宫水三叶的刷题日记

二叉树

Author ：宫水三叶
Date ：2021/10/07
QQ Group: 703311589
WeChat ：oaoaya

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

噔噔噔噔，这是公众号「宫水三叶的刷题日记」的原创专题「二叉树」合集。

本合集更新时间为 2021-10-07，大概每 2-4 周会集中更新一次。关注公众号，后台回复「二叉树」即可获取最新下载链接。

💡下面介绍使用本合集的最佳使用实践：

学习算法：

1. 打开在线目录（Github 版 & Gitee 版）；
2. 从侧边栏的类别目录找到「二叉树」；
3. 按照「推荐指数」从大到小进行刷题，「推荐指数」相同，则按照「难度」从易到难进行刷题‘
4. 拿到题号之后，回到本合集进行检索。

维持熟练度：

1. 按照本合集「从上往下」进行刷题。

学习过程中遇到任何困难，欢迎加入「每日一题打卡 QQ 群：703311589」进行交流 🍭🍭🍭

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **297. 二叉树的序列化与反序列化** ，难度为 **困难**。

Tag：「二叉树」、「层序遍历」

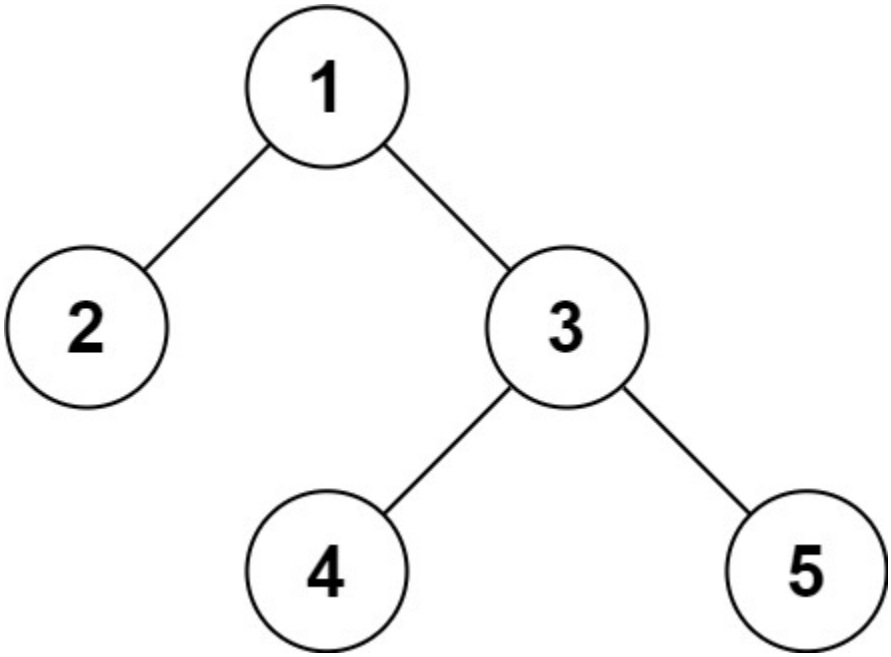
序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

提示: 输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的

格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

示例 1：




```
输入：root = [1,2,3,null,null,4,5]
输出：[1,2,3,null,null,4,5]
```

示例 2：

```
输入：root = []
输出：[]
```

示例 3：

```
输入：root = [1]
输出：[1]
```

示例 4：

```
输入：root = [1,2]
输出：[1,2]
```

提示：

- 树中结点数在范围 $[0, 10^4]$ 内

- -1000 <= Node.val <= 1000

基本思路

无论使用何种「遍历方式」进行二叉树存储，为了方便，我们都需要对空节点有所表示。

其实题目本身的样例就给我们提供了很好的思路：**使用层序遍历的方式进行存储，对于某个叶子节点的空节点进行存储，同时确保不递归存储空节点对应的子节点。**

层序遍历

根据节点值的数据范围 `-1000 <= Node.val <= 1000`（我是在 297. 二叉树的序列化与反序列化 看的，你也可以不使用数字，使用某个特殊字符进行表示，只要能在反序列时有所区分即可），我们可以建立占位节点 `emptyNode` 用来代指空节点（`emptyNode.val = INF`）。

序列化：先特判掉空树的情况，之后就是常规的层序遍历逻辑：

1. 起始时，将 `root` 节点入队；
2. 从队列中取出节点，检查节点是否有左/右节点：
    - 如果有的话，将值追加序列化字符中（注意使用分隔符），并将节点入队；
    - 如果没有，检查当前节点是否为 `emptyNode` 节点，如果不是 `emptyNode` 说明是常规的叶子节点，需要将其对应的空节点进行存储，即将 `emptyNode` 入队；
3. 循环流程 2，直到整个队列为空。

反序列：同理，怎么「序列化」就怎么进行「反序列」即可：

1. 起始时，构造出 `root` 并入队；
2. 每次从队列取出元素时，同时从序列化字符中截取两个值（对应左右节点），检查是否为 `INF`，若不为 `INF` 则构建对应节点；
3. 循环流程 2，直到整个序列化字符串被处理完（注意跳过最后一位分隔符）。

代码：

```Java
public class Codec {
    int INF = -2000;
    TreeNode emptyNode = new TreeNode(INF);
    public String serialize(TreeNode root) {
        if (root == null) return "";

        StringBuilder sb = new StringBuilder();
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        while (!d.isEmpty()) {
            TreeNode poll = d.pollFirst();
            sb.append(poll.val + "_");
            if (!poll.equals(emptyNode)) {
                d.addLast(poll.left != null ? poll.left : emptyNode);
                d.addLast(poll.right != null ? poll.right : emptyNode);
            }
        }
        return sb.toString();
    }

    public TreeNode deserialize(String data) {
        if (data.equals("")) return null;

        String[] ss = data.split("_");
        int n = ss.length;
        TreeNode root = new TreeNode(Integer.parseInt(ss[0]));
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        for (int i = 1; i < n - 1; i += 2) {
            TreeNode poll = d.pollFirst();
            int a = Integer.parseInt(ss[i]), b = Integer.parseInt(ss[i + 1]);
            if (a != INF) {
                poll.left = new TreeNode(a);
                d.addLast(poll.left);
            }
            if (b != INF) {
                poll.right = new TreeNode(b);
                d.addLast(poll.right);
            }
        }
        return root;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

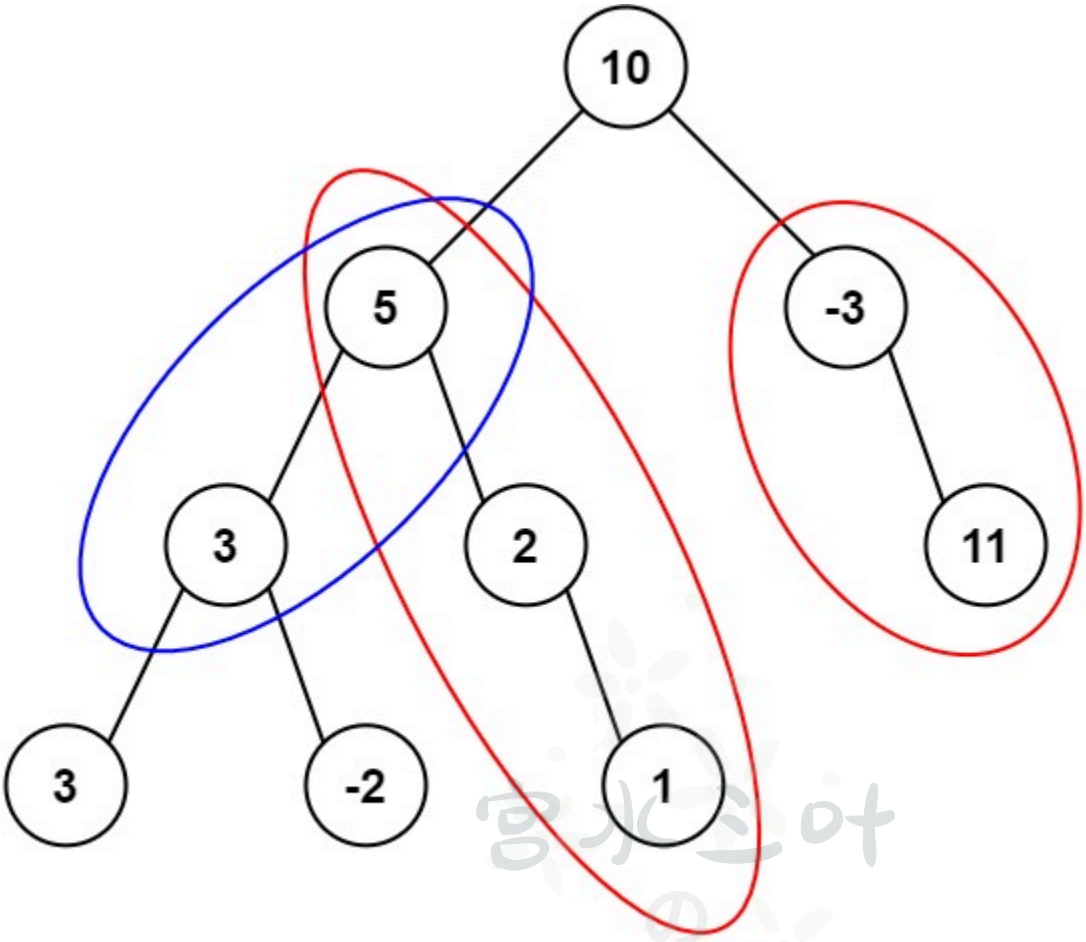
这是 LeetCode 上的 **437. 路径总和 III** ，难度为 **中等**。

Tag：「DFS」、「树的遍历」、「前缀和」

给定一个二叉树的根节点 root ，和一个整数 targetSum ，求该二叉树里节点值之和等于 targetSum 的 路径 的数目。

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

示例 1：



```
输入：root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8

输出：3

解释：和等于 8 的路径有 3 条，如图所示。
```

示例 2：

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

输出：3
```

提示:

- 二叉树的节点个数的范围是 [0,1000]
- -10^9 <= Node.val <= 10^9
- -1000 <= targetSum <= 1000

树的遍历 + DFS

一个朴素的做法是搜索以每个节点为根的（往下的）所有路径，并对路径总和为 $targetSum$ 的路径进行累加统计。

使用 `dfs1` 来搜索所有节点，复杂度为 $O(n)$；在 `dfs1` 中对于每个当前节点，使用 `dfs2` 搜索以其为根的所有（往下的）路径，同时累加路径总和为 $targetSum$ 的所有路径，复杂度为 $O(n)$。

整体复杂度为 $O(n^2)$，数据范围为 10^3，可以过。

代码：

```Java
class Solution {
    int ans, t;
    public int pathSum(TreeNode root, int _t) {
        t = _t;
        dfs1(root);
        return ans;
    }
    void dfs1(TreeNode root) {
        if (root == null) return;
        dfs2(root, root.val);
        dfs1(root.left);
        dfs1(root.right);
    }
    void dfs2(TreeNode root, int val) {
        if (val == t) ans++;
        if (root.left != null) dfs2(root.left, val + root.left.val);
        if (root.right != null) dfs2(root.right, val + root.right.val);
    }
}
```

- 时间复杂度：$O(n^2)$
- 空间复杂度：忽略递归带来的额外空间开销，复杂度为 $O(1)$

树的遍历 + 前缀和

在「解法一」中，我们统计的是以每个节点为根的（往下的）所有路径，也就是说统计的是以每个节点为「路径开头」的所有合法路径。

本题的一个优化切入点为「路径只能往下」，因此如果我们转换一下，统计以每个节点为「路径结尾」的合法数量的话，配合原本就是「从上往下」进行的数的遍历（最完整的路径必然是从原始根节点到当前节点的唯一路径），相当于只需要在完整路径中找到有多少个节点到当前节点的路径总和为 $targetSum$。

于是这个树上问题彻底转换一维问题：求解从原始起点（根节点）到当前节点 b 的路径中，有多少节点 a 满足 $sum[a...b] = targetSum$，由于从原始起点（根节点）到当前节点的路径唯一，因此这其实是一个「一维前缀和」问题。

具体的，我们可以在进行树的遍历时，记录下从原始根节点 $root$ 到当前节点 cur 路径中，从 $root$ 到任意中间节点 x 的路径总和，配合哈希表，快速找到满足以 cur 为「路径结尾」的、

使得路径总和为 $targetSum$ 的目标「路径起点」有多少个。

> 一些细节：由于我们只能统计往下的路径，但是树的遍历会同时搜索两个方向的子树。因此我们应当在搜索完以某个节点为根的左右子树之后，应当回溯地将路径总和从哈希表中删除，防止统计到跨越两个方向的路径。

代码：

```
class Solution {
    Map<Integer, Integer> map = new HashMap<>();
    int ans, t;
    public int pathSum(TreeNode root, int _t) {
        if (root == null) return 0;
        t = _t;
        map.put(0, 1);
        dfs(root, root.val);
        return ans;
    }
    void dfs(TreeNode root, int val) {
        if (map.containsKey(val - t)) ans += map.get(val - t);
        map.put(val, map.getOrDefault(val, 0) + 1);
        if (root.left != null) dfs(root.left, val + root.left.val);
        if (root.right != null) dfs(root.right, val + root.right.val);
        map.put(val, map.getOrDefault(val, 0) - 1);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

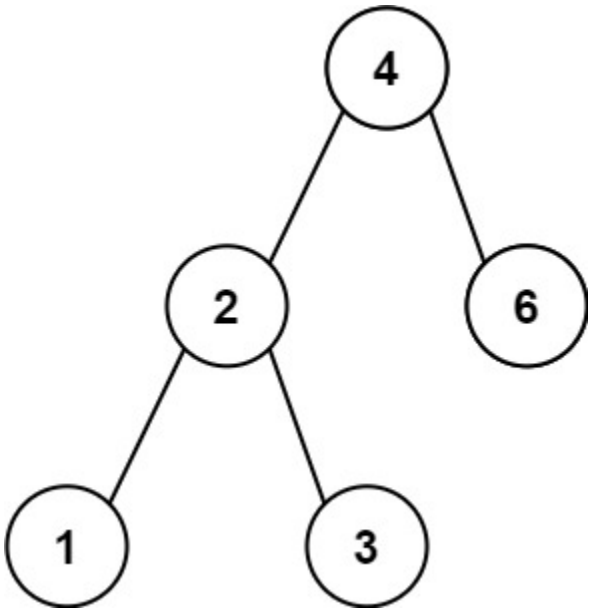
这是 LeetCode 上的 **783. 二叉搜索树节点最小距离** ，难度为 **简单**。

Tag：「树的搜素」、「迭代」、「非迭代」、「中序遍历」、「BFS」、「DFS」

给你一个二叉搜索树的根节点 root ，返回 树中任意两不同节点值之间的最小差值 。

注意：本题与 530：https://leetcode-cn.com/problems/minimum-absolute-difference-in-bst/ 相同

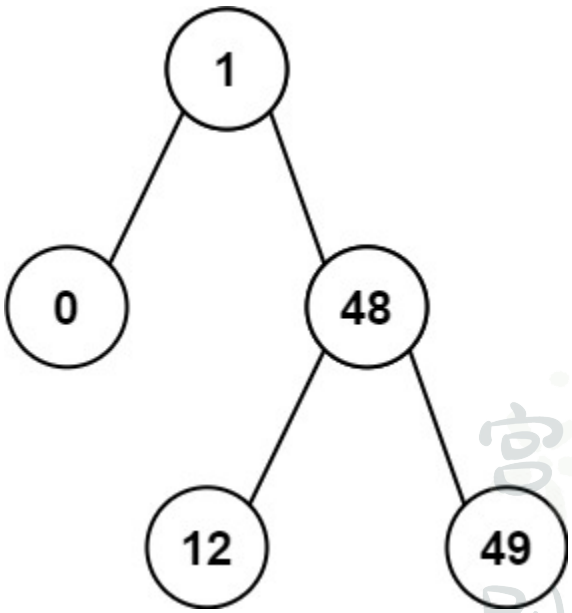
示例 1：




```
输入：root = [4,2,6,1,3]

输出：1
```

示例 2：



```
输入：root = [1,0,48,null,null,12,49]

输出：1
```

提示：

- 树中节点数目在范围 [2, 100] 内
- 0 <= Node.val <= 10^5
- 差值是一个正数，其数值等于两值之差的绝对值

朴素解法（BFS & DFS）

如果不考虑利用二叉搜索树特性的话，一个朴素的做法是将所有节点的 val 存到一个数组中。

对数组进行排序，并获取答案。

将所有节点的 val 存入数组，可以使用 BFS 或者 DFS。

代码：

```Java
class Solution {
    public int minDiffInBST(TreeNode root) {
        List<Integer> list = new ArrayList<>();

        // BFS
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        while (!d.isEmpty()) {
            TreeNode poll = d.pollFirst();
            list.add(poll.val);
            if (poll.left != null) d.addLast(poll.left);
            if (poll.right != null) d.addLast(poll.right);
        }

        // DFS
        // dfs(root, list);

        Collections.sort(list);
        int n = list.size();
        int ans = Integer.MAX_VALUE;
        for (int i = 1; i < n; i++) {
            int cur = Math.abs(list.get(i) - list.get(i - 1));
            ans = Math.min(ans, cur);
        }
        return ans;
    }
    void dfs(TreeNode root, List<Integer> list) {
        list.add(root.val);
        if (root.left != null) dfs(root.left, list);
        if (root.right != null) dfs(root.right, list);
    }
}
```

- 时间复杂度：$O(n \log n)$
- 空间复杂度：$O(n)$

中序遍历（栈模拟 & 递归）

不难发现，在朴素解法中，我们对树进行搜索的目的是为了获取一个「有序序列」，然后从「有序序列」中获取答案。

而二叉搜索树的中序遍历是有序的，因此我们可以直接对「二叉搜索树」进行中序遍历，保存遍

历过程中的相邻元素最小值即是答案。

代码：

```
class Solution {
    int ans = Integer.MAX_VALUE;
    TreeNode prev = null;
    public int minDiffInBST(TreeNode root) {
        // 栈模拟
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (prev != null) {
                ans = Math.min(ans, Math.abs(prev.val - root.val));
            }
            prev = root;
            root = root.right;
        }

        // 递归
        // dfs(root);

        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        dfs(root.left);
        if (prev != null) {
            ans = Math.min(ans, Math.abs(prev.val - root.val));
        }
        prev = root;
        dfs(root.right);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **863. 二叉树中所有距离为 K 的结点** ，难度为 **中等**。

Tag：「图论 BFS」、「图论 DFS」、「二叉树」

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

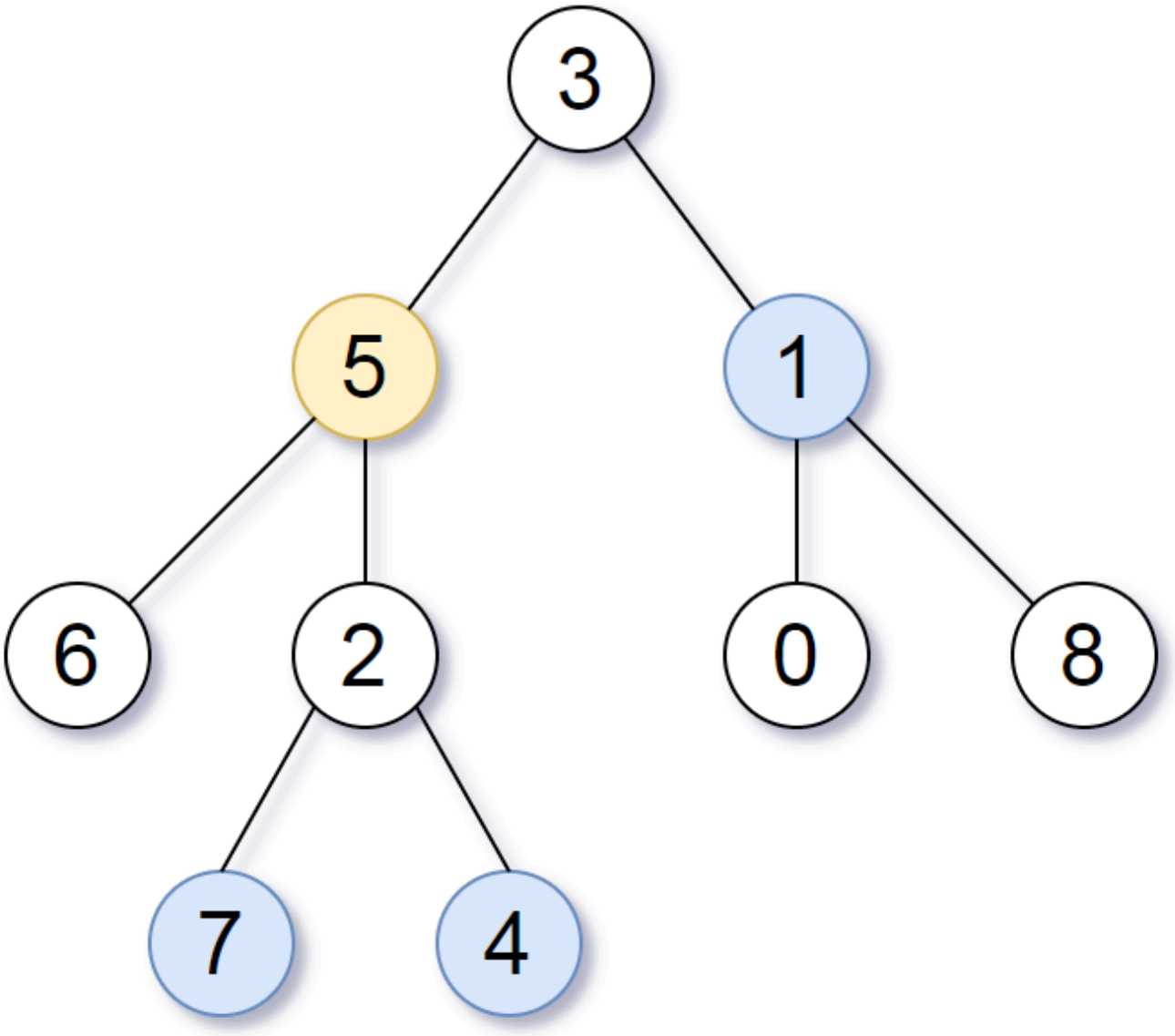
返回到目标结点 target 距离为 K 的所有结点的值的列表。 答案可以以任何顺序返回。

示例 1：

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

输出：[7,4,1]

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



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

提示：

- **给定的树是非空的。**
- **树上的每个结点都具有唯一的值** 0 <= node.val <= 500 **。**
- **目标结点** target **是树上的结点。**
- 0 <= K <= 1000.

基本分析

显然，如果题目是以图的形式给出的话，我们可以很容易通过「`BFS` / 迭代加深」找到距离为 k 的节点集。

而树是一类特殊的图，我们可以通过将二叉树转换为图的形式，再进行「`BFS` / 迭代加深」。

由于二叉树每个点最多有 2 个子节点，点和边的数量接近，属于稀疏图，因此我们可以使用「邻接表」的形式进行存储。

建图方式为：对于二叉树中相互连通的节点（`root` 与 `root.left`、`root` 和 `root.right`），建立一条无向边。

建图需要遍历整棵树，使用 `DFS` 或者 `BFS` 均可。

由于所有边的权重均为 1，我们可以使用「`BFS` / 迭代加深」找到从目标节点 `target` 出发，与目标节点距离为 k 的节点，然后将其添加到答案中。

> 一些细节：利用每个节点具有唯一的值，我们可以直接使用节点值进行建图和搜索。

建图 + `BFS`

由「基本分析」，可写出「建图 + `BFS`」的实现。

代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        Deque<Integer> d = new ArrayDeque<>();
        d.addLast(t.val);
        vis[t.val] = true;
        while (!d.isEmpty() && k >= 0) {
            int size = d.size();
            while (size-- > 0) {
                int poll = d.pollFirst();
                if (k == 0) {
                    ans.add(poll);
                    continue;
                }
                for (int i = he[poll]; i != -1 ; i = ne[i]) {
                    int j = e[i];
                    if (!vis[j]) {
                        d.addLast(j);
                        vis[j] = true;
                    }
                }
            }
            k--;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
```

```
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过 `BFS` 找到距离 $target$ 为 k 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

建图 + 迭代加深

由「基本分析」，可写出「建图 + 迭代加深」的实现。

迭代加深的形式，我们只需要结合题意，搜索深度为 k 的这一层即可。

代码：

```java
class Solution {
    int N = 1010, M = N * 2;
    int[] he = new int[N], e = new int[M], ne = new int[M];
    int idx;
    void add(int a, int b) {
        e[idx] = b;
        ne[idx] = he[a];
        he[a] = idx++;
    }
    boolean[] vis = new boolean[N];
    public List<Integer> distanceK(TreeNode root, TreeNode t, int k) {
        List<Integer> ans = new ArrayList<>();
        Arrays.fill(he, -1);
        dfs(root);
        vis[t.val] = true;
        find(t.val, k, 0, ans);
        return ans;
    }
    void find(int root, int max, int cur, List<Integer> ans) {
        if (cur == max) {
            ans.add(root);
            return ;
        }
        for (int i = he[root]; i != -1; i = ne[i]) {
            int j = e[i];
            if (!vis[j]) {
                vis[j] = true;
                find(j, max, cur + 1, ans);
            }
        }
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        if (root.left != null) {
            add(root.val, root.left.val);
            add(root.left.val, root.val);
            dfs(root.left);
        }
        if (root.right != null) {
            add(root.val, root.right.val);
            add(root.right.val, root.val);
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：通过 `DFS` 进行建图的复杂度为 $O(n)$；通过迭代加深找到距离 $target$ 为 k 的节点，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

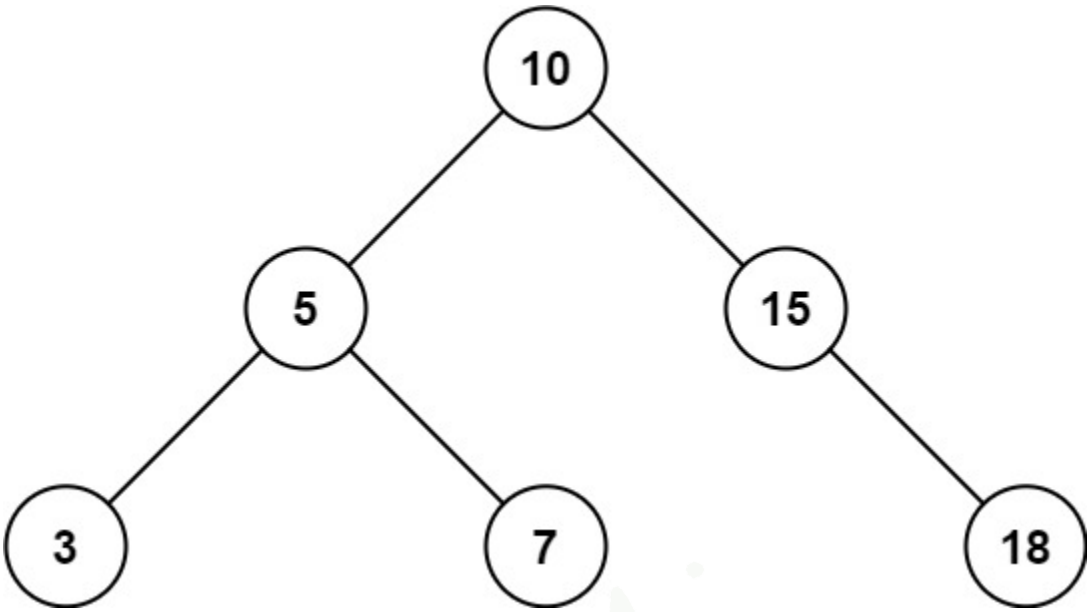
题目描述

这是 LeetCode 上的 **938. 二叉搜索树的范围和** ，难度为 **简单**。

Tag : 「树的搜索」、「DFS」、「BFS」

给定二叉搜索树的根结点 root，返回值位于范围 [low, high] 之间的所有结点的值的和。

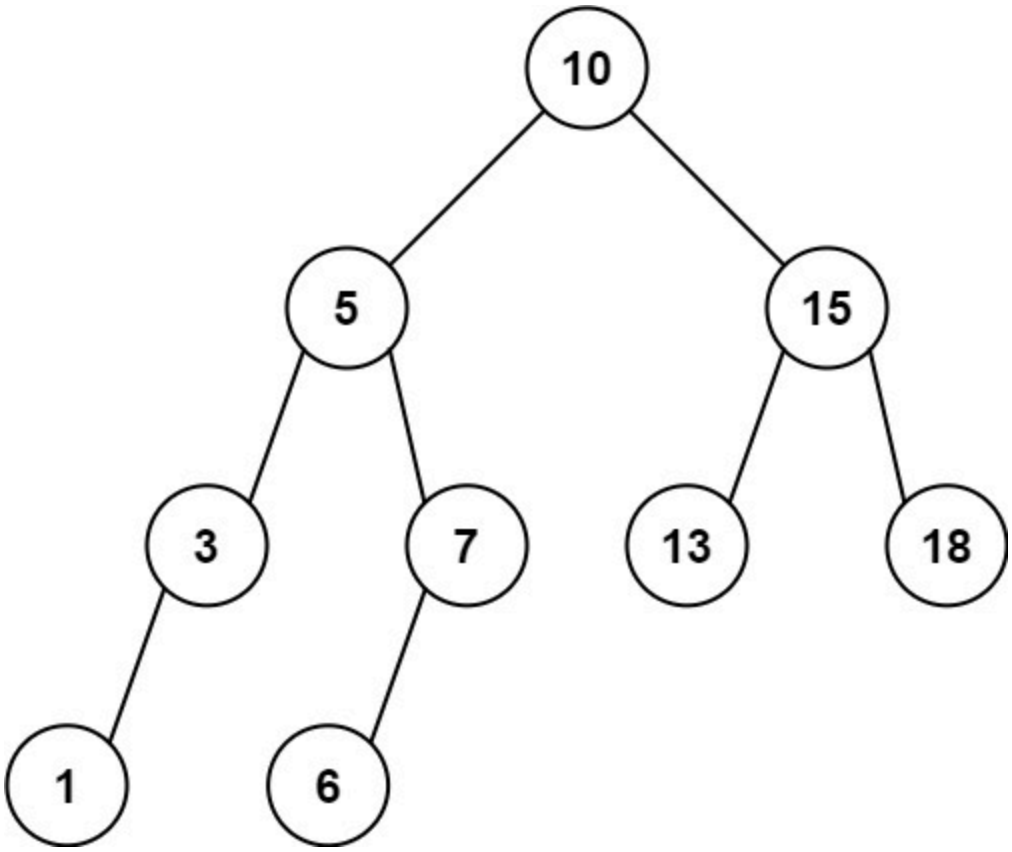
示例 1：




```
输入：root = [10,5,15,3,7,null,18], low = 7, high = 15

输出：32
```

示例 2：




```
输入：root = [10,5,15,3,7,13,18,1,null,6], low = 6, high = 10

输出：23
```

提示：

- 树中节点数目在范围 [1, 2 * 10^4] 内
- 1 <= Node.val <= 10^5
- 1 <= low <= high <= 10^5
- 所有 Node.val 互不相同

基本思路

这又是众多「二叉搜索树遍历」题目中的一道。

二叉搜索树的中序遍历是有序的。

只要对其进行「中序遍历」即可得到有序列表，在遍历过程中判断节点值是否符合要求，对于符

合要求的节点值进行累加即可。

二叉搜索树的「中序遍历」有「迭代」和「递归」两种形式。**由于给定了值范围 $[low, high]$，因此可以在遍历过程中做一些剪枝操作，但并不影响时空复杂度。**

递归

递归写法十分简单，属于树的遍历中最简单的实现方式。

代码：

```
class Solution {
    int low, high;
    int ans;
    public int rangeSumBST(TreeNode root, int _low, int _high) {
        low = _low; high = _high;
        dfs(root);
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        dfs(root.left);
        if (low <= root.val && root.val <= high) ans += root.val;
        dfs(root.right);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

迭代

迭代其实就是使用「栈」来模拟递归过程，也属于树的遍历中的常见实现形式。

一般简单的面试中如果问到树的遍历，面试官都不会对「递归」解法感到满意，因此掌握「迭代/非递归」写法同样重要。

代码：

```java
class Solution {
    public int rangeSumBST(TreeNode root, int low, int high) {
        int ans = 0;
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (low <= root.val && root.val <= high) {
                ans += root.val;
            }
            root = root.right;
        }
        return ans;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

> **🍥 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **987. 二叉树的垂序遍历** ，难度为 **困难**。

Tag：「数据结构运用」、「二叉树」、「哈希表」、「排序」、「优先队列」、「DFS」

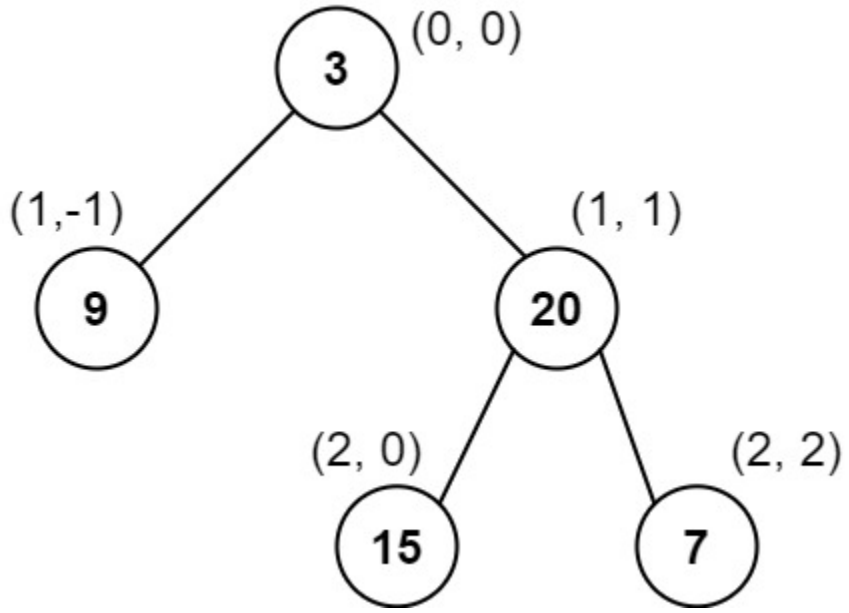
给你二叉树的根结点 root ，请你设计算法计算二叉树的 垂序遍历 序列。

对位于 (row, col) 的每个结点而言，其左右子结点分别位于 (row + 1, col - 1) 和 (row + 1, col + 1) 。树的根结点位于 (0, 0) 。

二叉树的 垂序遍历 从最左边的列开始直到最右边的列结束，按列索引每一列上的所有结点，形成一个按出现位置从上到下排序的有序列表。如果同行同列上有多个结点，则按结点的值从小到大进行排序。

返回二叉树的 垂序遍历 序列。

示例 1：

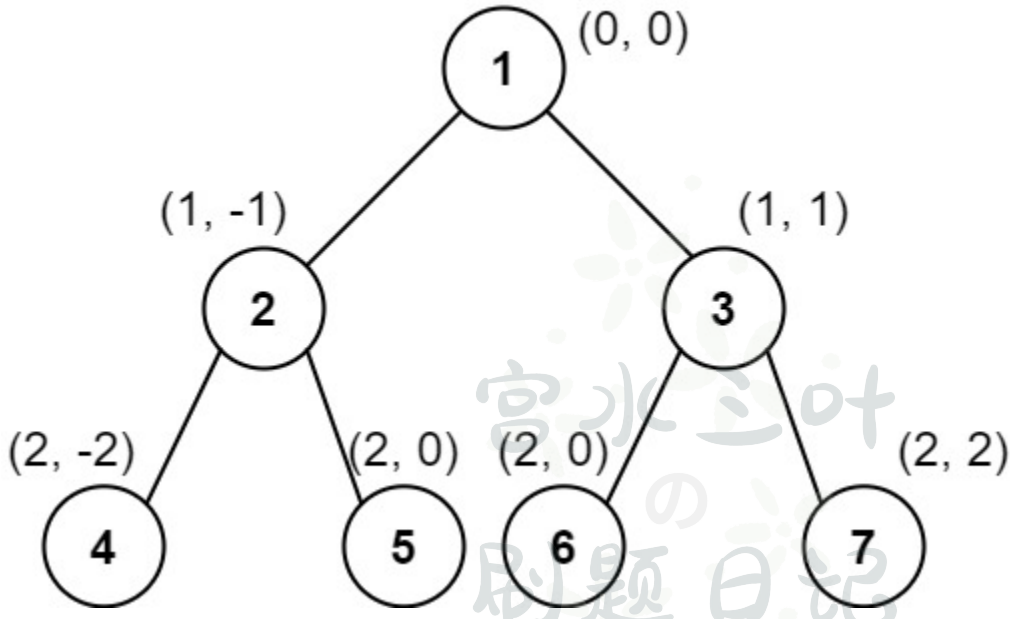



```
输入：root = [3,9,20,null,null,15,7]

输出：[[9],[3,15],[20],[7]]

解释：
列 -1 ：只有结点 9 在此列中。
列  0 ：只有结点 3 和 15 在此列中，按从上到下顺序。
列  1 ：只有结点 20 在此列中。
列  2 ：只有结点 7 在此列中。
```

示例 2：

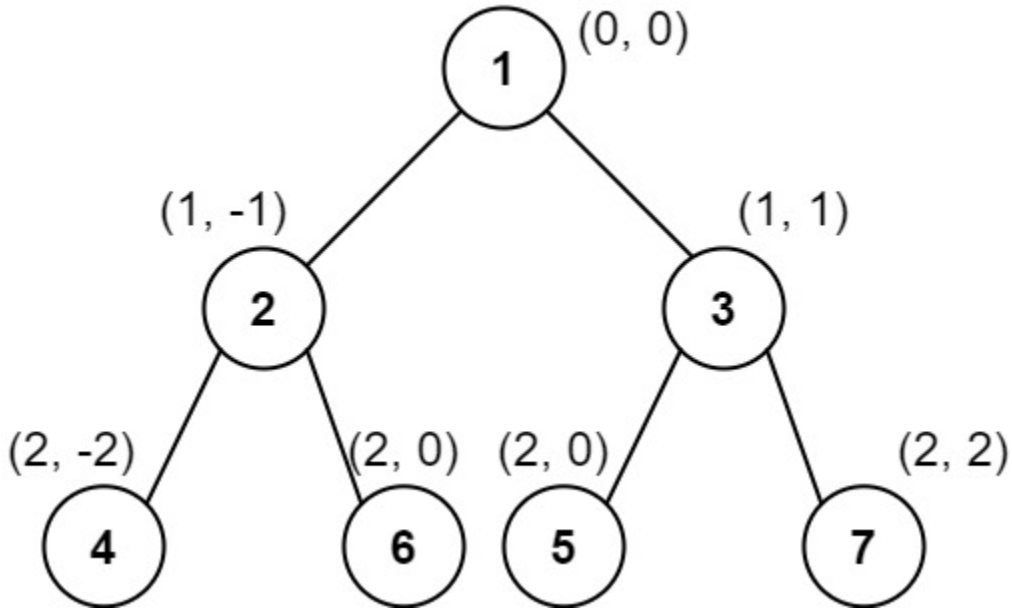


输入：root = [1,2,3,4,5,6,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
列 -2 ：只有结点 4 在此列中。
列 -1 ：只有结点 2 在此列中。
列 0 ：结点 1 、5 和 6 都在此列中。
1 在上面，所以它出现在前面。
5 和 6 位置都是 (2, 0) ，所以按值从小到大排序，5 在 6 的前面。
列 1 ：只有结点 3 在此列中。
列 2 ：只有结点 7 在此列中。

示例 3：



输入：root = [1,2,3,4,6,5,7]

输出：[[4],[2],[1,5,6],[3],[7]]

解释：
这个示例实际上与示例 2 完全相同，只是结点 5 和 6 在树中的位置发生了交换。
因为 5 和 6 的位置仍然相同，所以答案保持不变，仍然按值从小到大排序。

提示：

- 树中结点数目总数在范围 [1, 10]

DFS + 哈希表 + 排序

根据题意，我们需要按照优先级「**“列号从小到大”，对于同列节点，“行号从小到大”，对于同列同行元素，“节点值从小到大”**」进行答案构造。

因此我们可以对树进行遍历，遍历过程中记下这些信息 (col, row, val)，然后根据规则进行排序，并构造答案。

我们可以先使用「哈希表」进行存储，最后再进行一次性的排序。

代码：

```Java
class Solution {
    Map<TreeNode, int[]> map = new HashMap<>(); // col, row, val
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        map.put(root, new int[]{0, 0, root.val});
        dfs(root);
        List<int[]> list = new ArrayList<>(map.values());
        Collections.sort(list, (a, b)->{
            if (a[0] != b[0]) return a[0] - b[0];
            if (a[1] != b[1]) return a[1] - b[1];
            return a[2] - b[2];
        });
        int n = list.size();
        List<List<Integer>> ans = new ArrayList<>();
        for (int i = 0; i < n; ) {
            int j = i;
            List<Integer> tmp = new ArrayList<>();
            while (j < n && list.get(j)[0] == list.get(i)[0]) tmp.add(list.get(j++)[2]);
            ans.add(tmp);
            i = j;
        }
        return ans;
    }
    void dfs(TreeNode root) {
        if (root == null) return ;
        int[] info = map.get(root);
        int col = info[0], row = info[1], val = info[2];
        if (root.left != null) {
            map.put(root.left, new int[]{col - 1, row + 1, root.left.val});
            dfs(root.left);
        }
        if (root.right != null) {
            map.put(root.right, new int[]{col + 1, row + 1, root.right.val});
            dfs(root.right);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，填充哈希表时进行树的遍历，复杂度为 $O(n)$；构造答案时需要进行排序，复杂度为 $O(n \log n)$。整体复杂度为 $O(n \log n)$
- 空间复杂度：$O(n)$

DFS + 优先队列（堆）

显然，最终要让所有节点的相应信息有序，可以使用「优先队列（堆）」边存储边维护有序性。

代码：

```
class Solution {
    PriorityQueue<int[]> q = new PriorityQueue<>((a, b)->{ // col, row, val
        if (a[0] != b[0]) return a[0] - b[0];
        if (a[1] != b[1]) return a[1] - b[1];
        return a[2] - b[2];
    });
    public List<List<Integer>> verticalTraversal(TreeNode root) {
        int[] info = new int[]{0, 0, root.val};
        q.add(info);
        dfs(root, info);
        List<List<Integer>> ans = new ArrayList<>();
        while (!q.isEmpty()) {
            List<Integer> tmp = new ArrayList<>();
            int[] poll = q.peek();
            while (!q.isEmpty() && q.peek()[0] == poll[0]) tmp.add(q.poll()[2]);
            ans.add(tmp);
        }
        return ans;
    }
    void dfs(TreeNode root, int[] fa) {
        if (root.left != null) {
            int[] linfo = new int[]{fa[0] - 1, fa[1] + 1, root.left.val};
            q.add(linfo);
            dfs(root.left, linfo);
        }
        if (root.right != null) {
            int[] rinfo = new int[]{fa[0] + 1, fa[1] + 1, root.right.val};
            q.add(rinfo);
            dfs(root.right, rinfo);
        }
    }
}
```

- 时间复杂度：令总节点数量为 n，将节点信息存入优先队列（堆）复杂度为 $O(n\log n)$；构造答案复杂度为 $O(n\log n)$。整体复杂度为 $O(n\log n)$
- 空间复杂度：$O(n)$

> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **993. 二叉树的堂兄弟节点** ，难度为 **简单**。

Tag：「树的搜索」、「BFS」、「DFS」

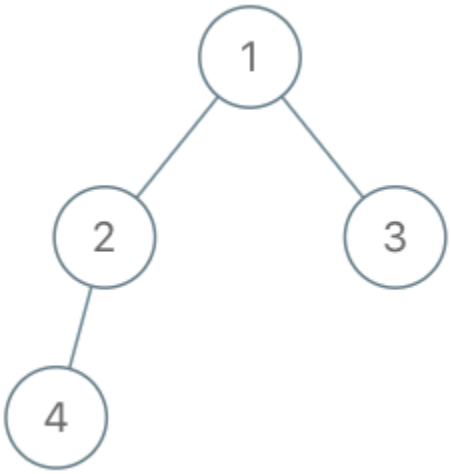
在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

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。

我们给出了具有唯一值的二叉树的根节点 root ，以及树中两个不同节点的值 x 和 y 。

只有与值 x 和 y 对应的节点是堂兄弟节点时，才返回 true 。否则，返回 false。

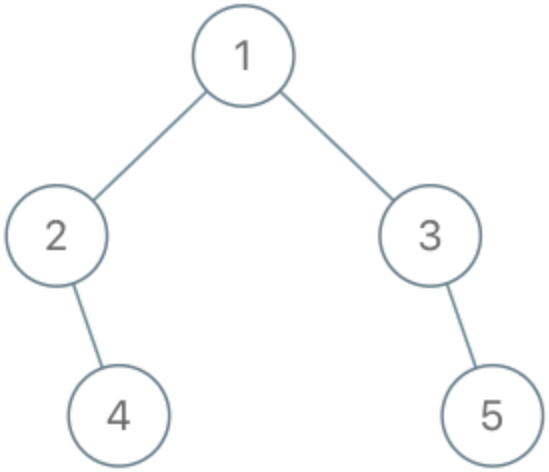
示例 1：




```
输入：root = [1,2,3,4], x = 4, y = 3

输出：false
```

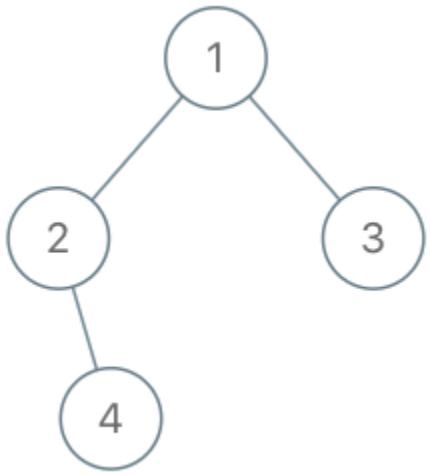
示例 2：




```
输入：root = [1,2,3,null,4,null,5], x = 5, y = 4

输出：true
```

示例 3：




```
输入：root = [1,2,3,null,4], x = 2, y = 3

输出：false
```

提示：

- 二叉树的节点数介于 2 到 100 之间。
- 每个节点的值都是唯一的、范围为 1 到 100 的整数。

DFS

显然，我们希望得到某个节点的「父节点」&「所在深度」，不难设计出如下「DFS 函数签名」：

```
/**
 * 查找 t 的「父节点值」&「所在深度」
 * @param root 当前搜索到的节点
 * @param fa root 的父节点
 * @param depth 当前深度
 * @param t 搜索目标值
 * @return [fa.val, depth]
 */
int[] dfs(TreeNode root, TreeNode fa, int depth, int t);
```

之后按照遍历的逻辑处理即可。

需要注意的时，我们需要区分出「搜索不到」和「搜索对象为 root（没有 fa 父节点）」两种情况。

我们约定使用 -1 代指没有找到目标值 t，使用 0 代表找到了目标值 t，但其不存在父节点。

代码：

```
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        int[] xi = dfs(root, null, 0, x);
        int[] yi = dfs(root, null, 0, y);
        return xi[1] == yi[1] && xi[0] != yi[0];
    }
    int[] dfs(TreeNode root, TreeNode fa, int depth, int t) {
        if (root == null) return new int[]{-1, -1}; // 使用 -1 代表为搜索不到 t
        if (root.val == t) {
            return new int[]{fa != null ? fa.val : 1, depth}; // 使用 1 代表搜索值 t 为 root
        }
        int[] l = dfs(root.left, root, depth + 1, t);
        if (l[0] != -1) return l;
        return dfs(root.right, root, depth + 1, t);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：忽略递归开销为 $O(1)$，否则为 $O(n)$

BFS

能使用 `DFS`，自然也能使用 `BFS`，两者大同小异。

代码：

```
class Solution {
    public boolean isCousins(TreeNode root, int x, int y) {
        int[] xi = bfs(root, x);
        int[] yi = bfs(root, y);
        return xi[1] == yi[1] && xi[0] != yi[0];
    }
    int[] bfs(TreeNode root, int t) {
        Deque<Object[]> d = new ArrayDeque<>(); // 存储值为 [cur, fa, depth]
        d.addLast(new Object[]{root, null, 0});
        while (!d.isEmpty()) {
            int size = d.size();
            while (size-- > 0) {
                Object[] poll = d.pollFirst();
                TreeNode cur = (TreeNode)poll[0], fa = (TreeNode)poll[1];
                int depth = (Integer)poll[2];

                if (cur.val == t) return new int[]{fa != null ? fa.val : 0, depth};
                if (cur.left != null) d.addLast(new Object[]{cur.left, cur, depth + 1});
                if (cur.right != null) d.addLast(new Object[]{cur.right, cur, depth + 1});
            }
        }
        return new int[]{-1, -1};
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

题目描述

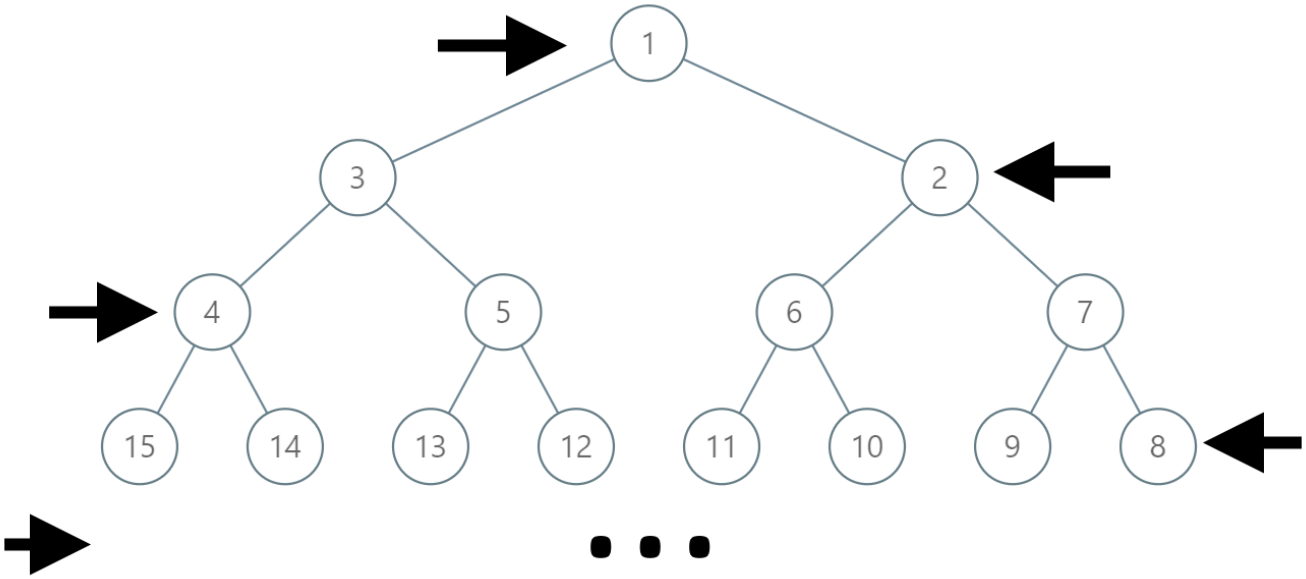
这是 LeetCode 上的 **1104. 二叉树寻路** ，难度为**中等**。

Tag：「二叉树」、「模拟」、「数学」

在一棵无限的二叉树上，每个节点都有两个子节点，树中的节点 逐行 依次按 “之” 字形进行标记。

如下图所示，在奇数行（即，第一行、第三行、第五行……）中，按从左到右的顺序进行标记；

而偶数行（即，第二行、第四行、第六行……）中，按从右到左的顺序进行标记。



给你树上某一个节点的标号 label，请你返回从根节点到该标号为 label 节点的路径，该路径是由途经的节点标号所组成的。

示例 1：

```
输入：label = 14

输出：[1,3,4,14]
```

示例 2：

```
输入：label = 26

输出：[1,2,6,10,26]
```

提示：

- 1 <= label <= 10^6

模拟

一个朴素的做法是根据题意进行模拟。

利用从根节点到任意一层都是满二叉树，我们可以先确定 `label` 所在的层级 `level`，然后计算出当前层起始节点值（最小值）和结束节点值（最大值）。

再利用「每层节点数量翻倍」&「隔层奇偶性翻转」，寻址出上一层的节点下标（令每层下标均「从左往右」计算，并从 1 开始），直到构造出答案（寻址到根节点）。

代码：

```java
class Solution {
    // 第 level 层的起始节点值
    int getStart(int level) {
        return (int)Math.pow(2, level - 1);
    }
    // 第 level 层的结束节点值
    int getEnd(int level) {
        int a = getStart(level);
        return a + a - 1;
    }
    public List<Integer> pathInZigZagTree(int n) {
        // 计算 n 所在层级
        int level = 1;
        while (getEnd(level) < n) level++;

        int[] ans = new int[level];
        int idx = level - 1, cur = n;
        while (idx >= 0) {
            ans[idx--] = cur;
            int tot = (int)Math.pow(2, level - 1);
            int start = getStart(level), end = getEnd(level);
            if (level % 2 == 0) {
                // 当前层为偶数层，则当前层节点「从右往左」数值递增，相应计算上一层下标也应该「从右往左」
                int j = tot / 2;
                for (int i = start; i <= end; i += 2, j--) {
                    if (i == cur || (i + 1) == cur) break;
                }
                int prevStart = getStart(level - 1);
                while (j-- > 1) prevStart++;
                cur = prevStart;
            } else {
                // 当前层为奇数层，则当前层节点「从左往右」数值递增，相应计算上一层下标也应该「从左往右」
                int j = 1;
                for (int i = start; i <= end; i += 2, j++) {
                    if (i == cur || (i + 1) == cur) break;
                }
                int prevEnd = getEnd(level - 1);
                while (j-- > 1) prevEnd--;
                cur = prevEnd;
            }
            level--;
        }
        List<Integer> list = new ArrayList<>();
        for (int i : ans) list.add(i);
        return list;
    }
```

```
}
```

- 时间复杂度：确定 n 所在层级复杂度为 $O(\log n)$；构造答案最坏情况下每个节点会被遍历一次，复杂度为 $O(n)$
- 空间复杂度：$O(1)$

数学

上述解法复杂度上界取决于「由当前行节点位置确定上层位置」的线性遍历。

如果二叉树本身不具有奇偶性翻转的话，显然某个节点 x 的父节点为 $\lfloor x/2 \rfloor$，但事实上存在奇偶性翻转，而在解法一中我们已经可以 $O(1)$ 计算某一层的起始值和结束值，有了「起始值 & 结算值」和「当前节点所在层的相对位置」，只需要利用“对称性”找到父节点在上层的相应位置，然后根据相应位置算出父节点值即可。

代码：

```java
class Solution {
    int getStart(int level) {
        return (int)Math.pow(2, level - 1);
    }
    int getEnd(int level) {
        int a = getStart(level);
        return a + a - 1;
    }
    public List<Integer> pathInZigZagTree(int n) {
        int level = 1;
        while (getEnd(level) < n) level++;
        int[] ans = new int[level];
        int idx = level - 1, cur = n;
        while (idx >= 0) {
            ans[idx--] = cur;
            int loc = ((1 << (level)) - 1 - cur) >> 1;
            cur = (1 << (level - 2)) + loc;
            level--;
        }
        List<Integer> list = new ArrayList<>();
        for (int i : ans) list.add(i);
        return list;
    }
}
```

- 时间复杂度：复杂度上界取决于确定 n 所在层级。复杂度为 $O(\log n)$
- 空间复杂度：$O(1)$

💡更新 Tips：本专题更新时间为 2021-10-07，大概每 2-4 周 集中更新一次。

最新专题合集资料下载，可关注公众号「宫水三叶的刷题日记」，回台回复「二叉树」获取下载链接。

觉得专题不错，可以请作者吃糖 🍭🍭🍭 ：

"给作者手机充个电"

YOLO 的赞赏码