宫水三叶的刷题日记

树的搜索

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「树的搜索」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「树的搜索」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「树的搜索」；
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

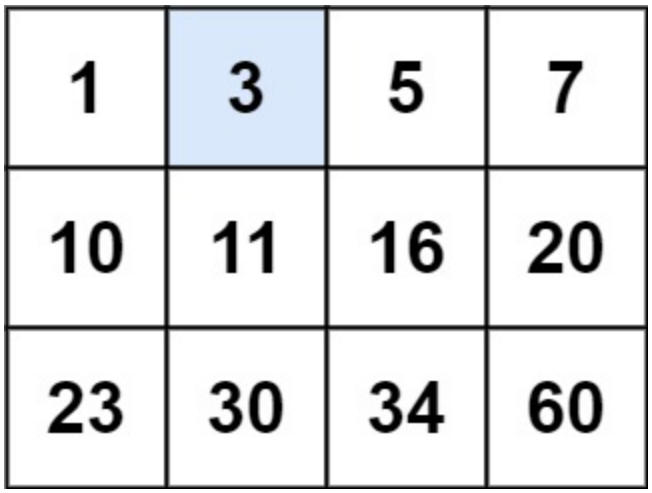
这是 LeetCode 上的 **74. 搜索二维矩阵** ，难度为 **中等**。

Tag : 「二叉搜索树」、「二分」

编写一个高效的算法来判断 m x n 矩阵中，是否存在一个目标值。该矩阵具有如下特性：

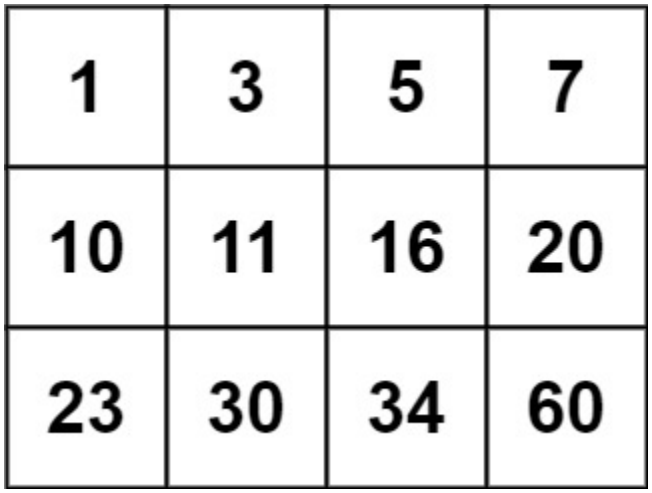
每行中的整数从左到右按升序排列。
每行的第一个整数大于前一行的最后一个整数。

示例 1：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

输出：true
```

示例 2：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

输出：false
```

提示：

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 100
- -10^4 <= matrix[i][j], target <= 10^4

二分解法（一）

由于二维矩阵固定列的「从上到下」或者固定行的「从左到右」都是升序的。

因此我们可以使用两次二分来定位到目标位置：

1. 第一次二分：从第 0 列中的「所有行」开始找，找到合适的行 `row`
2. 第二次二分：从 `row` 中「所有列」开始找，找到合适的列 `col`

代码：

```java
class Solution {
    public boolean searchMatrix(int[][] mat, int t) {
        int m = mat.length, n = mat[0].length;

        // 第一次二分：定位到所在行（从上往下，找到最后一个满足 mat[x][0] <= t 的行号）
        int l = 0, r = m - 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (mat[mid][0] <= t) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }

        int row = r;
        if (mat[row][0] == t) return true;
        if (mat[row][0] > t) return false;

        // 第二次二分：从所在行中定位到列（从左到右，找到最后一个满足 mat[row][x] <= t 的列号）
        l = 0; r = n - 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (mat[row][mid] <= t) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        int col = r;

        return mat[row][col] == t;
    }
}
```

- 时间复杂度：$O(\log m + \log n)$
- 空间复杂度：$O(1)$

二分解法（二）

当然，因为将二维矩阵的行尾和行首连接，也具有单调性。

我们可以将「二维矩阵」当做「一维矩阵」来做。

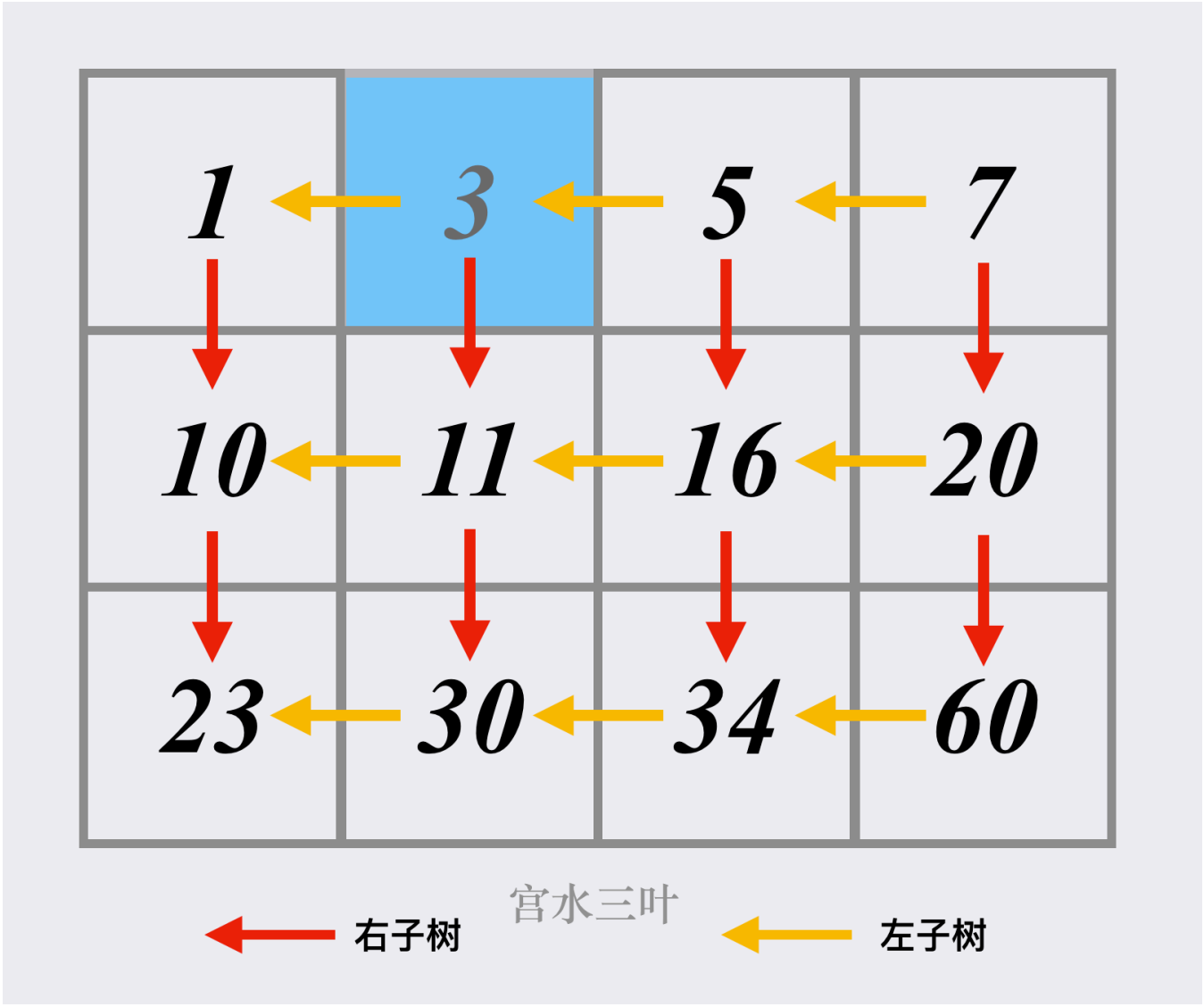
代码：

```
class Solution {
    public boolean searchMatrix(int[][] mat, int t) {
        int m = mat.length, n = mat[0].length;
        int l = 0, r = m * n - 1;
        while (l < r) {
            int mid = l + r + 1 >> 1;
            if (mat[mid / n][mid % n] <= t) {
                l = mid;
            } else {
                r = mid - 1;
            }
        }
        return mat[r / n][r % n] == t;
    }
}
```

- 时间复杂度：$O(\log (m * n))$
- 空间复杂度：$O(1)$

抽象 BST 解法

我们可以将二维矩阵抽象成「以右上角为根的 BST」：



那么我们可以从根（右上角）开始搜索，如果当前的节点不等于目标值，可以按照树的搜索顺序进行：

1. 当前节点「大于」目标值，搜索当前节点的「左子树」，也就是当前矩阵位置的「左方格子」，即 y--
2. 当前节点「小于」目标值，搜索当前节点的「右子树」，也就是当前矩阵位置的「下方格子」，即 x++

代码：

```Java
class Solution {
    int m, n;
    public boolean searchMatrix(int[][] mat, int t) {
        m = mat.length; n = mat[0].length;
        int x = 0, y = n - 1;
        while (check(x, y) && mat[x][y] != t) {
            if (mat[x][y] > t) {
                y--;
            } else {
                x++;
            }
        }
        return check(x, y) && mat[x][y] == t;
    }
    boolean check(int x, int y) {
        return x >= 0 && x < m && y >= 0 && y < n;
    }
}
```

- 时间复杂度：$O(m + n)$
- 空间复杂度：$O(1)$

拓展

如果你掌握了上述解法的话，你还可以试试这题：

240. 搜索二维矩阵 II

> **🍭 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 **173. 二叉搜索树迭代器** ，难度为 **中等**。

Tag：「树的搜索」、「中序遍历」

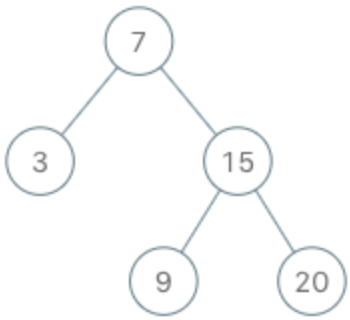
实现一个二叉搜索树迭代器类BSTIterator ，表示一个按中序遍历二叉搜索树（BST）的迭代器：

- BSTIterator(TreeNode root) 初始化 BSTIterator 类的一个对象。BST 的根节点 root 会作为构造函数的一部分给出。指针应初始化为一个不存在于 BST 中的数字，且该数字小于 BST 中的任何元素。
- boolean hasNext() 如果向指针右侧遍历存在数字，则返回 true ；否则返回 false 。
- int next()将指针向右移动，然后返回指针处的数字。

注意，指针初始化为一个不存在于 BST 中的数字，所以对 next() 的首次调用将返回 BST 中的最小元素。

你可以假设 next() 调用总是有效的，也就是说，当调用 next() 时，BST 的中序遍历中至少存在一个下一个数字。

示例：




```
输入
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
输出
[null, 3, 7, true, 9, true, 15, true, 20, false]

解释
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next();    // 返回 3
bSTIterator.next();    // 返回 7
bSTIterator.hasNext(); // 返回 True
bSTIterator.next();    // 返回 9
bSTIterator.hasNext(); // 返回 True
bSTIterator.next();    // 返回 15
bSTIterator.hasNext(); // 返回 True
bSTIterator.next();    // 返回 20
bSTIterator.hasNext(); // 返回 False
```

提示：

- 树中节点的数目在范围 [1, 10^5] 内
- 0 <= Node.val <= 10^6
- 最多调用 10^5 次 hasNext 和 next 操作

进阶：

- 你可以设计一个满足下述条件的解决方案吗？next() 和 hasNext() 操作均摊时间复杂度为 $O(1)$ ，并使用 $O(h)$ 内存。其中 h 是树的高度。

基本思路

这道题本质上考的是「将迭代版的中序遍历代码」做等价拆分。

我们知道，中序遍历的基本逻辑是：处理左子树 -> 处理当前节点 -> 处理右子树。

其中迭代做法是利用「栈」进行处理：

1. 先将当前节点的所有左子树压入栈，压到没有为止
2. 将最后一个压入的节点弹出（栈顶元素），加入答案
3. 将当前弹出的节点作为当前节点，重复步骤一

相应的裸题在这里：94. 二叉树的中序遍历

中序遍历的迭代代码：

```java
class Solution {
    List<Integer> ans = new ArrayList<>();
    Deque<TreeNode> d = new ArrayDeque<>();
    public List<Integer> inorderTraversal(TreeNode root) {
        while (root != null || !d.isEmpty()) {
            // 步骤 1
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }

            // 步骤 2
            root = d.pollLast();
            ans.add(root.val);

            // 步骤 3
            root = root.right;
        }
        return ans;
    }
}
```

总的来说是这么一个迭代过程：步骤 1 -> 步骤 2 -> 步骤 3 -> 步骤 1 ...

「中序遍历」代码的「等价拆分」

首先因为 `next()` 方法中我们需要输出一个值，执行的的是「步骤 2」的逻辑，同时我们需要在其前后添加「步骤 1」和「步骤 3」。

另外，我们还有一个 `hasNext()` 要处理，显然 `hasNext()` 应该对应我们的栈是否为空。

为此，我们**需要确保每次输出之后「步骤 1」被及时执行。**

综上，我们应该在初始化时，走一遍「步骤 1」，然后在 `next()` 方法中走「步骤 2」、「步骤 3」和「步骤 1」。

代码：

```java
class BSTIterator {
    Deque<TreeNode> d = new ArrayDeque<>();
    public BSTIterator(TreeNode root) {
        // 步骤 1
        dfsLeft(root);
    }

    public int next() {
        // 步骤 2
        TreeNode root = d.pollLast();
        int ans = root.val;
        // 步骤 3
        root = root.right;
        // 步骤 1
        dfsLeft(root);
        return ans;
    }

    void dfsLeft(TreeNode root) {
        while (root != null) {
            d.addLast(root);
            root = root.left;
        }
    }

    public boolean hasNext() {
        return !d.isEmpty();
    }
}
```

- 时间复杂度：由于每个元素都是严格「进栈」和「出栈」一次，复杂度为均摊 $O(1)$
- 空间复杂度：栈内最多保存与深度一致的节点数量，复杂度为 $O(h)$

进阶

事实上，我们空间复杂度也能做到 $O(1)$，该如何做呢？

题目描述

这是 LeetCode 上的 **331. 验证二叉树的前序序列化** ，难度为 **中等**。

Tag : 「二叉树」

序列化二叉树的一种方法是使用前序遍历。当我们遇到一个非空节点时，我们可以记录下这个节点的值。如果它是一个空节点，我们可以使用一个标记值记录，例如 #。

```
     _9_
    /   \
   3     2
  / \   / \
 4   1  #  6
/ \ / \   / \
# # # #   # #
```

例如，上面的二叉树可以被序列化为字符串 “9,3,4,#,#,1,#,#,2,#,6,#,#”，其中 # 代表一个空节点。

给定一串以逗号分隔的序列，验证它是否是正确的二叉树的前序序列化。编写一个在不重构树的条件下的可行算法。

每个以逗号分隔的字符或为一个整数或为一个表示 null 指针的 ‘#’ 。

你可以认为输入格式总是有效的，例如它永远不会包含两个连续的逗号，比如 “1,3” 。

示例 1:

```
输入: "9,3,4,#,#,1,#,#,2,#,6,#,#"
输出: true
```

示例 2:

```
输入: "1,#"
输出: false
```

示例 3:

```
输入: "9,#,#,1"
输出: false
```

二叉树规律解法

事实上，我们能利用「二叉树」的特性来做。

由于每一个非空节点都对应了 2 个出度，空节点都对应了 0 个出度；除了根节点，每个节点都有一个入度。

我们可以使用 `in` 和 `out` 来分别记录「入度」和「出度」的数量；`m` 和 `n` 分别代表「非空节点数量」和「空节点数量」。

同时，一颗合格的二叉树最终结果必然满足 `in == out`。

但我们又不能只利用最终 `in == out` 来判断是否合法，这很容易可以举出反例：考虑将一个合法序列的空节点全部提前，这样最终结果仍然满足 `in == out`，但这样的二叉树是不存在的。

我们还需要一些额外的特性，支持我们在遍历过程中提前知道一颗二叉树不合法。

例如，我们可以从合格二叉树的前提出发，挖掘遍历过程中 `in` 和 `out` 与 `n` 和 `m` 的关系。

证明 1（利用不等式）

我们令非空节点数量为 m，空节点数量为 n，入度和出度仍然使用 in 和 out 代表。

找一下 in 和 out 与 n 和 m 之间的关系。

一颗合格二叉树 m 和 n 的最小的比例关系是 1 : 2，也就是对应了这么一个形状：

```
 4
/ \
# #
```

而遍历过程中 m 和 n 的最小的比例关系则是 1 : 0，**这其实对应了二叉树空节点总是跟在非空节点的后面这一性质。**

换句话说，在没到最后一个节点之前，我们是不会遇到 空节点数量 > 非空节点数量 的情况的。

非空节点数量 >= 空节点数量 在遍历没结束前恒成立：m>=n

然后再结合「**每一个非空节点都对应了 2 个出度，空节点都对应了 0 个出度；除了根节点，每个节点都有一个入度**」特性。

在遍历尚未结束前，我们有以下关系：

1. $m >= n$
2. $in <= m + n - 1$
3. $out <= 2 * m$

简单的变形可得：

- 由 2 变形可得：$m >= in + 1 - n$
- 由 3 变形可得：$m >= out/2$

即有：

1. $m >= n$
2. $m >= in + 1 - n$
3. $m >= out/2$

再将 1 和 2 相加，抵消 `n`：$2m >= in + 1$

1. $2m >= in + 1$ => $in <= 2m - 1$
2. $m >= out/2$ => $out <= 2m$

因此，在遍历尚未完成时，`in` 和 `out` 始终满足上述关系（与空节点数量 `n` 无关）。

如果不从合格二叉树的前提（$m >= n$）出发，我们是无法得到上述关系式的。

因此，我们可以一边遍历一边统计「严格出度」和「严格入度」，然后写一个 check 函数去判定 `in` `out` `m` 三者关系是否符合要求，如果不符合则说明二叉树不合法。

```Java
class Solution {
  public boolean isValidSerialization(String s) {
    String[] ss = s.split(",");
    int n = ss.length;
    int in = 0, out = 0;
    for (int i = 0, m = 0; i < n; i++) {
      // 统计「严格出度」和「严格入度」...
      if (i != n - 1 && !check(m, in, out)) return false;
    }
    return in == out;
  }
  boolean check(int m, int in, int out) {
    boolean a = (in <= 2 * m - 1), b = (out <= 2 * m);
    return a && b;
  }
}
```

注意：因为我们这里的证明使用到的是不等式。因此统计的必须是「严格出度」&「严格入度」，不能假定一个「非空节点（非根）」必然对应两个「出度」和一个「入度」。

要想统计出「严格出度」&「严格入度」在编码上还是有一定难度的。那么是否可以推导出更加简单性质来使用呢？

请看「证明 2」。

证明 2（利用技巧转换为等式）

我们令非空节点数量为 `m`，空节点数量为 `n`，入度和出度仍然使用 `in` 和 `out` 代表。

找一下 `in` 和 `out` 与 `n` 和 `m` 之间的关系。

一颗合格二叉树 `m` 和 `n` 的最小的比例关系是 `1 : 2`，也就是对应了这么一个形状：

```
 4
/ \
# #
```

而遍历过程中 `m` 和 `n` 的最小的比例关系则是 `1 : 0`，这**其实对应了二叉树空节点总是跟在非空节点的后面这一性质。**

换句话说，在没到最后一个节点之前，我们是不会遇到 `空节点数量 > 非空节点数量` 的情况的。

`非空节点数量 >= 空节点数量` **在遍历没结束前恒成立：**$m >= n$

之后我们再采用一个技巧，就是遍历过程中每遇到一个「非空节点」就增加两个「出度」和一个「入度」，每遇到一个「空节点」只增加一个「入度」。而不管每个「非空节点」是否真实对应两个子节点。

那么我们的起始条件变成：

1. $m >= n$
2. $in = m + n - 1$
3. $out = 2 * m$

从第 2 个等式出发，结合第 1 个等式：

$$in = m + n - 1 <= m + m - 1 = 2m - 1 = out - 1$$

即可得 $in + 1 <= out$ ，也就是 `in < out` 恒成立。

代码：

```
class Solution {
    public boolean isValidSerialization(String s) {
        String[] ss = s.split(",");
        int n = ss.length;
        int in = 0, out = 0;
        for (int i = 0; i < n; i++) {
            if (!ss[i].equals("#")) out += 2;
            if (i != 0) in++;
            if (i != n - 1 && out <= in) return false;
        }
        return in == out;
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

题目描述

这是 LeetCode 上的 **671. 二叉树中第二小的节点** ，难度为 **简单**。

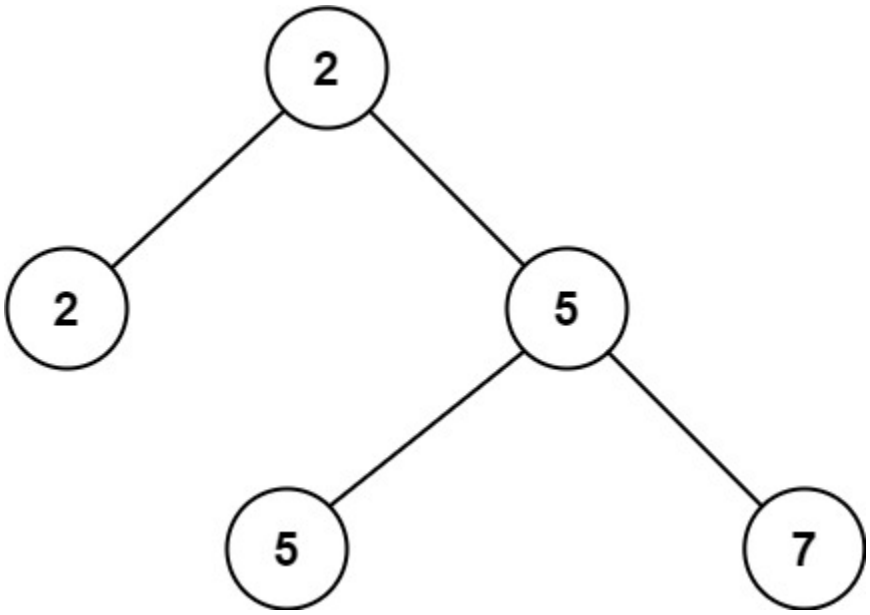
Tag：「二叉树」、「树的遍历」、「递归」

给定一个非空特殊的二叉树，每个节点都是正数，并且每个节点的子节点数量只能为 2 或 0。如果一个节点有两个子节点的话，那么该节点的值等于两个子节点中较小的一个。

更正式地说，root.val = min(root.left.val, root.right.val) 总成立。

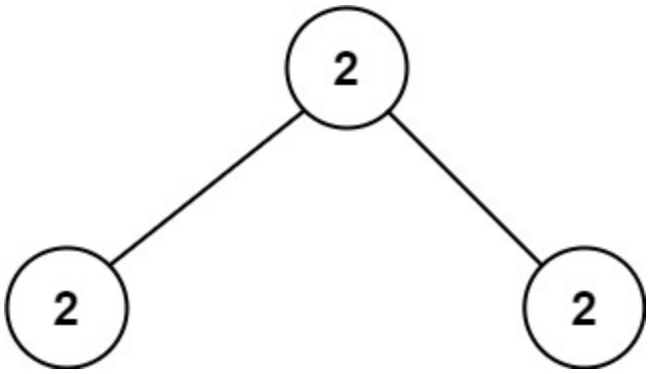
给出这样的一个二叉树，你需要输出所有节点中的第二小的值。如果第二小的值不存在的话，输出 -1 。

示例 1：




```
输入：root = [2,2,5,null,null,5,7]

输出：5

解释：最小的值是 2 ，第二小的值是 5 。
```

示例 2：



输入：`root = [2,2,2]`

输出：`-1`

解**释**：最小的值是 `2`, 但是不存在第二小的值。

提示：

- 树中节点数目在范围 $[1, 25]$ 内
- 1 <= Node.val <= 2^{31} - 1
- 对于树中每个节点 root.val == min(root.left.val, root.right.val)

树的遍历

一个朴素的做法是，直接对树进行遍历（广度 & 深度），使用 `HashSet` 进行存储，得到所有去重后的节点大小。

然后找次小值的方式有多种：可以通过排序找次小值，复杂度为 $O(n \log n)$；也可以使用经典的两个变量 & 一次遍历的方式，找到次小值，复杂度为 $O(n)$。

代码：

```java
class Solution {
    Set<Integer> set = new HashSet<>();
    public int findSecondMinimumValue(TreeNode root) {
        dfs(root);
        if (set.size() < 2) return -1;
        int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
        for (int i : set) {
            if (i <= first) {
                second = first;
                first = i;
            } else if (i <= second) {
                second = i;
            }
        }
        return second;
    }
    void dfs(TreeNode root) {
        if (root == null) return;
        set.add(root.val);
        dfs(root.left);
        dfs(root.right);
    }
}
```

```java
class Solution {
    Set<Integer> set = new HashSet<>();
    public int findSecondMinimumValue(TreeNode root) {
        bfs(root);
        if (set.size() < 2) return -1;
        int first = Integer.MAX_VALUE, second = Integer.MAX_VALUE;
        for (int i : set) {
            if (i <= first) {
                second = first;
                first = i;
            } else if (i <= second) {
                second = i;
            }
        }
        return second;
    }
    void bfs(TreeNode root) {
        Deque<TreeNode> d = new ArrayDeque<>();
        d.addLast(root);
        while (!d.isEmpty()) {
            TreeNode poll = d.pollFirst();
            set.add(poll.val);
            if (poll.left != null) d.addLast(poll.left);
            if (poll.right != null) d.addLast(poll.right);
        }
    }
}
```

- 时间复杂度：树的搜索复杂度为 $O(n)$，通过线性遍历找次小值，复杂度为 $O(n)$。整体复杂度为 $O(n)$
- 空间复杂度：$O(n)$

递归

解法一显然没有利用到本题核心条件：「`root.val = min(root.left.val, root.right.val)`」和「每个子节点数量要么是 `0` 要么是 `2`」。

我们可以设计如下递归函数，含义为 **从 `root` 为根的树进行搜索，找到值比 `cur` 大的最小数**。然后使用全局变量 `ans` 存储答案。

```
void dfs(TreeNode root, int cur)
```

那么最终搜索范围为 `dfs(root, root.val)`，这是因为 **性质 `root.val = min(root.left.val, root.right.val)`，即最小值会不断往上传递，最终根节点必然是全局最小值**。

然后再结合「每个子节点数量要么是 `0` 要么是 `2`」，我们可以特判一下 `ans` 是否为第一次赋值，如果给 `ans` 赋了新值或者更新了更小的 `ans`，则不再需要往下搜索了。

代码：

```
class Solution {
    int ans = -1;
    public int findSecondMinimumValue(TreeNode root) {
        dfs(root, root.val);
        return ans;
    }
    void dfs(TreeNode root, int cur) {
        if (root == null) return ;
        if (root.val != cur) {
            if (ans == -1) ans = root.val;
            else ans = Math.min(ans, root.val);
            return ;
        }
        dfs(root.left, cur);
        dfs(root.right, cur);
    }
}
```

- 时间复杂度：$O(n)$
- 空间复杂度：忽略递归带来的空间开销。复杂度为 $O(1)$

> **🍭 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 **778. 水位上升的泳池中游泳** ，难度为 **困难**。

Tag：「最小生成树」、「并查集」、「Kruskal」、「二分」、「BFS」

在一个 N x N 的坐标方格 grid 中，每一个方格的值 grid[i][j] 表示在位置 (i,j) 的平台高度。

现在开始下雨了。当时间为 t 时，此时雨水导致水池中任意位置的水位为 t 。

你可以从一个平台游向四周相邻的任意一个平台，但是前提是此时水位必须同时淹没这两个平台。

假定你可以瞬间移动无限距离，也就是默认在方格内部游动是不耗时的。

当然，在你游泳的时候你必须待在坐标方格里面。

你从坐标方格的左上平台 (0，0) 出发，最少耗时多久你才能到达坐标方格的右下平台 (N-1, N-1)？

示例 1:

```
输入: [[0,2],[1,3]]

输出: 3

解释:
时间为0时，你位于坐标方格的位置为 (0, 0)。
此时你不能游向任意方向，因为四个相邻方向平台的高度都大于当前时间为 0 时的水位。
等时间到达 3 时，你才可以游向平台 (1, 1). 因为此时的水位是 3，坐标方格中的平台没有比水位 3 更高的，所以你可以游向坐标方格中
```

示例2:

```
输入: [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]
输出: 16

解释:
 0  1  2  3  4
            5
12 13 14 15 16
11
10  9  8  7  6
```

提示:

- 2 <= N <= 50.
- grid[i][j] 是 `[0, ..., N*N - 1]` 的排列。

Kruskal

由于在任意点可以往任意方向移动，所以相邻的点（四个方向）之间存在一条无向边。

边的权重 w 是指两点节点中的最大高度。

按照题意，我们需要找的是从左上角点到右下角点的最优路径，其中最优路径是指**途径的边的最大权重值最小**，然后输入最优路径中的最大权重值。

我们可以先遍历所有的点，将所有的边加入集合，存储的格式为数组 $[a, b, w]$ ，代表编号为 a 的点和编号为 b 的点之间的权重为 w（按照题意，w 为两者的最大高度）。

对集合进行排序，按照 w 进行从小到达排序。

当我们有了所有排好序的候选边集合之后，我们可以对边从前往后处理，每次加入一条边之后，使用并查集来查询左上角的点和右下角的点是否连通。

当我们的合并了某条边之后，判定左上角和右下角的点联通，那么该边的权重即是答案。

这道题和前天的 1631. 最小体力消耗路径 几乎是完全一样的思路。

你甚至可以将那题的代码拷贝过来，改一下对于 w 的定义即可。

代码：

```Java
class Solution {
    int n;
    int[] p;
    void union(int a, int b) {
        p[find(a)] = p[find(b)];
    }
    boolean query(int a, int b) {
        return find(a) == find(b);
    }
    int find(int x) {
        if (p[x] != x) p[x] = find(p[x]);
        return p[x];
    }

    public int swimInWater(int[][] grid) {
        n = grid.length;

        // 初始化并查集
        p = new int[n * n];
        for (int i = 0; i < n * n; i++) p[i] = i;

        // 预处理出所有的边
        // edge 存的是 [a, b, w]：代表从 a 到 b 所需要的时间为 w
        // 虽然我们可以往四个方向移动，但是只要对于每个点都添加「向右」和「向下」两条边的话，其实就已经覆盖
        List<int[]> edges =  new ArrayList<>();
        for (int i = 0; i < n ;i++) {
            for (int j = 0; j < n; j++) {
                int idx = getIndex(i, j);
                p[idx] = idx;
                if (i + 1 < n) {
                    int a = idx, b = getIndex(i + 1, j);
                    int w = Math.max(grid[i][j], grid[i + 1][j]);
                    edges.add(new int[]{a, b, w});
                }
                if (j + 1 < n) {
                    int a = idx, b = getIndex(i, j + 1);
                    int w = Math.max(grid[i][j], grid[i][j + 1]);
                    edges.add(new int[]{a, b, w});
                }
            }
        }

        // 根据权值 w 升序
        Collections.sort(edges, (a,b)->a[2]-b[2]);

        // 从「小边」开始添加，当某一条边刚应用之后，恰好使用得「起点」和「结点」联通
```

```
            // 那么代表找到了「最短路径」中的「权重最大的边」
            int start = getIndex(0, 0), end = getIndex(n - 1, n - 1);
            for (int[] edge : edges) {
                int a = edge[0], b = edge[1], w = edge[2];
                union(a, b);
                if (query(start, end)) {
                    return w;
                }
            }
            return -1;
        }
        int getIndex(int i, int j) {
            return i * n + j;
        }
    }
```

节点的数量为 $n * n$，无向边的数量严格为 $2 * n * (n - 1)$，数量级上为 n^2。

- 时间复杂度：获取所有的边复杂度为 $O(n^2)$，排序复杂度为 $O(n^2 \log n)$，遍历得到最终解复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$。
- 空间复杂度：使用了并查集数组。复杂度为 $O(n^2)$。

注意：假定 Collections.sort() 使用 Arrays.sort() 中的双轴快排实现。

二分 + BFS/DFS

在与本题类型的 1631. 最小体力消耗路径中，有同学问到是否可以用「二分」。

答案是可以的。

题目给定了 $grid[i][j]$ 的范围是 $[0, n^2 - 1]$，所以答案必然落在此范围。

假设最优解为 min 的话（恰好能到达右下角的时间）。那么小于 min 的时间无法到达右下角，大于 min 的时间能到达右下角。

因此在以最优解 min 为分割点的数轴上具有两段性，可以通过「二分」来找到分割点 min。

> 注意：「二分」的本质是两段性，并非单调性。只要一段满足某个性质，另外一段不满足某个性质，就可以用「二分」。其中 33. 搜索旋转排序数组 是一个很好的说明例子。

接着分析，假设最优解为 min，我们在 $[l, r]$ 范围内进行二分，当前二分到的时间为 mid 时：

1. 能到达右下角：必然有 $min \leqslant mid$，让 $r = mid$
2. 不能到达右下角：必然有 $min > mid$，让 $l = mid + 1$

当确定了「二分」逻辑之后，我们需要考虑如何写 $check$ 函数。

显然 $check$ 应该是一个判断给定 时间/步数 能否从「起点」到「终点」的函数。

我们只需要按照规则走特定步数，边走边检查是否到达终点即可。

实现 $check$ 既可以使用 DFS 也可以使用 BFS。两者思路类似，这里就只以 BFS 为例。

代码：

```Java
class Solution {
    int[][] dirs = new int[][]{{1,0}, {-1,0}, {0,1}, {0,-1}};
    public int swimInWater(int[][] grid) {
        int n = grid.length;
        int l = 0, r = n * n;
        while (l < r) {
            int mid = l + r >> 1;
            if (check(grid, mid)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return r;
    }
    boolean check(int[][] grid, int time) {
        int n = grid.length;
        boolean[][] visited = new boolean[n][n];
        Deque<int[]> queue = new ArrayDeque<>();
        queue.addLast(new int[]{0, 0});
        visited[0][0] = true;
        while (!queue.isEmpty()) {
            int[] pos = queue.pollFirst();
            int x = pos[0], y = pos[1];
            if (x == n - 1 && y == n - 1) return true;

            for (int[] dir : dirs) {
                int newX = x + dir[0], newY = y + dir[1];
                int[] to = new int[]{newX, newY};
                if (inArea(n, newX, newY) && !visited[newX][newY] && canMove(grid, pos, to
                    visited[newX][newY] = true;
                    queue.addLast(to);
                }
            }
        }
        return false;
    }
    boolean inArea(int n, int x, int y) {
        return x >= 0 && x < n && y >= 0 && y < n;
    }
    boolean canMove(int[][] grid, int[] from, int[] to, int time) {
        return time >= Math.max(grid[from[0]][from[1]], grid[to[0]][to[1]]);
    }
}
```

- 时间复杂度：在 $[0, n^2]$ 范围内进行二分，复杂度为 $O(\log n)$；每一次 BFS 最多

有 n^2 个节点入队，复杂度为 $O(n^2)$。整体复杂度为 $O(n^2 \log n)$

- 空间复杂度：使用了 visited 数组。复杂度为 $O(n^2)$

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

题目描述

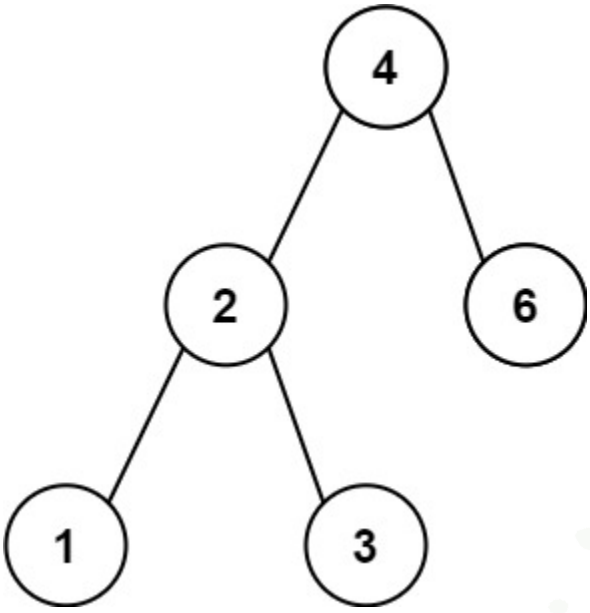
这是 LeetCode 上的 **783. 二叉搜索树节点最小距离** ，难度为 **简单**。

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 。

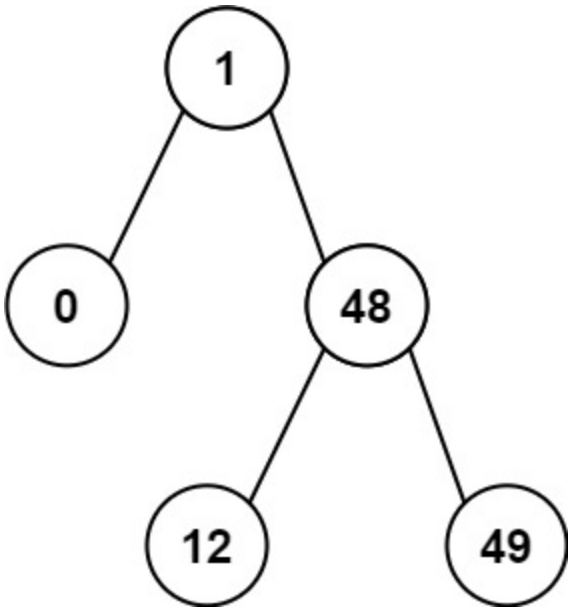
注意：本题与 530：https://leetcode-cn.com/problems/minimum-absolute-difference-in-bst/ 相同

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：$O(n)$

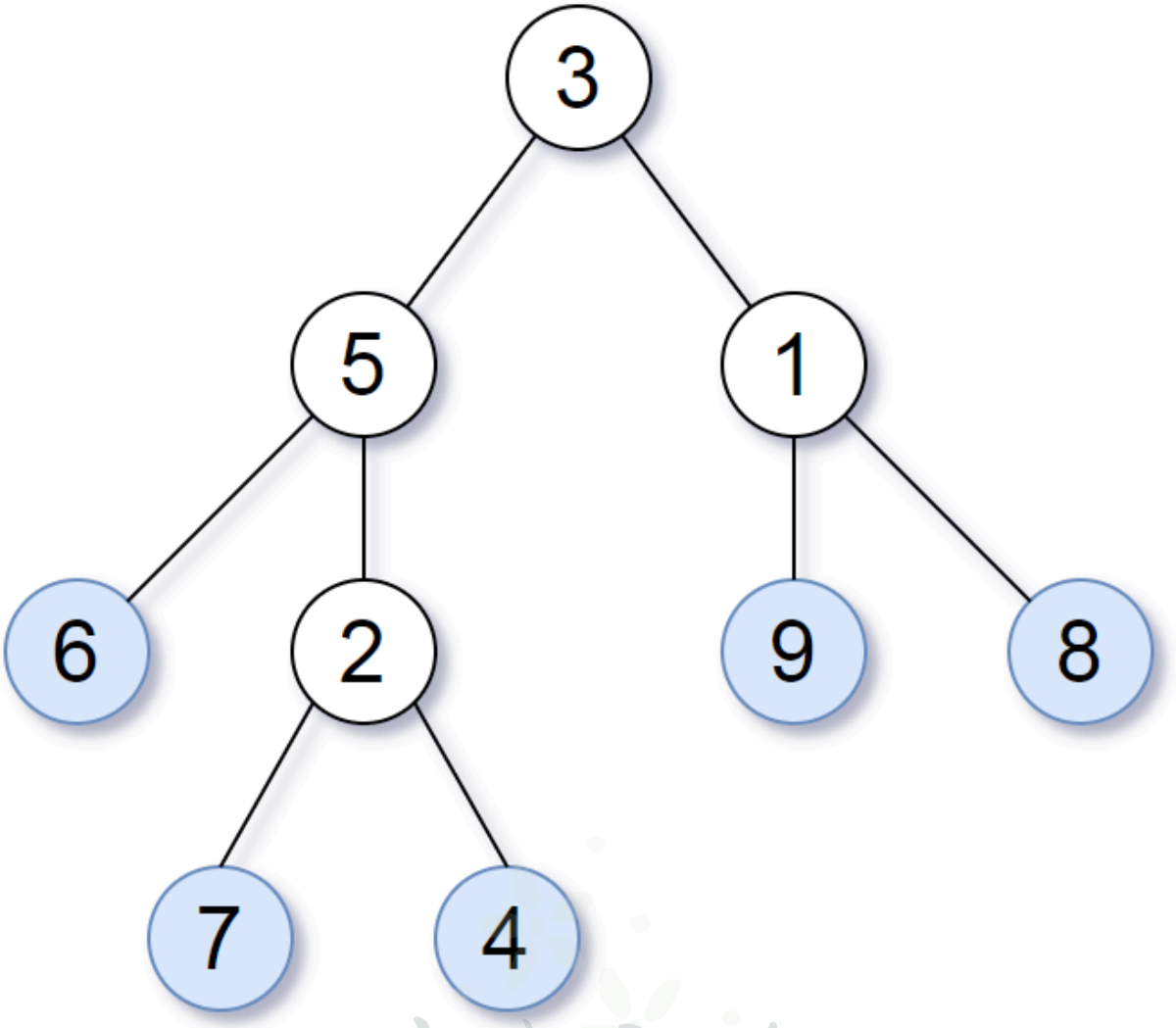
> **🍭 更多精彩内容，欢迎关注：公众号 / Github / LeetCode / 知乎 **

题目描述

这是 LeetCode 上的 **872. 叶子相似的树** ，难度为 **简单**。

Tag：「树的搜索」、「非递归」、「递归」、「DFS」

请考虑一棵二叉树上所有的叶子，这些叶子的值按从左到右的顺序排列形成一个 叶值序列 。

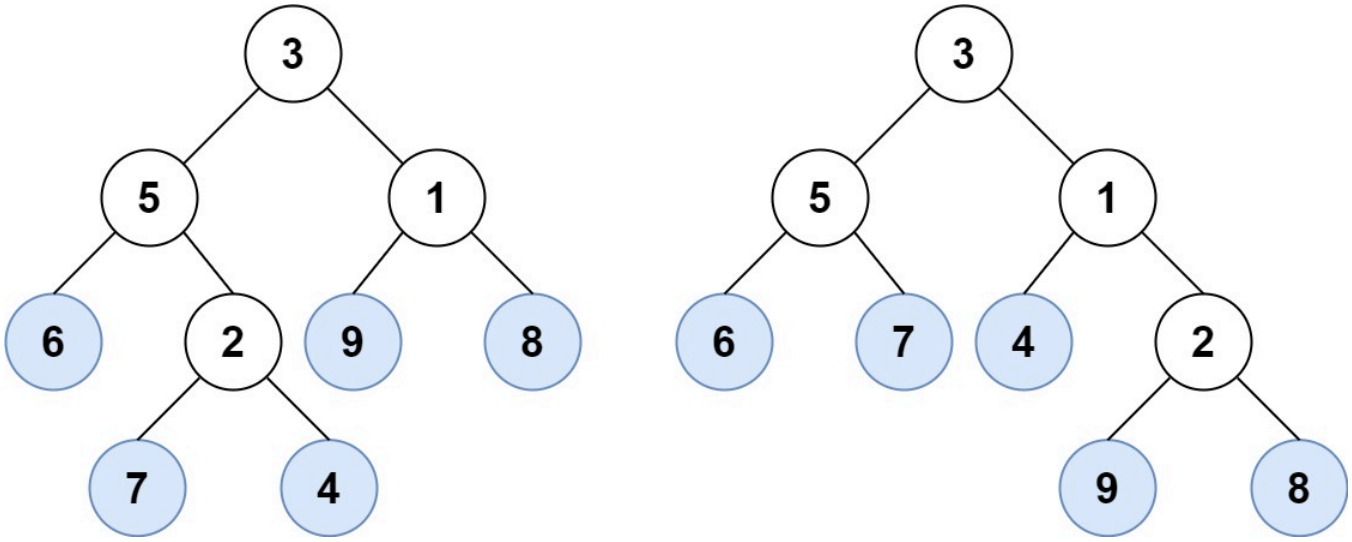


举个例子，如上图所示，给定一棵叶值序列为 (6, 7, 4, 9, 8) 的树。

如果有两棵二叉树的叶值序列是相同，那么我们就认为它们是 叶相似 的。

如果给定的两个根结点分别为 root1 和 root2 的树是叶相似的，则返回 true；否则返回 false 。

示例 1：




```
输入：
root1 = [3,5,1,6,2,9,8,null,null,7,4],
root2 = [3,5,1,6,7,4,2,null,null,null,null,null,null,9,8]

输出：true
```

示例 2：

```
输入：root1 = [1], root2 = [1]

输出：true
```

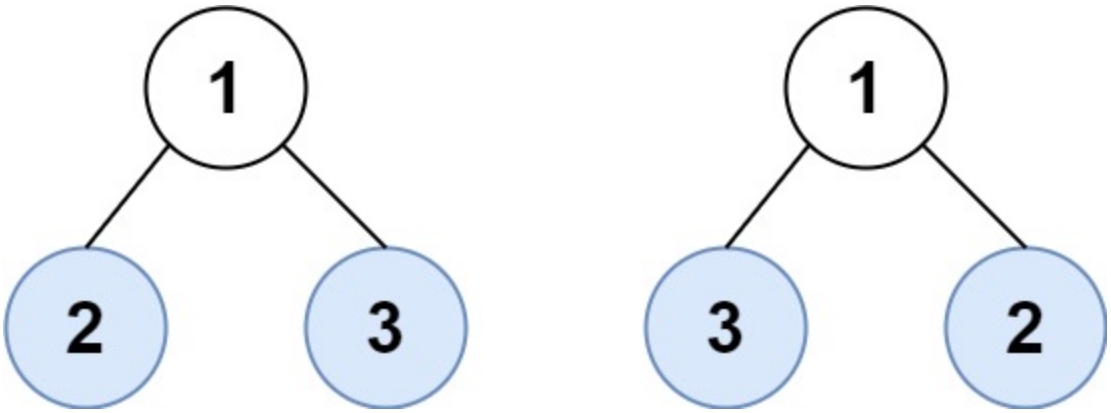
示例 3：

```
输入：root1 = [1], root2 = [2]

输出：false
```

示例 4：

```
输入：root1 = [1,2], root2 = [2,2]

输出：true
```

示例 5：




```
输入：root1 = [1,2,3], root2 = [1,3,2]

输出：false
```

提示：

- 给定的两棵树可能会有 1 到 200 个结点。
- 给定的两棵树上的值介于 0 到 200 之间。

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：

```Java
class Solution {
    public boolean leafSimilar(TreeNode t1, TreeNode t2) {
        List<Integer> l1 = new ArrayList<>(), l2 = new ArrayList<>();
        dfs(t1, l1);
        dfs(t2, l2);
        if (l1.size() == l2.size()) {
            for (int i = 0; i < l1.size(); i++) {
                if (!l1.get(i).equals(l2.get(i))) return false;
            }
            return true;
        }
        return false;
    }
    void dfs(TreeNode root, List<Integer> list) {
        if (root == null) return;
        if (root.left == null && root.right == null) {
            list.add(root.val);
            return;
        }
        dfs(root.left, list);
        dfs(root.right, list);
    }
}
```

- 时间复杂度：`n` 和 `m` 分别代表两棵树的节点数量。复杂度为 $O(n+m)$
- 空间复杂度：`n` 和 `m` 分别代表两棵树的节点数量，当两棵树都只有一层的情况，所有的节点值都会被存储在 $list$ 中。复杂度为 $O(n+m)$

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 boolean leafSimilar(TreeNode t1, TreeNode t2) {
        List<Integer> l1 = new ArrayList<>(), l2 = new ArrayList<>();
        process(t1, l1);
        process(t2, l2);
        if (l1.size() == l2.size()) {
            for (int i = 0; i < l1.size(); i++) {
                if (!l1.get(i).equals(l2.get(i))) return false;
            }
            return true;
        }
        return false;
    }
    void process(TreeNode root, List<Integer> list) {
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.addLast(root);
                root = root.left;
            }
            root = d.pollLast();
            if (root.left == null && root.right == null) list.add(root.val);
            root = root.right;
        }
    }
}
```

- 时间复杂度：`n` 和 `m` 分别代表两棵树的节点数量。复杂度为 $O(n+m)$
- 空间复杂度：`n` 和 `m` 分别代表两棵树的节点数量，当两棵树都只有一层的情况，所有的节点值都会被存储在 $list$ 中。复杂度为 $O(n+m)$

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

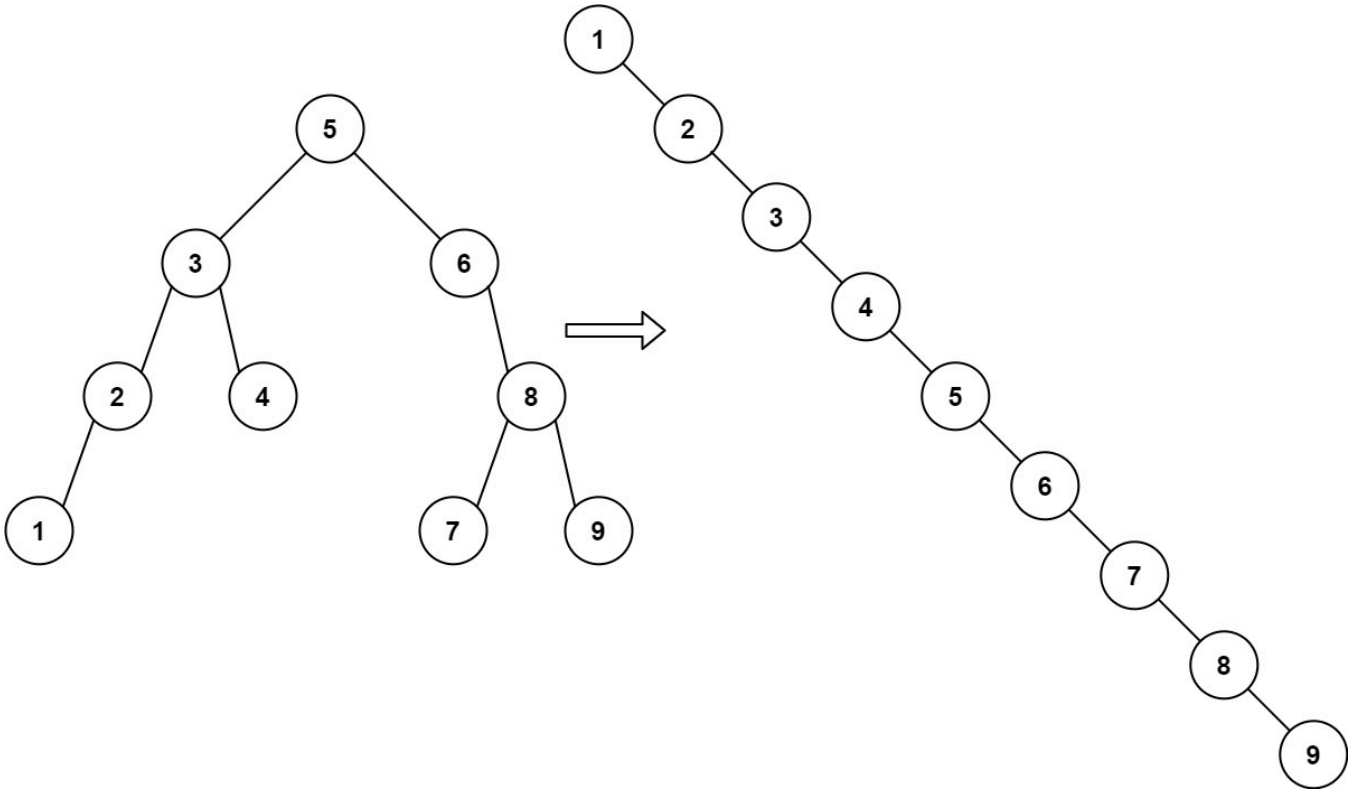
题目描述

这是 LeetCode 上的 **897. 递增顺序搜索树** ，难度为 **简单**。

Tag：「树的遍历」、「递归」、「非递归」

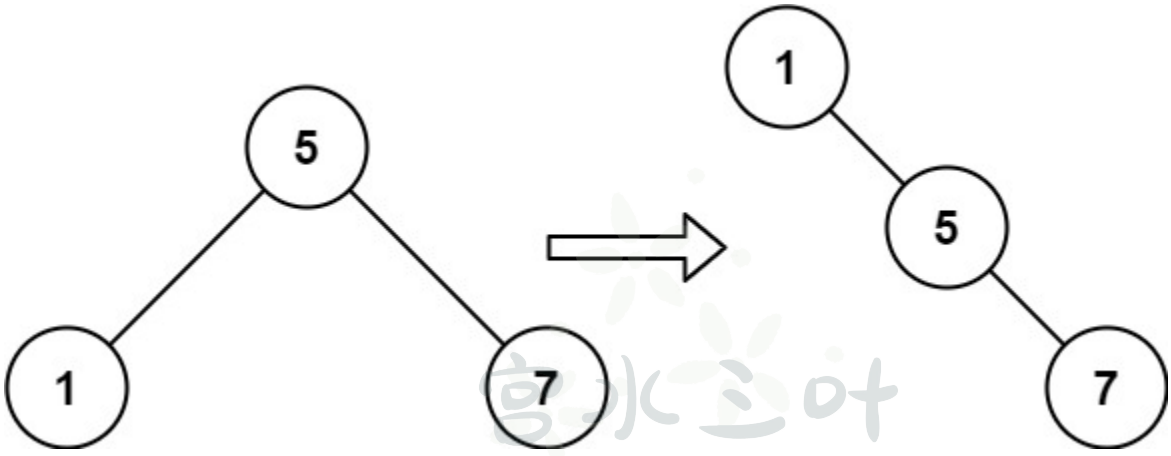
给你一棵二叉搜索树，请你 按中序遍历 将其重新排列为一棵递增顺序搜索树，使树中最左边的节点成为树的根节点，并且每个节点没有左子节点，只有一个右子节点。

示例 1：




```
输入：root = [5,3,6,2,4,null,8,1,null,null,null,7,9]

输出：[1,null,2,null,3,null,4,null,5,null,6,null,7,null,8,null,9]
```

示例 2：




```
输入：root = [5,1,7]

输出：[1,null,5,null,7]
```

提示：

- 树中节点数的取值范围是 [1, 100]
- 0 <= Node.val <= 1000

基本思路

由于给定的树是一棵「二叉搜索树」，因此只要对其进行「中序遍历」即可得到有序列表，再根据有序列表构建答案即可。

而二叉搜索树的「中序遍历」有「迭代」和「递归」两种形式。

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 {
    List<TreeNode> list = new ArrayList<>();
    public TreeNode increasingBST(TreeNode root) {
        dfs(root);
        TreeNode dummy = new TreeNode(-1);
        TreeNode cur = dummy;
        for (TreeNode node : list) {
            cur.right = node;
            node.left = null;
            cur = node;
        }
        return dummy.right;
    }
    void dfs(TreeNode root) {
        if (root == null) return ;
        dfs(root.left);
        list.add(root);
        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：

```
class Solution {
    List<TreeNode> list = new ArrayList<>();
    public TreeNode increasingBST(TreeNode root) {
        Deque<TreeNode> d = new ArrayDeque<>();
        while (root != null || !d.isEmpty()) {
            while (root != null) {
                d.add(root);
                root = root.left;
            }
            root = d.pollLast();
            list.add(root);
            root = root.right;
        }
        TreeNode dummy = new TreeNode(-1);
        TreeNode cur = dummy;
        for (TreeNode node : list) {
            cur.right = node;
            node.left = null;
            cur = node;
        }
        return dummy.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 **

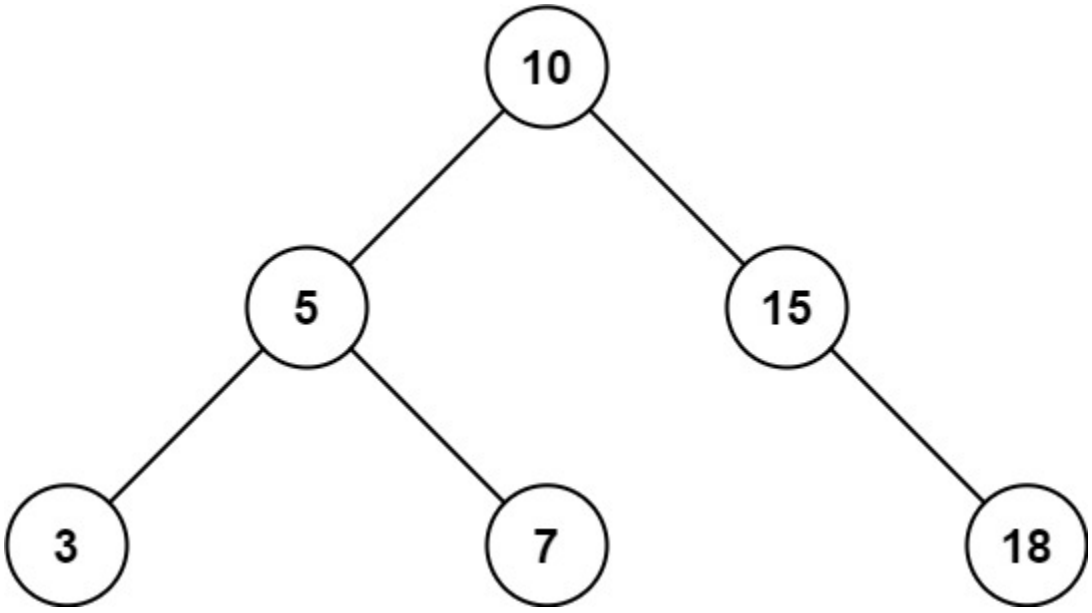
题目描述

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

Tag：「树的搜索」、「DFS」、「BFS」

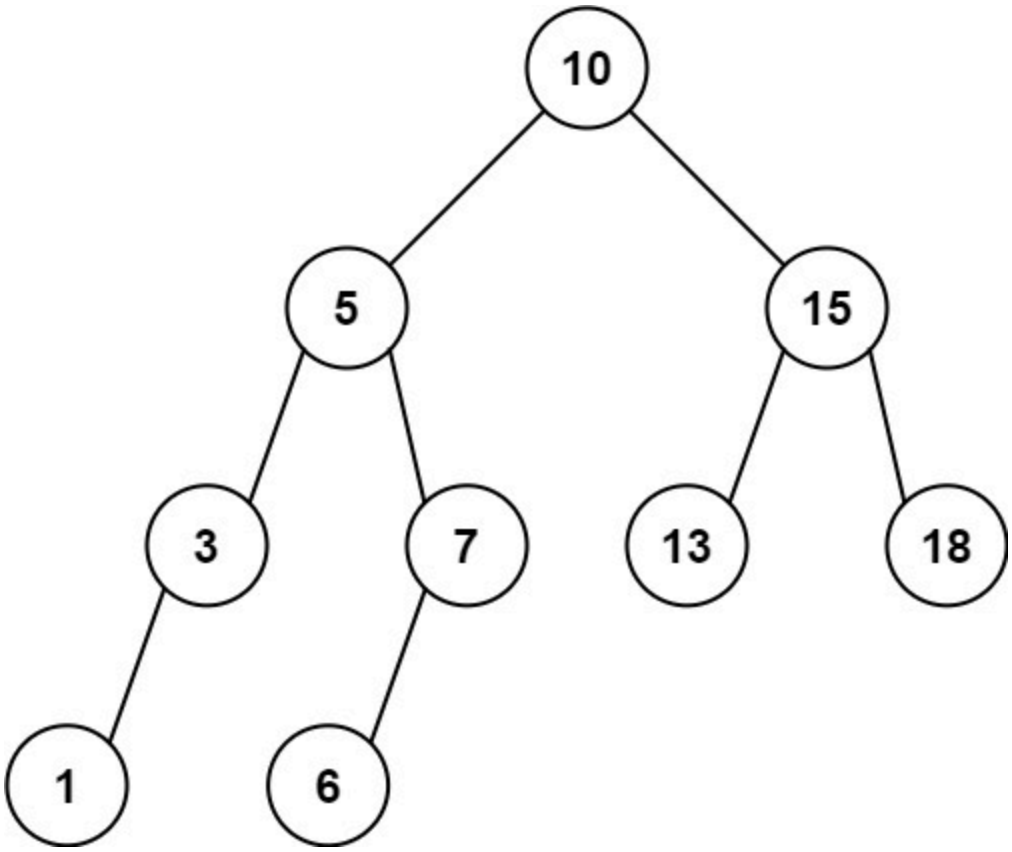
给定二叉搜索树的根结点 root，返回值位于范围 [low, high] 之间的所有结点的值的和。

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 **993. 二叉树的堂兄弟节点** ，难度为 **简单**。

Tag : 「树的搜索」、「BFS」、「DFS」

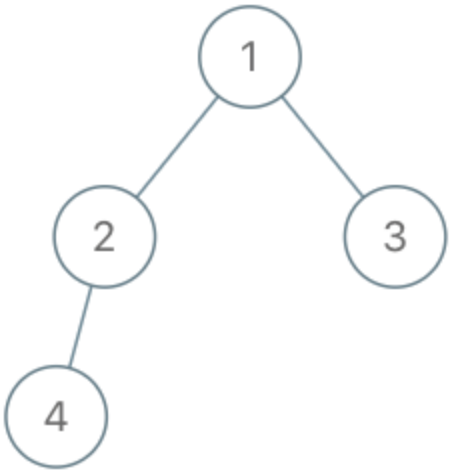
在二叉树中，根节点位于深度 0 处，每个深度为 k 的节点的子节点位于深度 k+1 处。

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 。

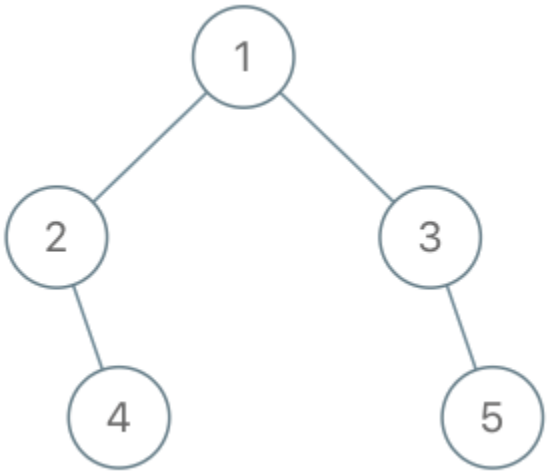
只有与值 x 和 y 对应的节点是堂兄弟节点时，才返回 true 。否则，返回 false。

示例 1：



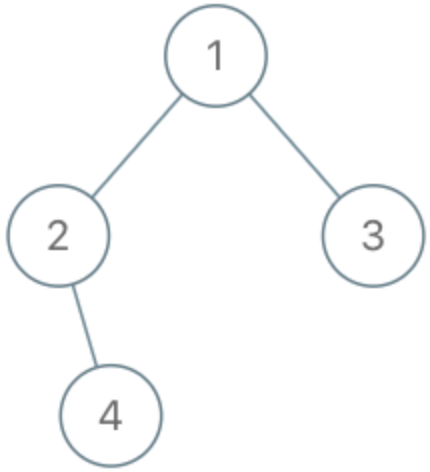

```
输入：root = [1,2,3,4], x = 4, y = 3

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」：

```Java
/**
 * 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：

```java
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};
    }
}
```

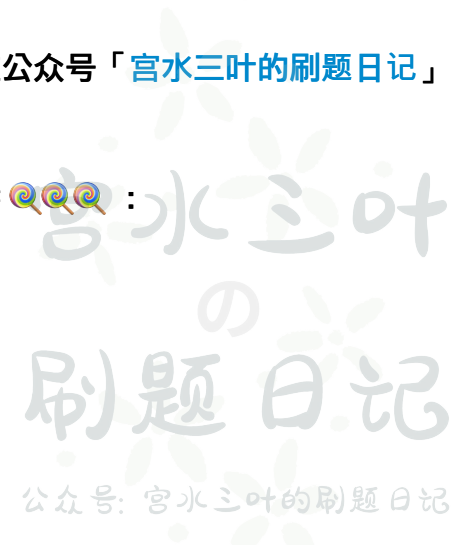
- 时间复杂度：$O(n)$
- 空间复杂度：$O(n)$

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

💡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「树的搜索」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