九章算法班2022版直播第5章

解决99%二叉树问题的算法 —— 分治法 Divide & Conquer

主讲：夏天

二叉树要点

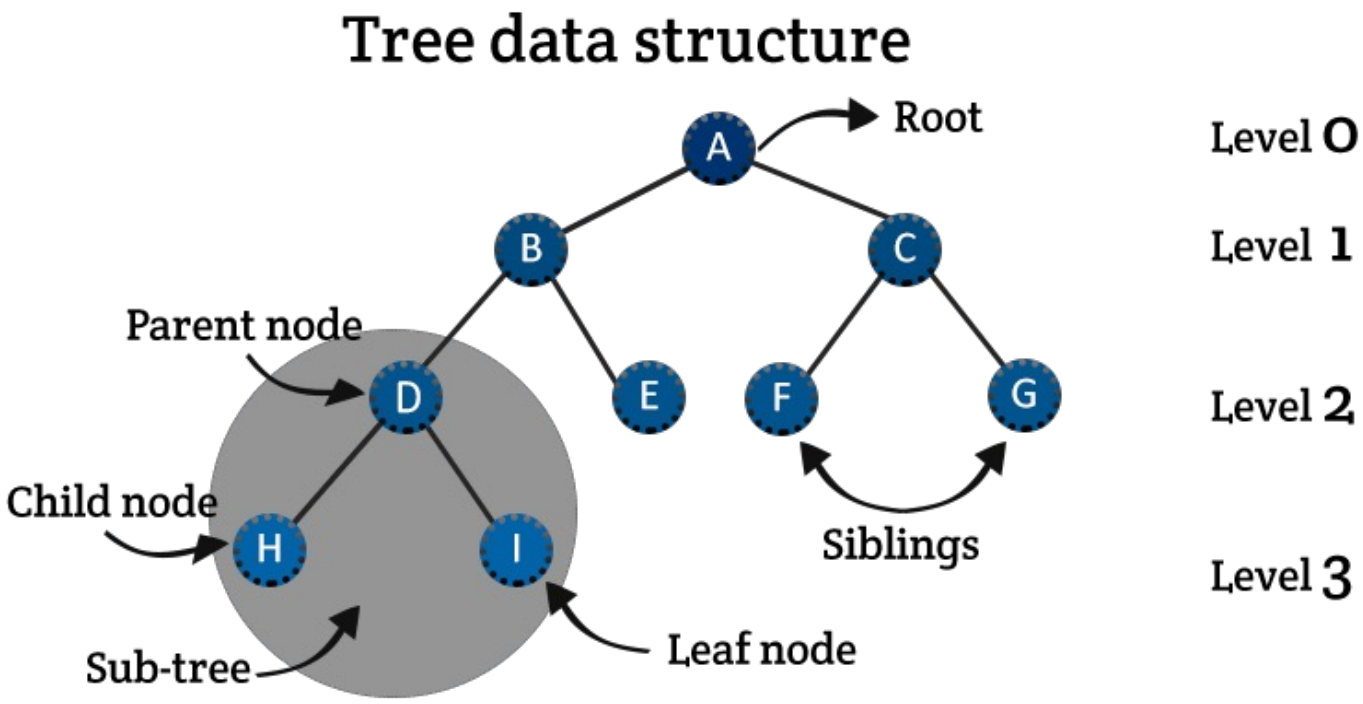
定义

二叉树是每个节点**最多有两个子树**的树结构

二叉树节点定义

```python
class TreeNode:
    def __init__(self, val):
        # 节点值，可以为任意数据类型
        self.val = val
        # 节点的左孩子和右孩子，初始化为None
        self.left, self.right = None, None
```

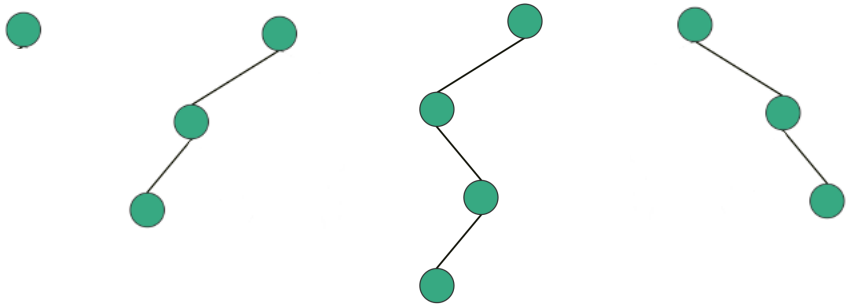
```java
public class TreeNode {
    // 节点值，可以为任意数据类型
    public int val;
    // 节点的左孩子和右孩子，默认初始值为null
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
    }
}
```



二叉树的种类

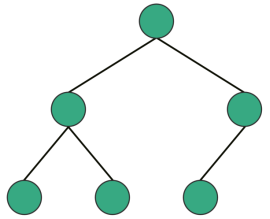
空二叉树

根节点为None/null



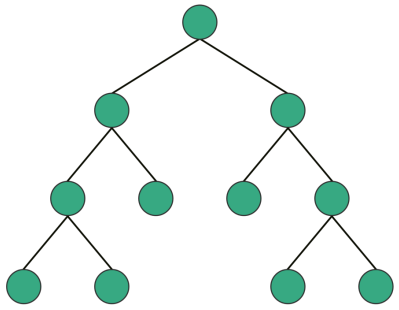
退化二叉树

Degenerate



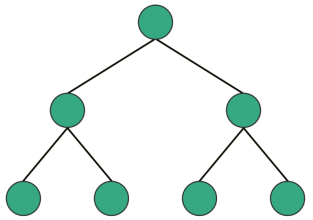
完全二叉树

Complete



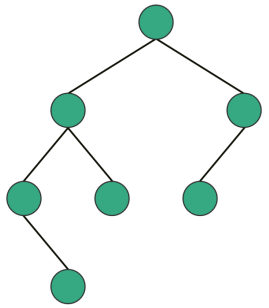
满二叉树

Full



完美二叉树

Perfect



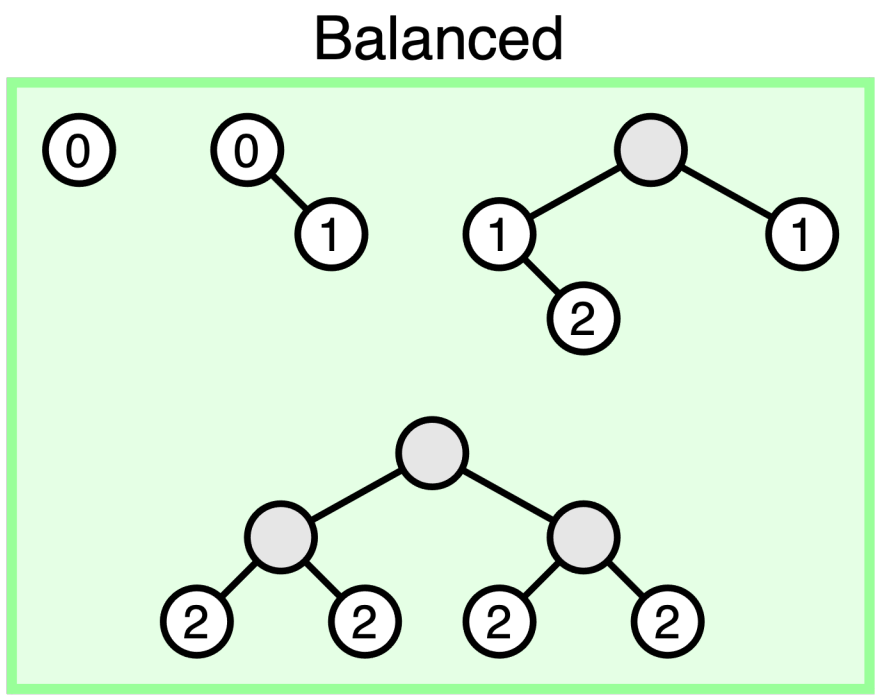
平衡二叉树

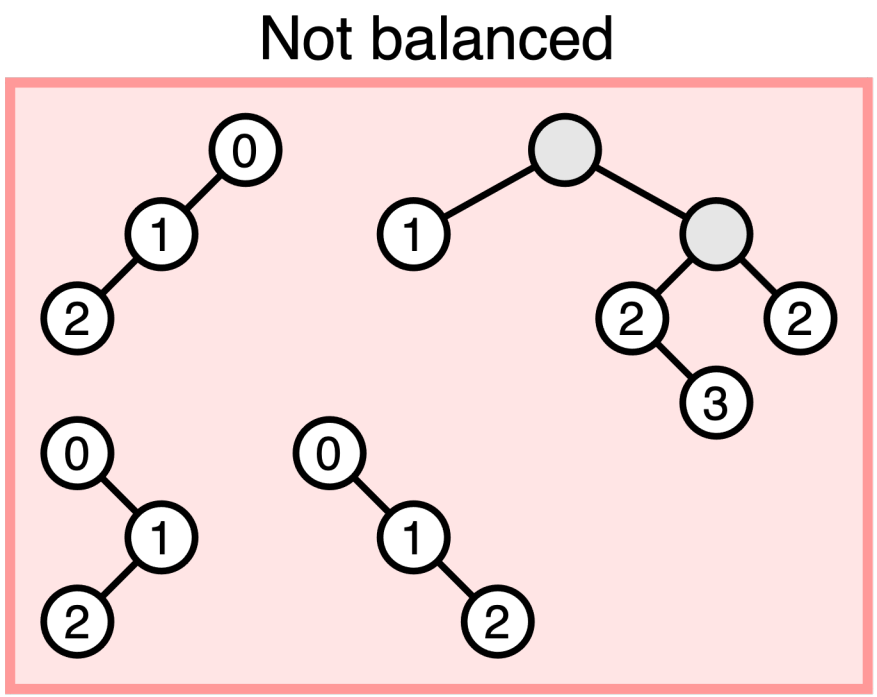
Balanced

平衡二叉树 Balanced Binary Tree

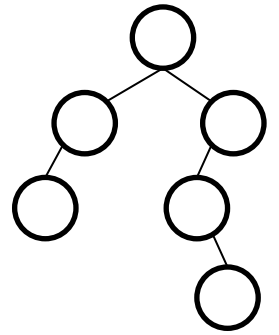
平衡二叉树是不太倾斜的二叉树，尽量保证两边平衡。

空树	平衡二叉树可以是一棵空树
非空树	左右子树的高度之差 <= 1 任意子树也必须是一颗平衡二叉树





这个是平衡二叉树吗？

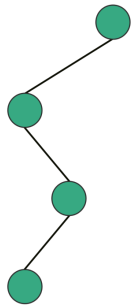


二叉树的高度

最坏 O(N) ，一路向下的二叉树

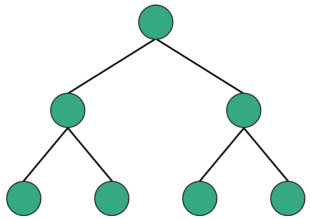
最好 O(logN)，只有 Balanced Binary Tree (平衡二叉树) 才是 O(logN)

一般用 O(h) 来表示更合适



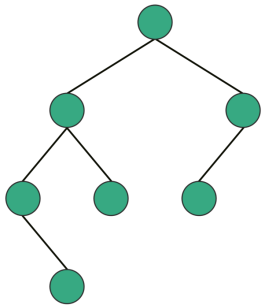
退化二叉树

Degenerate



完美二叉树

Perfect

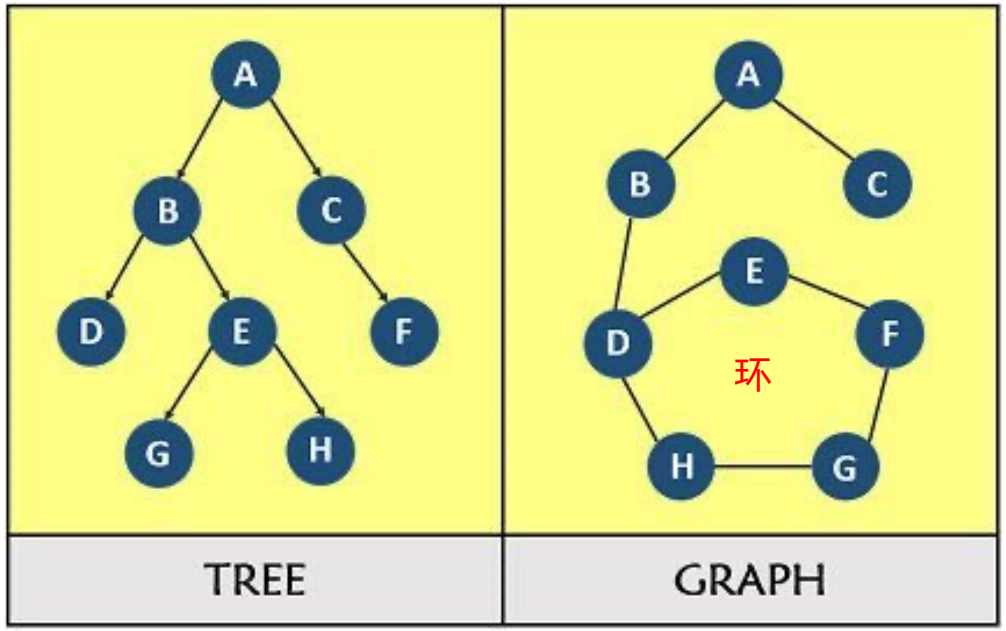


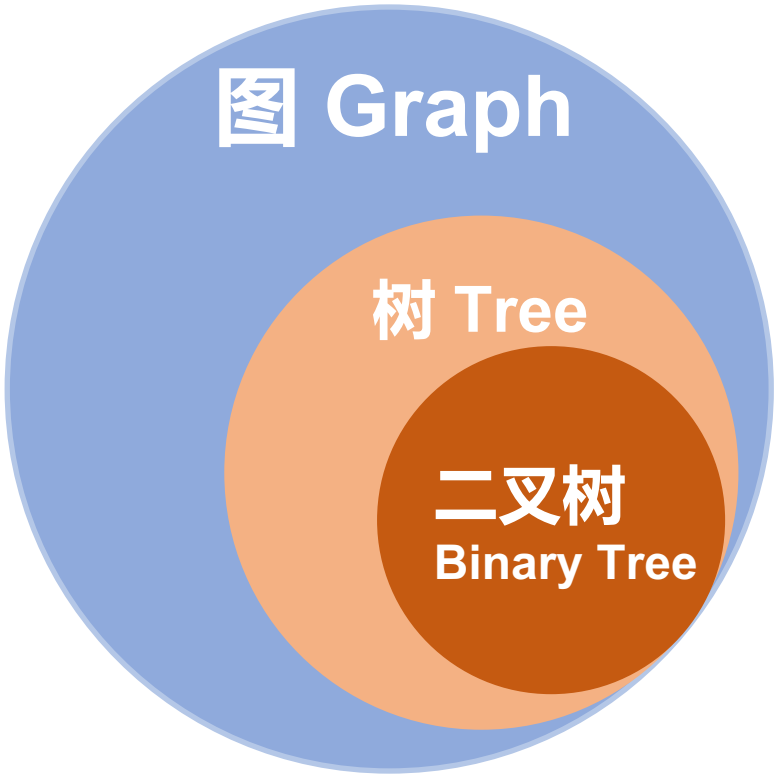
平衡二叉树

Balanced

树 vs 图

树是一种特殊的图，树是**没有环**的图

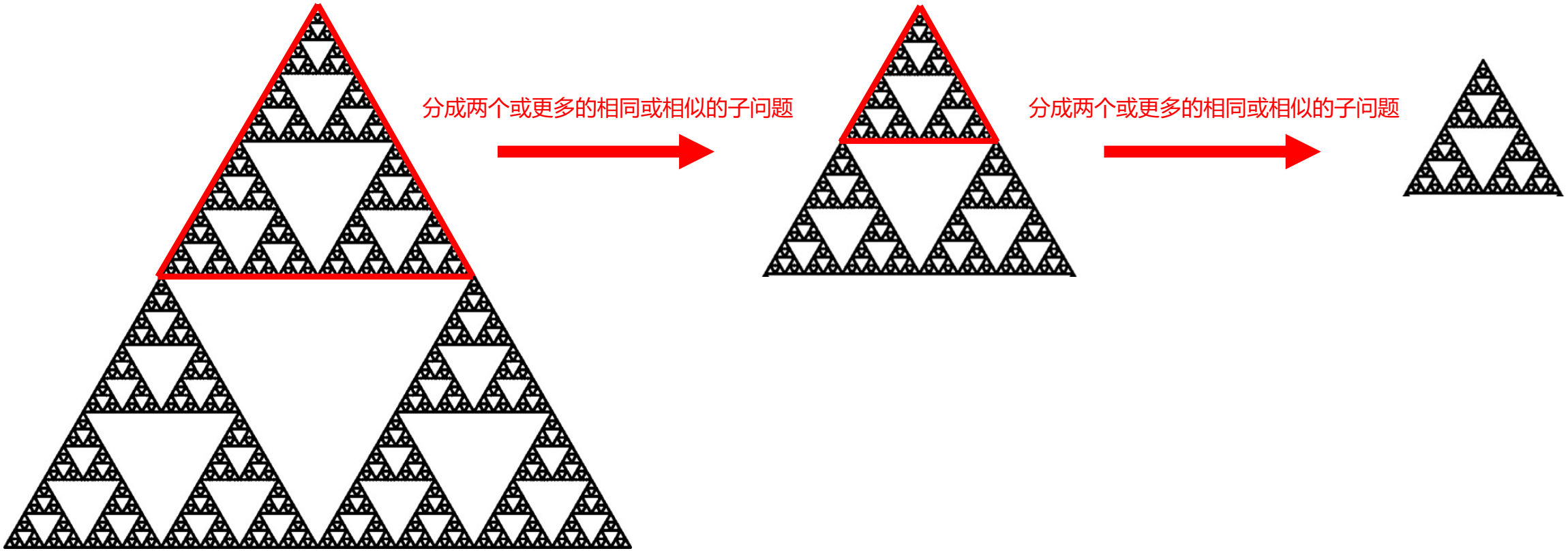




分治法 Divide & Conquer

将大规模问题拆分为若干个**同类型子问题**去处理的算法思想。

把一个复杂的大问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。

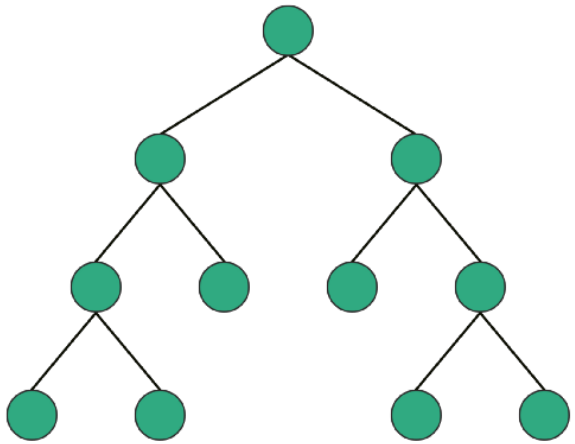


谢尔宾斯基三角 Sierpiński triangle

什么样的数据结构适合分治法？

二叉树：整棵树的左子树和右子树都是二叉树。二叉树的大部分题都可以使用分治法解决

遇到二叉树的问题，就想想整棵树在该问题上的结果和左右子树在该问题上的结果之间有什么联系



数组：一个大数组可以拆分为若干个不相交的子数组

快速排序（Quick Sort），归并排序（Merge Sort），都是基于数组的分治法

二叉树考点剖析

考察形态：二叉树上求值，求路径

代表例题：http://www.lintcode.com/problem/subtree-with-maximum-average/

考点本质：深度优先搜索（Depth First Search）

考察形态：二叉树结构变化

代表例题：http://www.lintcode.com/problem/invert-binary-tree/

考点本质：深度优先搜索（Depth First Search）

考察形态：二叉查找树（Binary Search Tree）

代表例题：http://www.lintcode.com/problem/validate-binary-search-tree/

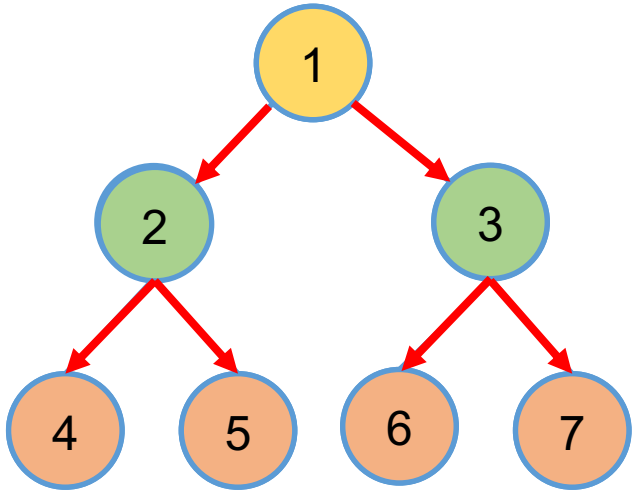
考点本质：深度优先搜索（Depth First Search）

二叉树DFS的三种方式 —— 前序（先根），中序（中根），后序（后根）

不管二叉树的题型如何变化，很多考点都是基于树的深度优先搜索

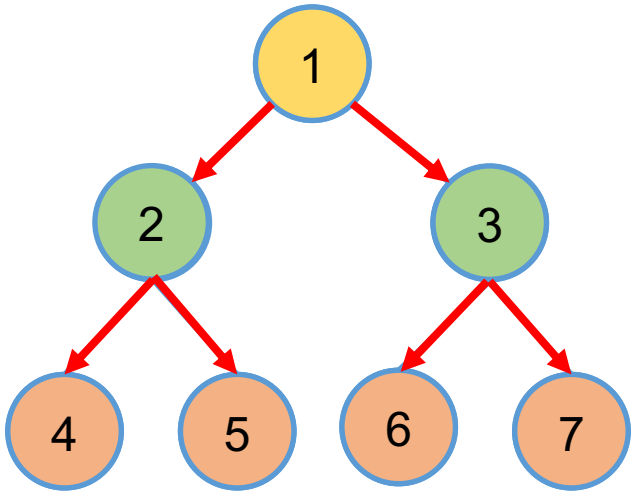
前中后代表根被遍历的位置

前序遍历 Preorder（根 ➔ 左 ➔ 右）



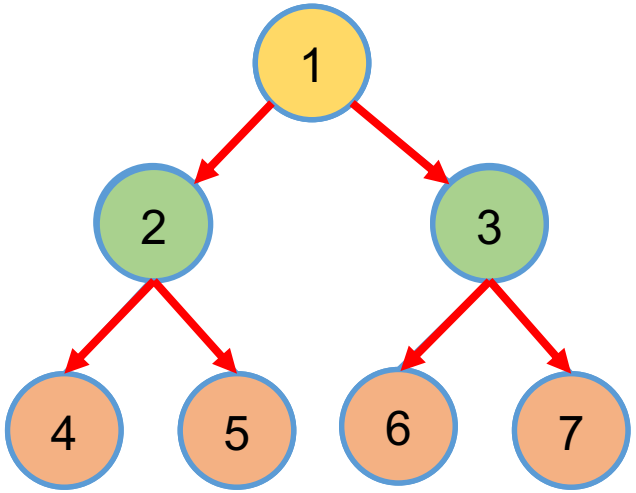
遍历结果 1 2 4 5 3 6 7

中序遍历 Inorder（左➔ 根 ➔ 右）



4 2 5 1 6 3 7

后序遍历 Postorder（左 ➔ 右 ➔ 根）



4 5 2 6 7 3 1

第一类考察形态

考察形态一：二叉树上求值(Max / Min / Average / Sum)，求路径(Paths)

考察形态二：二叉树结构变化

考察形态三：二叉查找树 Binary Search Tree

596 Minimum Subtree 最小子树

Given a binary tree, find the subtree with minimum sum. Return the root of the subtree.

The range of input and output data is in int.

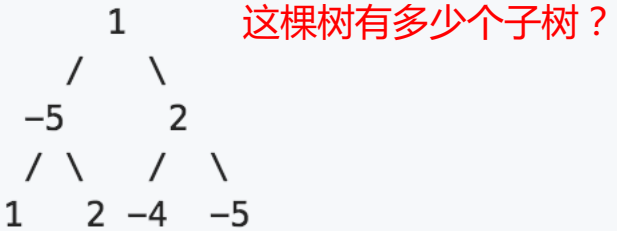
给一棵二叉树, 找到和为最小的子树, **返回其根节点（不是根节点的和）**。

输入输出数据范围都在 int 内。

*保证**只有一棵**和最小的子树*

并且给出的二叉树不是一棵空树

输入:
{1,-5,2,1,2,-4,-5}
输出:1 返回最小子树根节点
说明
这棵树如下所示:



这棵树有多少个子树？

整颗树的和是最小的，所以返回根节点1.

输入:
{1}
输出:1
说明:
这棵树如下所示:

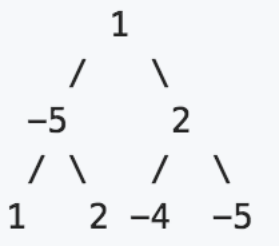
这棵树只有整体这一个子树，所以返回1.

遇到二叉树的问题，就思考整棵树在该问题上的结果和根+左子树+右子树在该问题上的结果之间有什么关系

树的和 = 根节点值 + 左子树和 + 右子树和

代码解析 —— 使用了全局变量的分治法

输入:
{1,-5,2,1,2,-4,-5}
输出:1
说明
这棵树如下所示:



整颗树的和是最小的，所以返回根节点1.

全局变量的坏处

不利于多线程化，对共享变量加锁带来效率下降

```python
    def findSubtree(self, root):
        # 最小和初始值为正无穷
        self.minimum_weight = float('inf')
        # 最小和子树根节点
        self.minimum_subtree_root = None
        self.getTreeSum(root)

        return self.minimum_subtree_root

    # 得到 root 为根的二叉树的所有节点之和
    # 顺便打个擂台求出 minimum subtree
    # 递归三要素之一：递归的定义
    def getTreeSum(self, root):
        # 递归三要素之三：递归的出口
        if root is None:
            return 0

        # 递归三要素之二：递归的拆解
        # 左子树之和
        left_weight = self.getTreeSum(root.left)
        # 右子树之和
        right_weight = self.getTreeSum(root.right)
        # 当前树之和
        root_weight = left_weight + right_weight + root.val

        # 如果当前树之和更小，更新最小和，以及最小和节点
        if root_weight < self.minimum_weight:
            self.minimum_weight = root_weight
            self.minimum_subtree_root = root

        # 返回当前和
        return root_weight
```

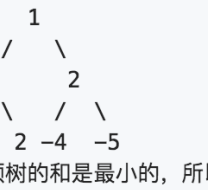
全局变量
instance
variables

```java
public class Solution {
    // 最小和初始值为正无穷
    private int minSum;
    // 最小和子树根节点
    private TreeNode minRoot;

    // 得到 root 为根的二叉树的所有节点之和
    // 顺便打个擂台求出 minimum subtree
    // 递归三要素之一：递归的定义
    public TreeNode findSubtree(TreeNode root) {
        minSum = Integer.MAX_VALUE;
        minRoot = null;
        getTreeSum(root);
        return minRoot;
    }

    private int getTreeSum(TreeNode root) {
        // 递归三要素之三：递归的出口
        if (root == null) {
            return 0;
        }

        // 递归三要素之二：递归的拆解
        // 左子树之和
        int leftSum = getTreeSum(root.left);
        // 右子树之和
        int rightSum = getTreeSum(root.right);
        // 当前树之和
        int rootSum = leftSum + rightSum + root.val;

        // 如果当前树之和更小，更新最小和，以及最小和节点
        if (rootSum < minSum) {
            minSum = rootSum;
            minRoot = root;
        }

        // 返回当前和
        return rootSum;
    }
}
```

全局变量
member variables

代码解析 —— 无全局变量的分治法

输入：
{1,-5,2,1,2,-4,-5}
输出:1
说明
这棵树如下所示：



整颗树的和是最小的，所以返回根节点1.

有全局变量

```python
    def findSubtree(self, root):
        # 最小和初始值为正无穷
        self.minimum_weight = float('inf')
        # 最小和子树根节点
        self.minimum_subtree_root = None
        self.getTreeSum(root)

        return self.minimum_subtree_root

    # 得到 root 为根的二叉树的所有节点之和
    # 顺便打个擂台求出 minimum subtree
    # 递归三要素之一：递归的定义
    def getTreeSum(self, root):
        # 递归三要素之三：递归的出口
        if root is None:
            return 0

        # 递归三要素之二：递归的拆解
        # 左子树之和
        left_weight = self.getTreeSum(root.left)
        # 右子树之和
        right_weight = self.getTreeSum(root.right)
        # 当前树之和
        root_weight = left_weight + right_weight + root.val

        # 如果当前树之和更小，更新最小和，以及最小和节点
        if root_weight < self.minimum_weight:
            self.minimum_weight = root_weight
            self.minimum_subtree_root = root

        # 返回当前和
        return root_weight
```

全局变量 instance variables

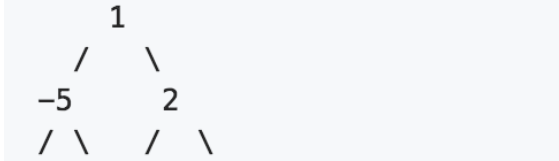
无全局变量

```python
class Solution:

    def findSubtree(self, root):
        minimum, subtree, sum_of_root = self.get_min_tree_sum(root)
        return subtree

    # 返回 最小和 最小子树根 当前树之和
    def get_min_tree_sum(self, root):
        if root is None:
            return sys.maxsize, None, 0

        # 左子树之和
        left_minimum, left_subtree, left_sum = self.get_min_tree_sum(root.left)
        # 右子树之和
        right_minimum, right_subtree, right_sum = self.get_min_tree_sum(root.right)

        # 当前树之和
        sum_of_root = left_sum + right_sum + root.val

        # 如果左子树最小，返回左子树
        if left_minimum == min(left_minimum, right_minimum, sum_of_root):
            return left_minimum, left_subtree, sum_of_root

        # 如果右子树最小，返回右子树
        if right_minimum == min(left_minimum, right_minimum, sum_of_root):
            return right_minimum, right_subtree, sum_of_root

        # 如果当前树最小，返回当前树
        return sum_of_root, root, sum_of_root
```

Python可以一次性返回多个结果

代码解析 —— 无全局变量的分治法

输入:
{1,-5,2,1,2,-4,-5}
输出:1
说明
这棵树如下所示:



整颗树的和是最小的，所以返回根节点1.

```java
public class Solution {
    // 最小和初始值为正无穷
    private int minSum;
    // 最小和子树根节点
    private TreeNode minRoot;

    // 得到 root 为根的二叉树的所有节点之和
    // 顺便打个擂台求出 minimum subtree
    // 递归三要素之一：递归的定义
    public TreeNode findSubtree(TreeNode root) {
        minSum = Integer.MAX_VALUE;
        minRoot = null;
        getTreeSum(root);
        return minRoot;
    }

    private int getTreeSum(TreeNode root) {
        // 递归三要素之三：递归的出口
        if (root == null) {
            return 0;
        }

        // 递归三要素之二：递归的拆解
        // 左子树之和
        int leftSum = getTreeSum(root.left);
        // 右子树之和
        int rightSum = getTreeSum(root.right);
        // 当前树之和
        int rootSum = leftSum + rightSum + root.val;

        // 如果当前树之和更小，更新最小和，以及最小和节点
        if (rootSum < minSum) {
            minSum = rootSum;
            minRoot = root;
        }

        // 返回当前和
        return rootSum;
    }
}
```

全局变量
member variables

```java
// 结果包含三个值，封装在class里
class ResultType {
    // 最小子树根
    public TreeNode minSubtree;
    // 当前树之和，最小和
    public int sum, minSum;
    public ResultType(TreeNode minSubtree, int minSum, int sum) {
        this.minSubtree = minSubtree;
        this.minSum = minSum;
        this.sum = sum;
    }
}
```

把返回的多个结果封装到一个类里

```java
public class Solution {

    public TreeNode findSubtree(TreeNode root) {
        ResultType result = helper(root);
        return result.minSubtree;
    }

    public ResultType helper(TreeNode node) {
        if (node == null) {
            return new ResultType(null, Integer.MAX_VALUE, 0);
        }

        ResultType leftResult = helper(node.left);
        ResultType rightResult = helper(node.right);

        TreeNode minSubTree = node;
        int minSum = leftResult.sum + rightResult.sum + node.val;
        int sum = minSum;

        // 用leftResult更新minSubTree和minSum
        if (leftResult.minSum <= minSum) {
            minSubTree = leftResult.minSubtree;
            minSum = leftResult.minSum;
        }

        // 用rightResult更新minSubTree和minSum
        if (rightResult.minSum <= minSum) {
            minSubTree = rightResult.minSubtree;
            minSum = rightResult.minSum;
        }

        return new ResultType(minSubTree, minSum, sum);
    }
}
```

474 Lowest Common Ancestor II 最近公共祖先 II

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes.

The nearest common ancestor of two nodes refers to the nearest common node among all the parent nodes of two nodes (including the two nodes).

In addition to the left and right son pointers, each node also contains a father pointer, parent, pointing to its own father.

给一棵二叉树和二叉树中的两个节点，找到这两个节点的最近公共祖先LCA。

两个节点的最近公共祖先，是指两个节点的所有父亲节点中（包括这两个节点），离这两个节点最近的公共的节点。

每个节点除了左右儿子指针以外，还包含**一个父亲指针parent，指向自己的父亲**。

注意：

- **这里输入的两个点是node objects，不是数字**
- **自己可以是自己的祖先**

```
输入: {4,3,7,#,#,5,6},3,5
输出: 4
解释:
     4
    / \
   3   7
      / \
     5   6
LCA(3, 5) = 4
```

```
输入: {4,3,7,#,#,5,6},5,6
输出: 7
解释:
     4
    / \
   3   7
      / \
     5   6
LCA(5, 6) = 7
```

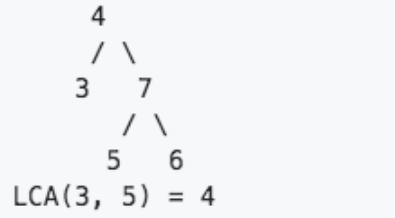
```python
class ParentTreeNode:
    def __init__(self, val):
        self.val = val
        self.parent, self.left, self.right = None, None, None
```

```java
class ParentTreeNode {
    public int val;
    public ParentTreeNode parent, left, right;
}
```

指向父亲的指针很方便，我们**不需要搜索**，只需要**反向顺藤摸瓜**，就可以找到**从根节点到某子节点的路径**

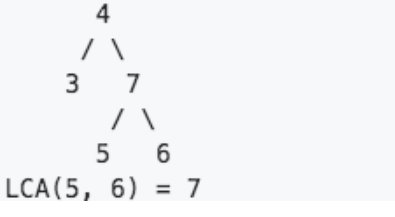
代码解析

输入：{4,3,7,#,#,5,6},3,5
输出：4
解释：



LCA(3, 5) = 4

输入：{4,3,7,#,#,5,6},5,6
输出：7
解释：



LCA(5, 6) = 7

```python
"""
Definition of ParentTreeNode:
class ParentTreeNode:
    def __init__(self, val):
        self.val = val
        self.parent, self.left, self.right = None, None, None
"""

class Solution:
    def lowestCommonAncestorII(self, root, A, B):
        parent_set = set()
        # 把A的祖先节点加入到哈希表中
        curr = A
        while curr is not None:
            parent_set.add(curr)
            curr = curr.parent

        # 遍历B的祖先节点，第一个在哈希表中出现的即为答案
        curr = B
        while curr is not None:
            if curr in parent_set:
                return curr
            curr = curr.parent
        return None

```

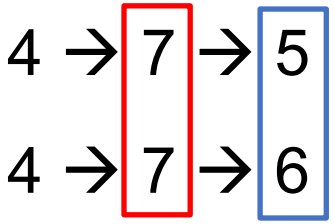
```java
/**
 * Definition of ParentTreeNode:
 *
 * class ParentTreeNode {
 *     public ParentTreeNode parent, left, right;
 * }
 */

public class Solution {
    public ParentTreeNode lowestCommonAncestorII(ParentTreeNode root,
                                                 ParentTreeNode A, ParentTreeNode B) {
        Set<ParentTreeNode> parentSet = new HashSet<>();

        // 把A的祖先节点都加入到哈希表中
        ParentTreeNode curr = A;
        while (curr != null) {
            parentSet.add(curr);
            curr = curr.parent;
        }
        // 遍历B的祖先节点，第一个在哈希表中出现的即为答案
        curr = B;
        while (curr != null) {
            if (parentSet.contains(curr)) {
                return curr;
            }
            curr = curr.parent;
        }
        return null;
    }
}
```

思路分析

还有没有什么其他的方法？ 答：找到两个点的路径，从根节点逐个比较，**最后一个相同点为LCA**

```
输入: {4,3,7,#,#,5,6},5,6
输出: 7
解释:
      4
     / \
    3   7
       / \
      5   6
LCA(5, 6) = 7
```

最后一个相同点为LCA

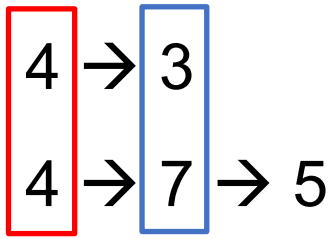


第一次出现不一样

可不可以从后向前比较？

```
输入: {4,3,7,#,#,5,6},3,5
输出: 4
解释:
      4
     / \
    3   7
       / \
      5   6
LCA(3, 5) = 4
```

最后一个相同点为LCA



第一次出现不一样

88 Lowest Common Ancestor of a Binary Tree 最近公共祖先

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes.

The lowest common ancestor is the node with largest depth which is the ancestor of both nodes.

给定二叉树的根节点和两个子节点，找到两个节点的**最近公共父节点**(LCA)。最近公共祖先是两个节点的公共的祖先节点且具有最大深度。

假设给出的两个节点一定都在树中存在 **注意：这里输入的两个点是node objects，不是数字。自己可以是自己的祖先。**

输入：

tree = {4,3,7,#,#,5,6}

A = 3

B = 5

输出：

4

输入：

tree = {1}（**特殊情况**，二叉树只有一个节点）

A = 1

B = 1

输出：

1

遇到二叉树的问题，就思考整棵树在该问题上的结果和根+左子树+右子树在该问题上的结果之间有什么关系

树存在LCA与左右子树存在LCA的关系

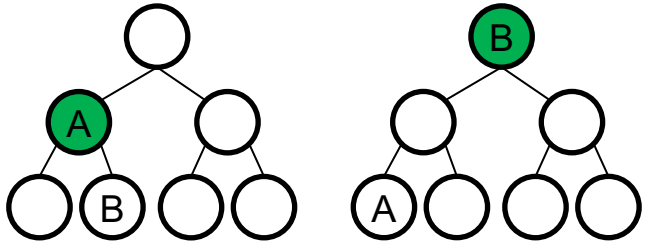
```python
class TreeNode:
    def __init__(self, val):
        self.val = val
        self.left, self.right = None, None
```

```java
public class TreeNode {
    public int val;
    public TreeNode left, right;
    public TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}
```

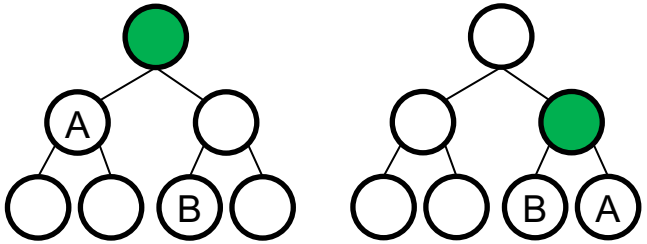
代码解析

假设代码执行到绿色点的位置

情况1： root为A或B



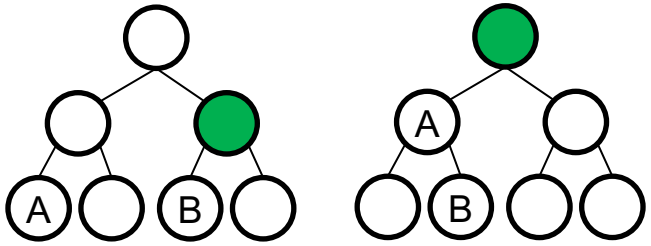
情况2： A，B分别存在于两棵子树，root为LCA



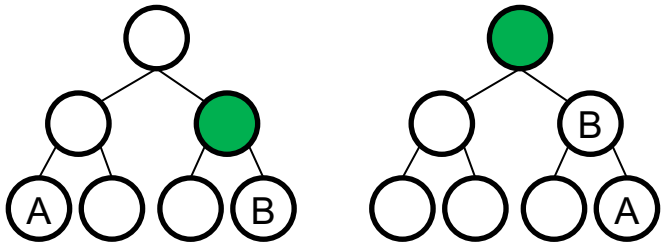

```python
def lowestCommonAncestor(self, root, A, B):
    if root is None:
        return None
    # 如果root为A或B，立即返回，无需继续向下寻找
1   if root == A or root == B:
        return root
    # 分别去左右子树寻找A和B
    left = self.lowestCommonAncestor(root.left, A, B)
    right = self.lowestCommonAncestor(root.right, A, B)

    # 如果A，B分别存在于两棵子树，root为LCA，返回root
2   if left and right:
        return root
    # 左子树有一个点或者左子树有LCA
3   if left:
        return left
    # 右子树有一个点或者右子树有LCA
4   if right:
        return right
    # 左右子树啥都没有
5   return None
```

```java
public TreeNode lowestCommonAncestor(TreeNode root,
                                     TreeNode A, TreeNode B) {
    if(root == null) {
        return null;
    }
    // 如果root为A或B，立即返回，无需继续向下寻找
1   if(root == A || root == B) {
        return root;
    }

    // 分别去左右子树寻找A和B
    TreeNode left = lowestCommonAncestor(root.left, A, B);
    TreeNode right = lowestCommonAncestor(root.right, A, B);

    // 如果A，B分别存在于两棵子树，root为LCA，返回root
2   if(left != null && right != null) {
        return root;
    }
    // 左子树有一个点或者左子树有LCA
3   if(left != null ) {
        return left;
    }
    // 右子树有一个点或者右子树有LCA
4   if(right != null) {
        return right;
    }
    // 左右子树啥都没有
5   return null;
}
```

情况3： 左子树有一个点或者左子树有LCA



情况4： 右子树有一个点或者右子树有LCA



578 Lowest Common Ancestor III 最近公共祖先 III

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes.

The nearest common ancestor of two nodes refers to the nearest common node among all the parent nodes of two nodes (including the two nodes).

Return null if LCA does not exist.

给一棵二叉树和二叉树中的两个节点，找到这两个节点的最近公共祖先LCA。

两个节点的最近公共祖先，是指两个节点的所有父亲节点中（包括这两个节点），离这两个节点最近的公共的节点。

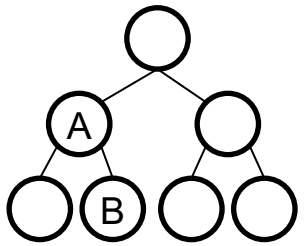
返回 null 如果两个节点在这棵树上不存在最近公共祖先的话。

这两个节点未必都在这棵树上出现。 **注意：这里输入的两个点是node objects，不是数字**

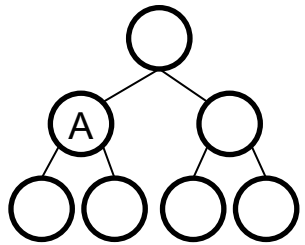
```
输入:
{4, 3, 7, #, #, 5, 6}
3 5
5 6
6 7
5 8
输出:
4
7
7
null
```

```
解释:
  4
 / \
3   7
   / \
  5   6

LCA(3, 5) = 4
LCA(5, 6) = 7
LCA(6, 7) = 7
LCA(5, 8) = null
```

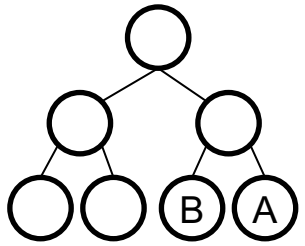
```
输入:
{1}
1 1
输出:
1
说明:
这棵树只有一个值为1的节点
```



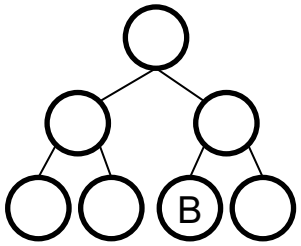
AB都存在



只有A存在



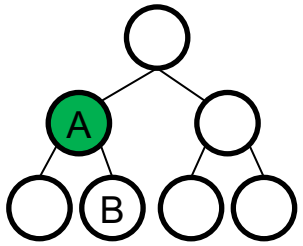
AB都存在

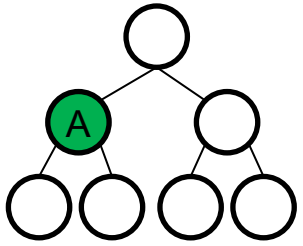


只有B存在

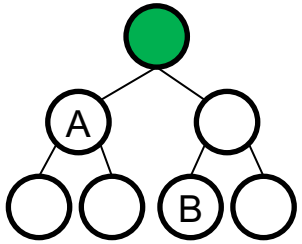
代码解析

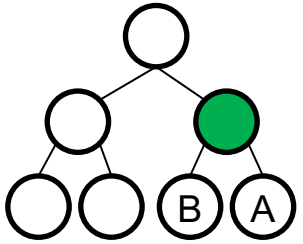
情况1： root为A或B



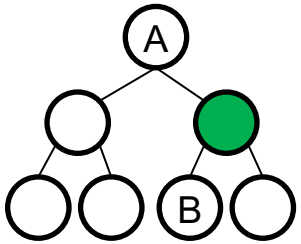


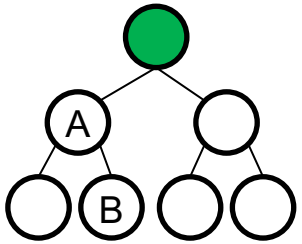
情况2： A，B分别存在于两棵子树，root为LCA





情况3： 左子树有一个点或者左子树有LCA





假设代码执行到绿色点的位置

两个点一定存在

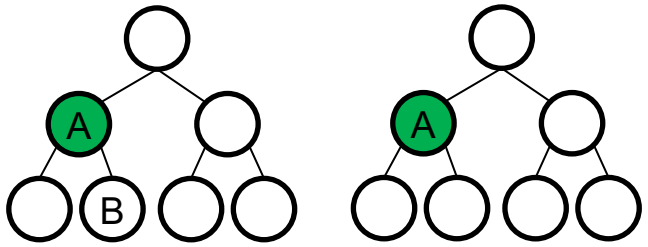
```python
def lowestCommonAncestor(self, root, A, B):
    if root is None:
        return None
    # 如果root为A或B，立即返回，无需继续向下寻找
    if root == A or root == B:
        return root
    # 分别去左右子树寻找A和B
    left = self.lowestCommonAncestor(root.left, A, B)
    right = self.lowestCommonAncestor(root.right, A, B)

    # 如果A，B分别存在于两棵子树，root为LCA，返回root
    if left and right:
        return root
    # 左子树有一个点或者左子树有LCA
    if left:
        return left
    # 右子树有一个点或者右子树有LCA
    if right:
        return right
    # 左右子树啥都没有
    return None
```

两个点可能存在

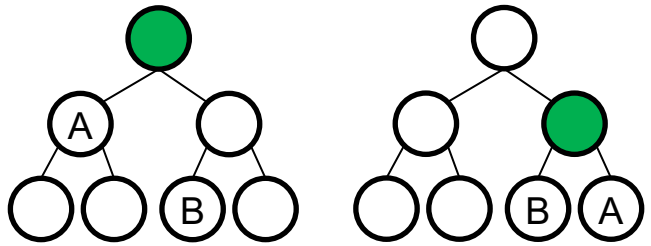
```python
def lowestCommonAncestor3(self, root, A, B):
    a_exist, b_exist, lca = self.helper(root, A, B)
    # 如果AB都存在，才返回
    return lca if a_exist and b_exist else None

def helper(self, root, A, B):
    if root is None:
        return False, False, None

    # 分别去左右子树寻找A和B
    left_a_exist, left_b_exist, left_node = self.helper(root.left, A, B)
    right_a_exist, right_b_exist, right_node = self.helper(root.right, A, B)

    # 如果左边有A，或者右边有A，或者root本身是A，那么root这棵树有A
    a_exist = left_a_exist or right_a_exist or root == A
    # 如果左边有B，或者右边有B，或者root本身是B，那么root这棵树有B
    b_exist = left_b_exist or right_b_exist or root == B

    # 如果root为A或B，返回root(当前root有可能为LCA)
    if root == A or root == B:
        return a_exist, b_exist, root

    # 如果A，B分别存在于两棵子树，root为LCA，返回root
    if left_node is not None and right_node is not None:
        return a_exist, b_exist, root
    # 左子树有一个点或者左子树有LCA
    if left_node is not None:
        return a_exist, b_exist, left_node
    # 右子树有一个点或者右子树有LCA
    if right_node is not None:
        return a_exist, b_exist, right_node
    # 左右子树啥都没有
    return a_exist, b_exist, None
```

代码解析

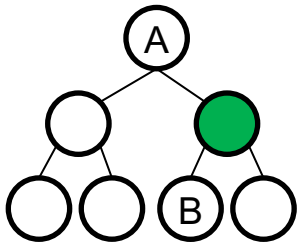
情况1： root为A或B

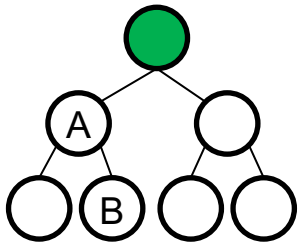


情况2： A，B分别存在于两棵子树，root为LCA



情况3： 左子树有一个点或者左子树有LCA





假设代码执行到绿色点的位置

两个点一定存在

```java
public TreeNode lowestCommonAncestor(TreeNode root,
                                     TreeNode A, TreeNode B) {
    if(root == null) {
        return null;
    }
    // 如果root为A或B，立即返回，无需继续向下寻找
    if(root == A || root == B) {
        return root;
    }

    // 分别去左右子树寻找A和B
    TreeNode left = lowestCommonAncestor(root.left, A, B);
    TreeNode right = lowestCommonAncestor(root.right, A, B);

    // 如果A，B分别存在于两棵子树，root为LCA，返回root
    if(left != null && right != null) {
        return root;
    }
    // 左子树有一个点或者左子树有LCA
    if(left != null ) {
        return left;
    }
    // 右子树有一个点或者右子树有LCA
    if(right != null) {
        return right;
    }
    // 左右子树啥都没有
    return null;
}
```

两个点可能存在

```java
class ResultType {
    public boolean a_exist, b_exist;
    public TreeNode node;
    ResultType(boolean a, boolean b, TreeNode n) {
        a_exist = a;
        b_exist = b;
        node = n;
    }
}

public class Solution {
    public TreeNode lowestCommonAncestor3(TreeNode root, TreeNode A,
                                          TreeNode B) {
        ResultType rt = helper(root, A, B);
        // 如果AB都存在，才返回
        return (rt.a_exist && rt.b_exist) ? rt.node : null;
    }

    public ResultType helper(TreeNode root, TreeNode A, TreeNode B) {

        if (root == null)
            return new ResultType(false, false, null);

        // 分别去左右子树寻找A和B
        ResultType left_rt = helper(root.left, A, B);
        ResultType right_rt = helper(root.right, A, B);

        // 如果左边有A，或者右边有A，或者root本身是A，那么root这棵树有A
        boolean a_exist = left_rt.a_exist || right_rt.a_exist || root == A;
        // 如果左边有B，或者右边有B，或者root本身是B，那么root这棵树有B
        boolean b_exist = left_rt.b_exist || right_rt.b_exist || root == B;

        // 如果root为A或B，返回root(当前root有可能为LCA)
        if (root == A || root == B)
            return new ResultType(a_exist, b_exist, root);
        // 如果A，B分别存在于两棵子树，root为LCA，返回root
        if (left_rt.node != null && right_rt.node != null)
            return new ResultType(a_exist, b_exist, root);
        // 左子树有一个点或者左子树有LCA
        if (left_rt.node != null)
            return new ResultType(a_exist, b_exist, left_rt.node);
        // 右子树有一个点或者右子树有LCA
        if (right_rt.node != null)
            return new ResultType(a_exist, b_exist, right_rt.node);
        // 左右子树啥都没有
        return new ResultType(a_exist, b_exist, null);
    }
}
```

第二类考察形态

考察形态一：二叉树上求值(Maximum / Minimum / Average / Sum)，求路径(Paths)

考察形态二：二叉树结构变化

考察形态三：二叉查找树 Binary Search Tree

453 Flatten Binary Tree to Linked List 将二叉树拆成链表

Flatten a binary tree to a fake "linked list" in pre-order traversal.

Here we use the *right* pointer in TreeNode as the *next* pointer in ListNode.

将一棵二叉树按照前序遍历拆解成为一个假链表。所谓的假链表是说，用二叉树的 *right* 指针，来表示链表中的 *next* 指针。

```
输入: {1,2,5,3,4,#,6}
输出: {1,#,2,#,3,#,4,#,5,#,6}
解释:
     1
    / \
   2   5
  / \   \
 3   4   6

1
\
 2
  \
   3
    \
     4
      \
       5
        \
         6
```

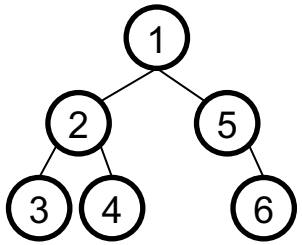
```
输入: {1}
输出: {1}
解释:
     1
     1
```

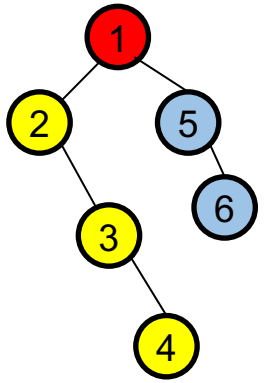
把这道题目翻译成人话：DFS前序遍历这棵树，然后把结果一路向右串联起来

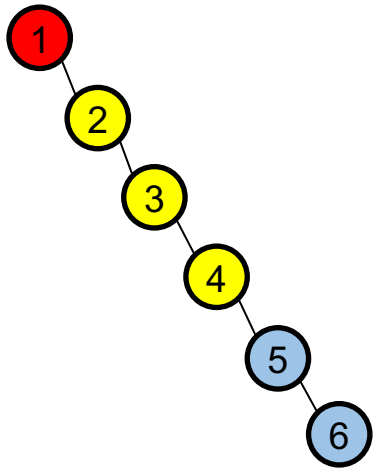
遇到二叉树的问题，就思考整棵树在该问题上的结果和根+左子树+右子树在该问题上的结果之间有什么关系

树的链表 = 树的根节点 + 左子树链表 + 右子树链表

代码解析








```python
    def flatten(self, root):
        self.flatten_and_return_last_node(root)

    # 把root这棵树摊平（形成一路向右的假链表），并返回摊平的树的最尾部节点
    def flatten_and_return_last_node(self, root):
        # 如果root为空，无需摊平，直接返回
        if root is None:
            return None

        # 分别flatten左右子树，并返回flatten之后的最后一个点
        left_last = self.flatten_and_return_last_node(root.left)
        right_last = self.flatten_and_return_last_node(root.right)

        # 如果左子树不为空，需要重组root，摊平的左子树和摊平的右子树
        # 如果左子树为空，不需要重组，root的右子树已经与摊平的右子树相连
        if left_last is not None:
            # 把摊平的左子树的终点和摊平的右子树的起点连接起来
            left_last.right = root.right
            # 把root的右孩子指向摊平的左子树的起点
            root.right = root.left
            # 把root的左孩子置空
            root.left = None

        # root这棵树被摊平后，返回这棵树的最尾部节点
        # 如果rightLast存在，那么rightLast是最尾部节点
        # 否则，如果leftLast存在，那么leftLast是最尾部节点
        # 否则，root是最尾部节点
        return right_last or left_last or root
```

```java
    public void flatten(TreeNode root) {
        flattenAndReturnLastNode(root);
    }

    // 把root这棵树摊平（形成一路向右的假链表），并返回摊平的树的最尾部节点
    private TreeNode flattenAndReturnLastNode(TreeNode root) {
        // 如果root为空，无需摊平，直接返回
        if (root == null) {
            return null;
        }

        // 分别flatten左右子树，并返回flatten之后的最后一个点
        TreeNode leftLast = flattenAndReturnLastNode(root.left);
        TreeNode rightLast = flattenAndReturnLastNode(root.right);

        // 如果左子树不为空，需要重组root，摊平的左子树和摊平的右子树
        // 如果左子树为空，不需要重组，root的右子树已经与摊平的右子树相连
        if (root.left != null) {
            // 把摊平的左子树的终点和摊平的右子树的起点连接起来
            leftLast.right = root.right;
            // 把root的右孩子指向摊平的左子树的起点
            root.right = root.left;
            // 把root的左孩子置空
            root.left = null;
        }

        // root这棵树被摊平后，返回这棵树的最尾部节点
        // 如果rightLast存在，那么rightLast是最尾部节点
        // 否则，如果leftLast存在，那么leftLast是最尾部节点
        // 否则，root是最尾部节点
        if (rightLast != null) {
            return rightLast;
        } else if (leftLast != null) {
            return leftLast;
        }
        return root;
    }
```

课间休息

第三类考察形态

考察形态一：二叉树上求值(Maximum / Minimum / Average / Sum)，求路径(Paths)

考察形态二：二叉树结构变化

考察形态三：二叉查找树 Binary Search Tree

Binary Search Tree 要点

定义

左子树节点的值 < **根**节点的值 < **右子树**节点的值

BST中的**任意一个子树都是BST**

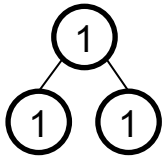
相等的情况

值相等的点**可能在左子树**，或者**可能在右子树**，需要跟面试官**澄清**

中序遍历

中序遍历结果有序（不下降的顺序，有些相邻点可能相等）

- 如果二叉树的中序遍历不是“不下降”序列，则一定不是BST
- 如果二叉树的中序遍历是不下降，也未必是BST，反例：



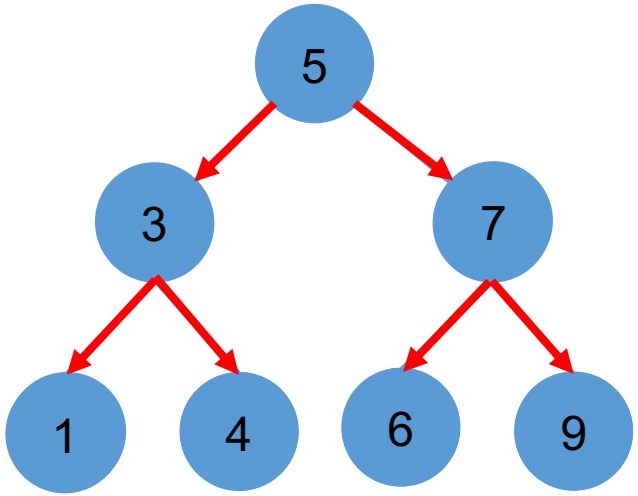
二叉查找树的高度

最坏 O(n)

最好 O(logN) 平衡二叉树

用 O(h) 表示更合适

中序遍历 Inorder（左➔ 根 ➔ 右）



1 3 4 5 6 7 9

BST 基本操作

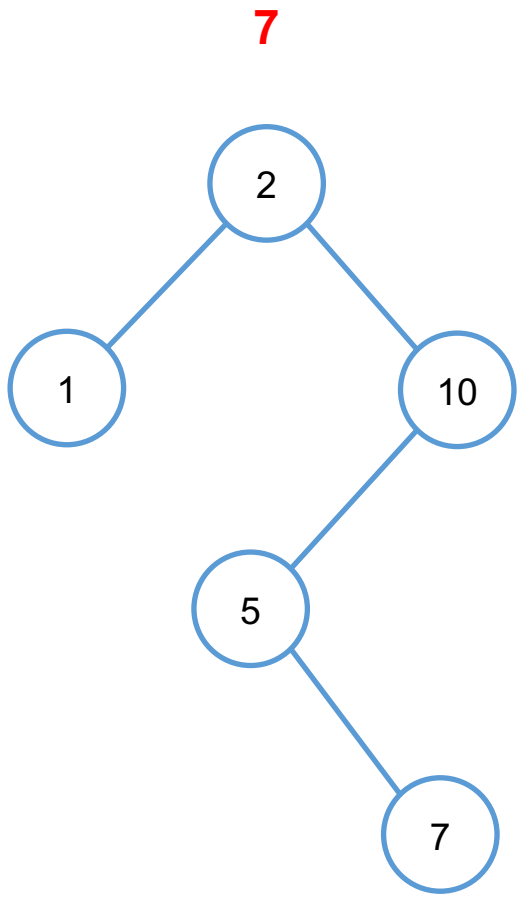
Build - 1359. Convert Sorted Array to Binary Search Tree

Insert - 85. Insert Node in a Binary Search Tree

Search - 1524. Search in a Binary Search Tree

Delete - 701. Trim a Binary Search Tree

Iterate - 86. Binary Search Tree Iterator



红黑树 Red-Black Tree

红黑树是一种 Balanced BST

Java	C++	Python
TreeMap / TreeSet	map / set	标准库没有，第三方库有

应用

O(logN) 的时间内实现增删查改

O(logN) 的时间内实现找最大找最小

O(logN) 的时间内实现找比某个数小的最大值(upperBound)和比某个数大的最小值(lower Bound)

只可能考红黑树的应用，不会考红黑树的实现！

902 Kth Smallest Element in a BST, BST中第K小的元素

Given a binary search tree, write a function kthSmallest to find the kth smallest element in it.

给一棵二叉搜索树，写一个 KthSmallest 函数来找到其中第 K 小的元素。

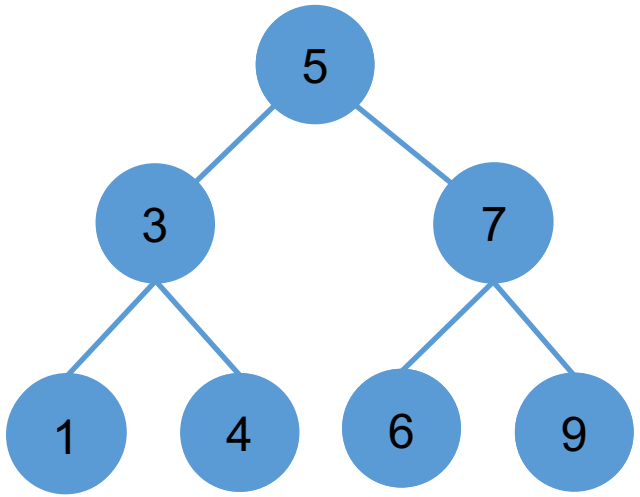
你可以假设 k 总是有效的，1 ≤ k ≤ 树的总节点数。

输入：

BST = {5, 3, 7, 1, 4, 6, 9},

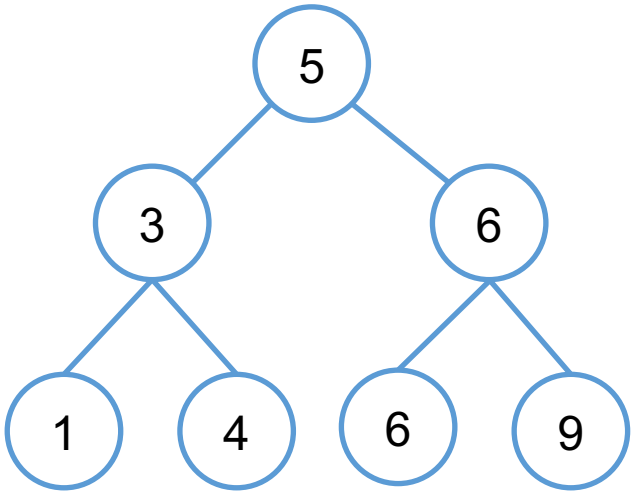
k = 3 (第3小的值)

输出：4



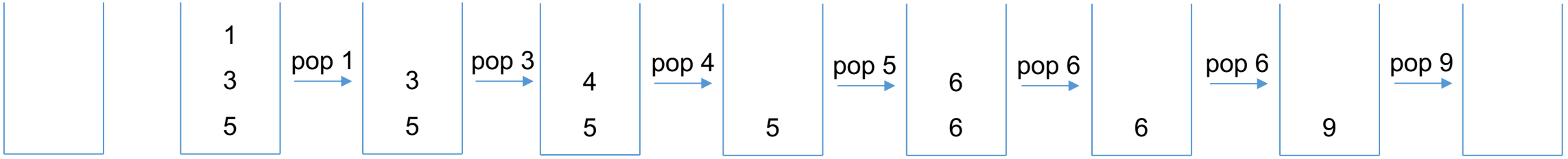
1 3 4 5 6 7 9

代码解析，二叉树的中序遍历的非递归实现




```python
def kthSmallest(self, root, k):
    # 使用stack进行非递归算法的数据存取
    stack = []

    # 一路向左，把树的左边缘的点全部入栈
    while (root):
        stack.append(root)
        root = root.left
    # 0到k-2总共包括k-1次操作
    # 经历k-1次才做，可以把第k个数调整到栈顶
    for i in range(k - 1):
        # 前一个元素出栈
        node = stack.pop()
        # 如果出栈元素有右子树，把右子树的左边缘的点全部入栈
        if node.right:
            node = node.right
            # 一路向左，把树的左边缘的点全部入栈
            while node:
                stack.append(node)
                node = node.left

    # 当前栈顶就是第K个元素
    return stack[-1].val
```

```java
public int kthSmallest(TreeNode root, int k) {
    // 使用stack进行非递归算法的数据存取
    Stack<TreeNode> stack = new Stack<>();

    // 一路向左，把树的左边缘的点全部入栈
    while (root != null) {
        stack.push(root);
        root = root.left;
    }

    // 0到k-2总共包括k-1次操作
    // 经历k-1次才做，可以把第k个数调整到栈顶
    for (int i = 0; i < k - 1; i++) {
        // 前一个元素出栈
        TreeNode node = stack.pop();
        // 如果出栈元素有右子树，把右子树的左边缘的点全部入栈
        if (node.right != null) {
            node = node.right;
            // 一路向左，把树的左边缘的点全部入栈
            while (node != null) {
                stack.push(node);
                node = node.left;
            }
        }
    }
    // 当前栈顶就是第K个元素
    return stack.peek().val;
}
```



900 Closest Binary Search Tree Value 二叉搜索树中最接近的值

Given a non-empty binary search tree and a target value, find the value in the BST that is closest to the target.

给一棵非空二叉搜索树以及一个target值，找到在BST中最接近给定值的节点值

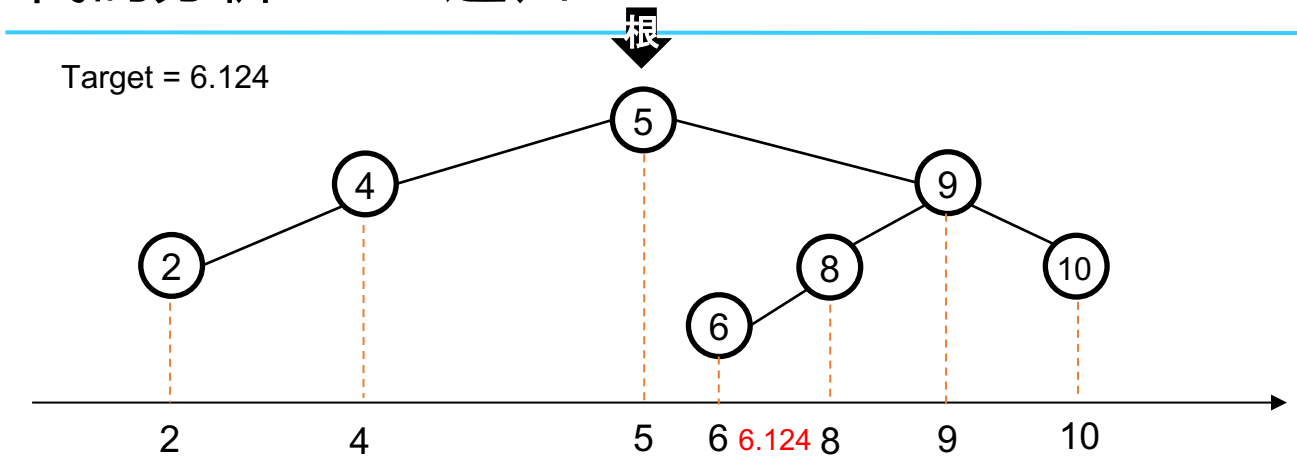
给出的目标值为浮点数

我们可以保证只有唯一1个最接近给定值的节点

```
输入: root = {5,4,9,2,#,8,10} and target = 6.124780
输出: 5
解释:
二叉树 {5,4,9,2,#,8,10}, 表示如下的树结构:
        5
       / \
     4    9
    /    / \
   2    8  10
```

```
输入: root = {3,2,4,1} and target = 4.142857
输出: 4
解释:
二叉树 {3,2,4,1}, 表示如下的树结构:
     3
    / \
  2    4
 /
1
```

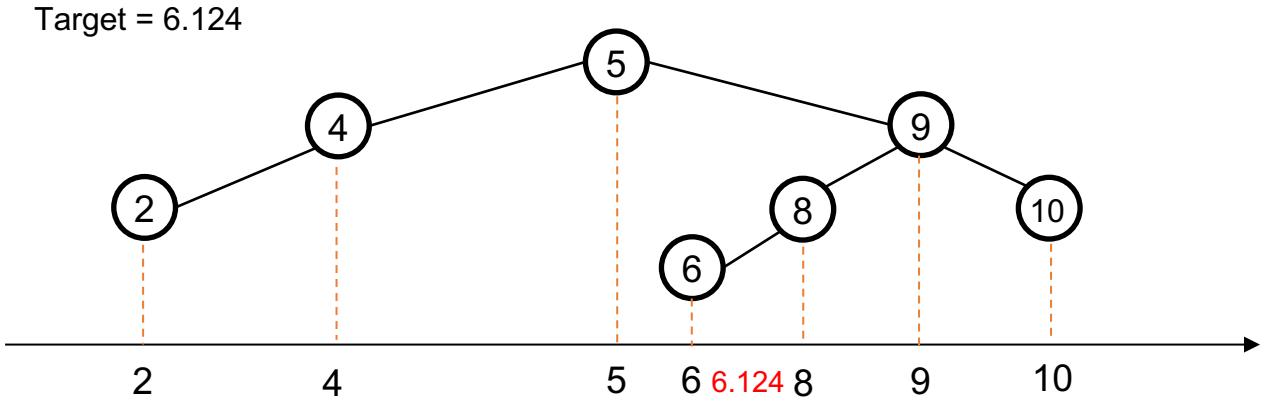
代码分析 —— 递归




```java
public int closestValue(TreeNode root, double target) {
    if (root == null) {
        return 0;
    }

    TreeNode lowerNode = lowerBound(root, target);
    TreeNode upperNode = upperBound(root, target);
    System.out.println(lowerNode.val);
    System.out.println(upperNode.val);
    if (lowerNode == null) {
        return upperNode.val;
    }

    if (upperNode == null) {
        return lowerNode.val;
    }

    if (target - lowerNode.val > upperNode.val - target) {
        return upperNode.val;
    }

    return lowerNode.val;
}
```

```java
// 找到小于等于target的最大值
private TreeNode lowerBound(TreeNode root, double target) {
    if (root == null) {
        return null;
    }

    if (target < root.val) {
        return lowerBound(root.left, target);
    }

    // root.val <= target, root已经是一个lower bound
    // 继续在右子树中寻找更接近target的lower bound
    TreeNode lowerNode = lowerBound(root.right, target);
    // 如果找到了更接近target的lower bound, 则返回, 否则返回root
    return (lowerNode != null) ? lowerNode : root;
}

// 找到比大于target的最小值
private TreeNode upperBound(TreeNode root, double target) {
    if (root == null) {
        return null;
    }

    if (root.val <= target) {
        return upperBound(root.right, target);
    }

    // target < root.val, root已经是一个upper bound
    // 继续在左子树中寻找更接近target的upper bound
    TreeNode upperNode = upperBound(root.left, target);
    // 如果找到了更接近target的upper bound, 则返回, 否则返回root
    return (upperNode != null) ? upperNode : root;
}
```

代码分析 —— 递归

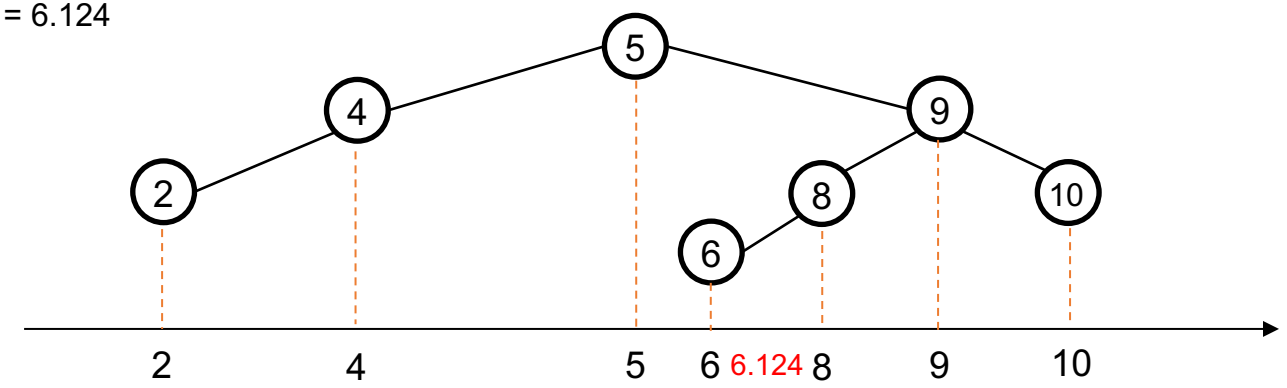



```python
    def closestValue(self, root, target):
        if root is None:
            return None

        lower = self.get_lower_bound(root, target)
        upper = self.get_upper_bound(root, target)

        if lower is None:
            return upper.val
        if upper is None:
            return lower.val

        if target - lower.val < upper.val - target:
            return lower.val
        return upper.val
```

```python
    # 找到小于等于target的最大值
    def get_lower_bound(self, root, target):
        if root is None:
            return None

        if target < root.val:
            return self.get_lower_bound(root.left, target)

        # root.val <= target, root已经是一个lower bound
        # 继续在右子树中寻找更接近target的lower bound
        lower = self.get_lower_bound(root.right, target)
        # 如果找到了更接近target的lower bound，则返回，否则返回root
        return root if lower is None else lower

    # 找到比大于target的最小值
    def get_upper_bound(self, root, target):
        if root is None:
            return None

        if root.val <= target:
            return self.get_upper_bound(root.right, target)

        # target < root.val, root已经是一个upper bound
        # 继续在左子树中寻找更接近target的upper bound
        upper = self.get_upper_bound(root.left, target)
        # 如果找到了更接近target的upper bound，则返回，否则返回root
        return root if upper is None else upper
```

代码分析 —— 非递归

Target = 6.124




```python
def closestValue(self, root, target):
    # 起始上下边界均为root
    upper = root
    lower = root

    while root:
        # 如果target比root大
        # 那么target不可能在左子树，只可能在根节点以及右子树
        # 下边界（low）设置为root，去右子树继续探索
        if root.val < target:
            lower = root
            root = root.right
        # 如果target比root小
        # 那么target不可能在右子树，只可能在根节点以及左子树
        # 上边界（high）设置为root，去左子树继续探索
        elif target < root.val:
            upper = root
            root = root.left
        # target == root.val，找到target，返回
        else:
            return root.val
    is_upper_closer = abs(upper.val - target) <= abs(lower.val - target)
    # 返回上下边界中更接近的值
    return upper.val if is_upper_closer else lower.val
```

```java
public int closestValue(TreeNode root, double target) {
    // 起始上下边界均为root
    TreeNode upper = root;
    TreeNode lower = root;

    while (root != null){
        // 如果target比root大，
        // 那么target不可能在左子树，只可能在根节点以及右子树
        // 下边界（low）设置为root，去右子树继续探索
        if (root.val < target){
            lower = root;
            root = root.right;
        }
        // 如果target比root小，
        // 那么target不可能在右子树，只可能在根节点以及左子树
        // 上边界（high）设置为root，去左子树继续探索
        else if (target < root.val){
            upper = root;
            root = root.left;
        }
        // target == root.val，找到target，返回
        else {
            return root.val;
        }
    }
    boolean isUpperCase = Math.abs(upper.val - target)
                        <= Math.abs(target - lower.val);

    // 返回上下边界中更接近的值
    return isUpperCase ? upper.val : lower.val;
}
```

901 Closest Binary Search Tree Value II 二叉搜索树中最接近的值 II

Given a non-empty binary search tree and a target value, find k values in the BST that are closest to the target.

给定一棵非空二叉搜索树以及一个target值，找到 BST 中最接近给定值的 k 个数。

给出的target值为浮点数

你可以假设 k 总是合理的，即 k ≤ 总节点数

我们可以保证给出的 BST 中只有唯一一个最接近给定值的 k 个值的集合

```
输入:
{1}
0.000000
1
输出:
[1]
解释:
二叉树 {1}，表示如下的树结构:
 1
```

```
输入:
{3,1,4,#,2}
0.275000
2
输出:
[1,2]
解释:
二叉树 {3,1,4,#,2}，表示如下的树结构:
  3
 /  \
1    4
 \
  2
```

不是求最近的一个值，而是求最近的K个值

思路分析

1. 用 inorder traversal 求出中序遍历
2. 用二分法找到第一个 >= target 的位置 index
3. 从 index-1 和 index 出发，设置两根指针一左一右，获得最近的 k 个整数

直播课第三章二分法做过类似题目

460 Find K Closest Elements在排序数组中找最接近的K个数

九章算法 www.jiuzhang.com

Given target, a non-negative integer k and an integer array A sorted in ascending order, find the k closest numbers to target in A, sorted in ascending order by the difference between the number and target. Otherwise, sorted in ascending order by number if the difference is same.

给一个目标数 target, 一个非负整数 k, 一个按照升序排列的数组 A。在A中找与target最接近的k个整数。返回这k个数并按照与target的接近程度从小到大排序，**如果接近程度相当，那么小的数排在前面。**

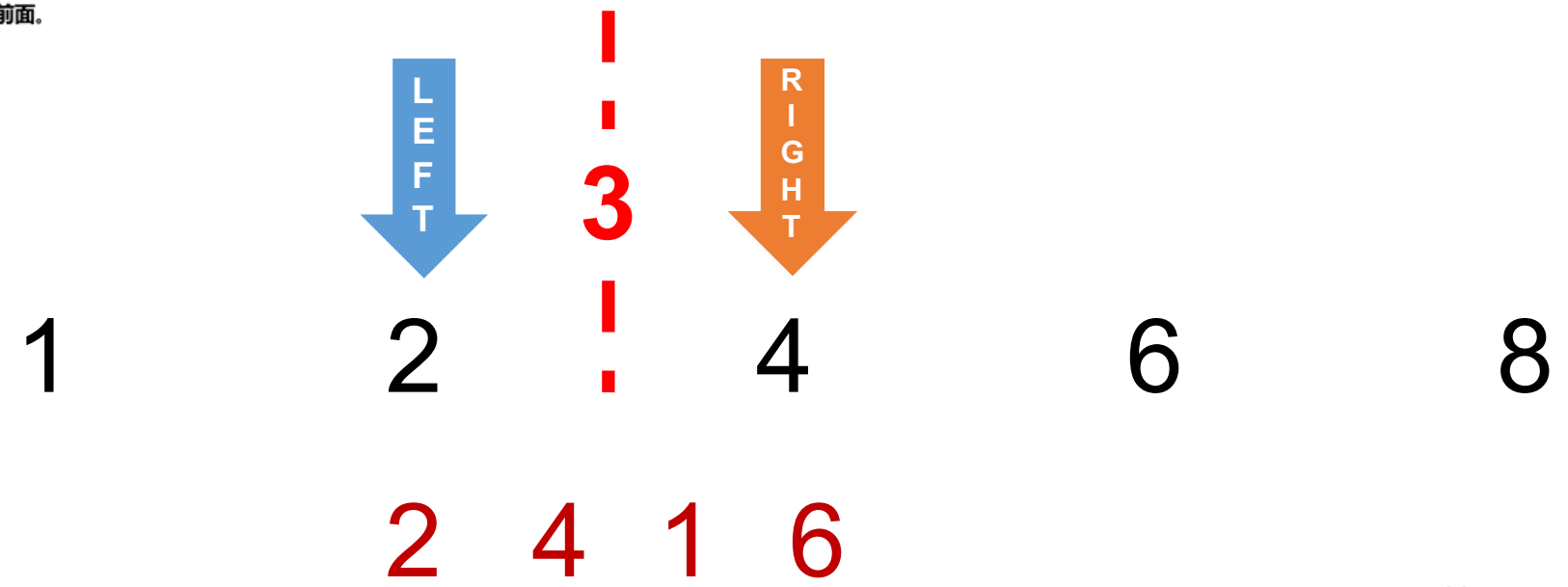
输入: A = [1, 2, 4, 6, 8], target = 3, k = 4

输出: [2, 4, 1, 6]

target值有可能不在数据中

array + target + sorted ➡ binary search

target = 3, k = 4



Related Questions

- Search Range in Binary Search Tree
- http://www.lintcode.com/problem/search-range-in-binary-search-tree/
- Insert Node in a Binary Search Tree
- http://www.lintcode.com/problem/insert-node-in-a-binary-search-tree/
- Remove Node in a Binary Search Tree
- http://www.lintcode.com/problem/remove-node-in-binary-search-tree/
- http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/BST-delete.html

在第 7 周的互动课中继续学习如下二叉树的内容

- 第 34 章 后序遍历非递归与 Morris 算法
  - 不作为必须要掌握的知识点，但是学了可以提高 Coding 能力
- 第 35 章 二叉查找树的增删查改
  - 必须掌握增查改，删除操作不作要求

做作业

做ladder

群里提问题

看回放

课前预习

课后复习

刷题

互动课