左走到底
        stack.append(current)
        current = current.left
    else:
        # 处理栈顶节点
        current = stack.pop()
        result.append(current.val)
        # 转向右子树
        current = current.right

return result
```

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

```
"""
```

后序遍历 - 非递归实现（双栈法）

算法思路:

1. 使用两个栈
2. 第一个栈用于遍历，第二个栈用于收集结果
3. 先序遍历的变种: 根->右->左，然后反转结果

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，最坏情况下为 $O(n)$

```
"""
```

```
if not root:
```

```

    return []

result = []
stack1 = [root]
stack2 = []

# 第一个栈用于遍历
while stack1:
    node = stack1.pop()
    stack2.append(node)

    # 先压入左子树，再压入右子树
    if node.left:
        stack1.append(node.left)
    if node.right:
        stack1.append(node.right)

# 从第二个栈中弹出元素即为后序遍历结果
while stack2:
    result.append(stack2.pop().val)

return result

```

```

def postorder_traversal_one_stack(root: Optional[TreeNode]) -> List[int]:
"""

```

后序遍历 - 非递归实现（单栈法）

算法思路：

1. 使用一个栈和一个指针记录最近访问的节点
2. 一直向左走到底，将路径上的节点都压入栈中
3. 查看栈顶节点，如果右子树存在且未被访问过，则转向右子树
4. 否则访问栈顶节点，并标记为已访问

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，最坏情况下为 O(n)

```

"""

```

```

if not root:
    return []

```

```

result = []
stack = []
last_visited = None

```

```

current = root

while current or stack:
    if current:
        # 一直向左走到底
        stack.append(current)
        current = current.left
    else:
        # 查看栈顶节点
        peek_node = stack[-1]
        # 如果右子树存在且未被访问过
        if peek_node.right and last_visited != peek_node.right:
            current = peek_node.right
        else:
            # 访问栈顶节点
            result.append(peek_node.val)
            last_visited = peek_node
            stack.pop()

return result

```

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

```
"""

```

层序遍历（广度优先遍历）

算法思路：

1. 使用队列进行广度优先搜索
2. 每次处理一层的所有节点
3. 将下一层的节点加入队列

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，最坏情况下为 O(n)

```
"""

```

```
if not root:
```

```
    return []
```

```
result = []
```

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

```
while queue:
```

```
    level_size = len(queue) # 当前层的节点数
```

```
    level_nodes = [] # 存储当前层的节点值
```

```

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

    # 将下一层的节点加入队列
    if node.left:
        queue.append(node.left)
    if node.right:
        queue.append(node.right)

result.append(level_nodes)

return result

```

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

```
"""

```

锯齿形层序遍历

算法思路：

1. 类似层序遍历，但需要交替改变每层的遍历方向
2. 使用一个布尔变量控制方向

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，最坏情况下为 O(n)

```
"""

```

```

if not root:
    return []

result = []
queue = deque([root])
left_to_right = True # 控制遍历方向

```

```

while queue:
    level_size = len(queue)
    level_nodes = [0] * level_size

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

        # 根据方向决定插入位置

```

```

index = i if left_to_right else level_size - 1 - i
level_nodes[index] = node.val

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

result.append(level_nodes)
left_to_right = not left_to_right # 切换方向

return result

```

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

```
"""

```

计算二叉树的最大深度

算法思路：

1. 使用层序遍历计算层数
2. 每处理完一层，深度加 1

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，最坏情况下为 O(n)

```
"""

```

```
if not root:
    return 0
```

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

```
depth = 0
```

```
while queue:
```

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

```
    # 处理当前层的所有节点
```

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

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

```
        if node.left:
```

```
            queue.append(node.left)
```

```
        if node.right:
```

```
            queue.append(node.right)
```

```
    depth += 1 # 每处理完一层，深度加 1
```

```
    return depth
```

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

```
    """
```

```
翻转二叉树
```

算法思路：

1. 使用层序遍历访问每个节点
2. 交换每个节点的左右子树

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，最坏情况下为 $O(n)$

```
"""
```

```
if not root:
```

```
    return None
```

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

```
while queue:
```

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

```
# 交换左右子树
```

```
    node.left, node.right = node.right, node.left
```

```
# 将非空子节点加入队列
```

```
    if node.left:
```

```
        queue.append(node.left)
```

```
    if node.right:
```

```
        queue.append(node.right)
```

```
return root
```

```
def main():
```

```
    """测试函数"""

```

```
# 构建测试二叉树：
```

```
#      1
```

```
#      / \

```

```
#      2   3

```

```
#      / \ / \

```

```
#      4  5 6  7

```

```
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)

print("== 二叉树遍历测试 ==")

# 前序遍历
preorder = preorder_traversal(root)
print(f"前序遍历: {' '.join(map(str, preorder))}")

# 中序遍历
inorder = inorder_traversal(root)
print(f"中序遍历: {' '.join(map(str, inorder))}")

# 后序遍历 (双栈法)
postorder1 = postorder_traversal_two_stacks(root)
print(f"后序遍历(双栈法): {' '.join(map(str, postorder1))}")

# 后序遍历 (单栈法)
postorder2 = postorder_traversal_one_stack(root)
print(f"后序遍历(单栈法): {' '.join(map(str, postorder2))}")

# 层序遍历
levelorder = level_order(root)
print(f"层序遍历: {' '.join([str(level) for level in levelorder])}")

# 锯齿形层序遍历
zigzag = zigzag_level_order(root)
print(f"锯齿形层序遍历: {' '.join([str(level) for level in zigzag])}")

# 二叉树的最大深度
depth = max_depth(root)
print(f"二叉树的最大深度: {depth}")

# 翻转二叉树
inverted = invert_tree(root)
inverted_level_order = level_order(inverted)
print(f"翻转后的层序遍历: {' '.join([str(level) for level in inverted_level_order])}")
```

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

```
# ===== 以下是新增的补充题目 =====
```

```
,,
```

题目 1: LeetCode 107 - 二叉树的层序遍历 II

题目来源: <https://leetcode.cn/problems/binary-tree-level-order-traversal-ii/>

题目描述:

给定一个二叉树，返回其节点值自底向上的层序遍历。

解题思路:

1. 使用队列进行正常的层序遍历
2. 将每一层的结果添加到列表中
3. 最后将列表反转即可得到自底向上的结果

时间复杂度: $O(n)$ – 需要遍历所有 n 个节点一次

空间复杂度: $O(n)$ – 队列最多存储树的最大宽度(最坏情况下为 $n/2$)

是否为最优解: 是

```
,,
```

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

```
    if not root:
```

```
        return []
```

```
    result = []
```

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

```
    while queue:
```

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

```
        level_nodes = []
```

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

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

```
            level_nodes.append(node.val)
```

```
            if node.left:
```

```
                queue.append(node.left)
```

```
            if node.right:
```

```
                queue.append(node.right)
```

```
    result.append(level_nodes)

# 反转结果
return result[::-1]

,,,
```

题目 2: LeetCode 637 - 二叉树的层平均值

题目来源: <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。

解题思路:

1. 使用层序遍历
2. 对每一层的节点值求和, 然后除以节点数

时间复杂度: $O(n)$

空间复杂度: $O(w)$, w 为最大宽度

是否为最优解: 是

,,,

```
def average_of_levels(root: Optional[TreeNode]) -> List[float]:
```

```
    if not root:
```

```
        return []
```

```
    result = []
```

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

```
    while queue:
```

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

```
        level_sum = 0
```

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

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

```
            level_sum += node.val
```

```
            if node.left:
```

```
                queue.append(node.left)
```

```
            if node.right:
```

```
                queue.append(node.right)
```

```
        result.append(level_sum / level_size)
```

```
    return result
```

,,

题目 3: LeetCode 515 – 在每个树行中找最大值

题目来源: <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 , 请找出存在于每一层的最大值。

解题思路:

1. 使用层序遍历
2. 对每一层记录最大值

时间复杂度: $O(n)$

空间复杂度: $O(w)$

是否为最优解: 是

,,

```
def largest_values(root: Optional[TreeNode]) -> List[int]:  
    if not root:  
        return []  
  
    result = []  
    queue = deque([root])  
  
    while queue:  
        level_size = len(queue)  
        max_val = float('-inf') # 初始化为最小值  
  
        for _ in range(level_size):  
            node = queue.popleft()  
            max_val = max(max_val, node.val)  
  
            if node.left:  
                queue.append(node.left)  
            if node.right:  
                queue.append(node.right)  
  
        result.append(int(max_val))  
  
    return result
```

,,

题目 4: LeetCode 513 – 找树左下角的值

题目来源: <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 最底层 最左边 节点的值。

解题思路:

1. 使用层序遍历
2. 记录每一层的第一个节点
3. 最后一层的第一个节点就是答案

时间复杂度: $O(n)$

空间复杂度: $O(w)$

是否为最优解: 是

,,,

```
def find_bottom_left_value(root: Optional[TreeNode]) -> int:  
    queue = deque([root])  
    leftmost = root.val  
  
    while queue:  
        level_size = len(queue)  
  
        for i in range(level_size):  
            node = queue.popleft()  
  
            if i == 0: # 记录每一层的第一个节点  
                leftmost = node.val  
  
            if node.left:  
                queue.append(node.left)  
            if node.right:  
                queue.append(node.right)  
  
    return leftmost
```

,,,

题目 5: LeetCode 662 - 二叉树最大宽度

题目来源: <https://leetcode.cn/problems/maximum-width-of-binary-tree/>

题目描述:

给定一个二叉树，编写一个函数来获取这个树的最大宽度。

树的宽度是所有层中节点的最大数量。

解题思路:

1. 使用层序遍历，为每个节点编号

2. 每一层的宽度 = 最右边节点编号 - 最左边节点编号 + 1
3. 左子节点编号 = 父节点编号 * 2
4. 右子节点编号 = 父节点编号 * 2 + 1

时间复杂度: $O(n)$

空间复杂度: $O(w)$

是否为最优解: 是

, , ,

```
def width_of_binary_tree(root: Optional[TreeNode]) -> int:  
    if not root:  
        return 0  
  
    queue = deque([(root, 0)]) # (node, id)  
    max_width = 0  
  
    while queue:  
        level_size = len(queue)  
        _, left = queue[0] # 当前层最左边节点的编号  
  
        for i in range(level_size):  
            node, node_id = queue.popleft()  
  
            if node.left:  
                queue.append((node.left, node_id * 2))  
            if node.right:  
                queue.append((node.right, node_id * 2 + 1))  
  
        # 计算当前层的宽度  
        max_width = max(max_width, node_id - left + 1)  
  
    return max_width
```
