

解决99%二叉树问题的算法—— 分治法 Divide & Conquer

主讲人 令狐冲
课程版本 v7.0

版权声明

九章的所有课程均受法律保护，不允许录像与传播录像
一经发现，将被追究法律责任和赔偿经济损失

分治法 Divide & Conquer

将大规模问题拆分为若干个小规模的**同类型问题**去处理的算法
分治法和二分法（Binary Search）有什么区别？

分治法 Divide & Conquer

二分法会每次丢弃掉一半
分治法分完以后两边都要处理



什么样的数据结构适合分治法？

数组：一个大数组可以拆分为若干个不相交的子数组

归并排序，快速排序，都是基于数组的分治法

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

独孤九剑 —— 破枪式

碰到二叉树的问题，就想想整棵树在该问题上的结果和左右儿子在该问题上的结果之间的联系是什么

二叉树的高度是多少？

A: $O(n)$

B: $O(\log n)$

C: $O(h)$

二叉树的高度是多少？

最坏 $O(n)$ 最好 $O(\log n)$
一般用 $O(h)$ 来表示更合适

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）

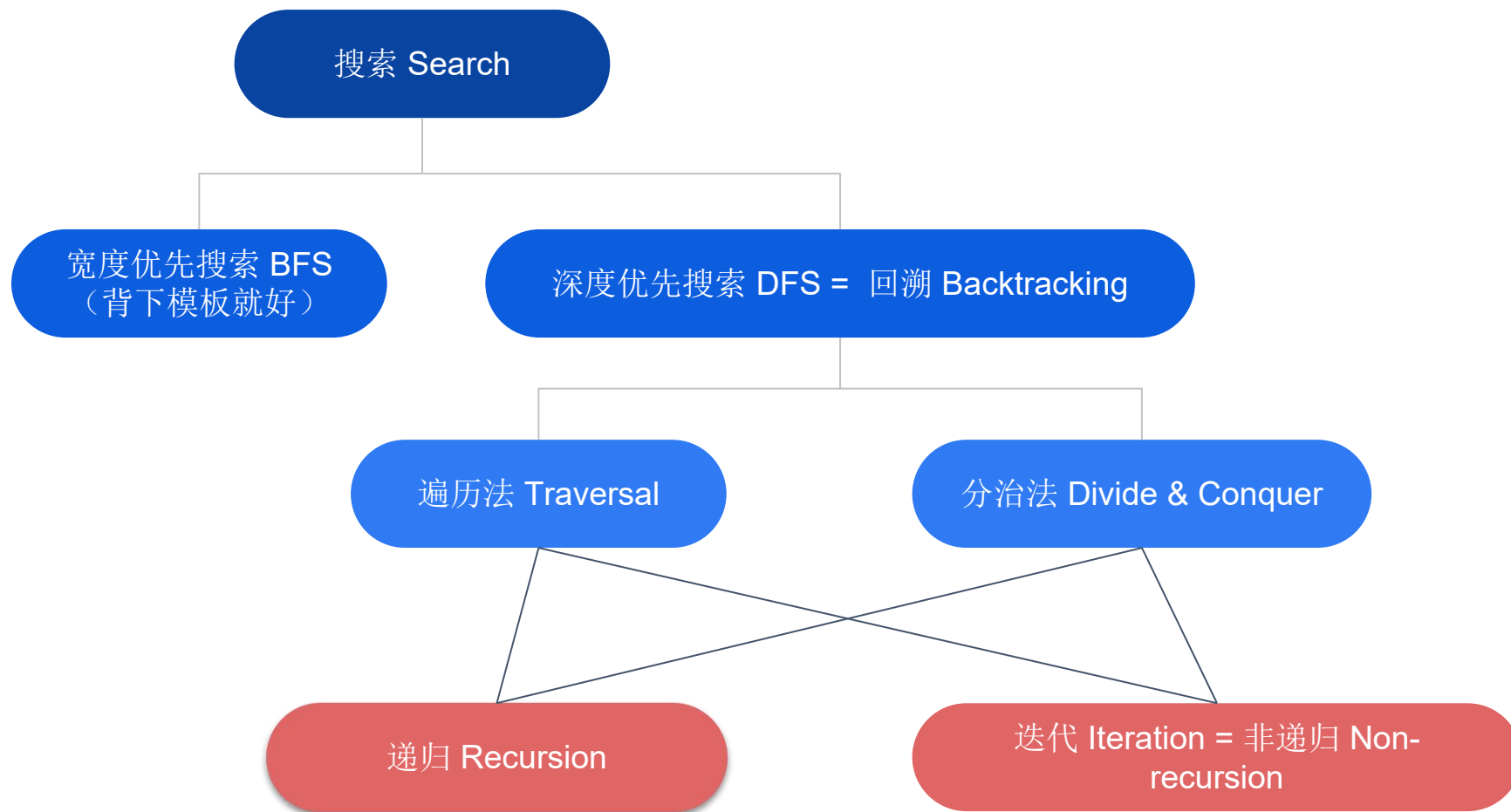
考察形态：二叉查找树（Binary Search Tree）

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）

Tree-based Depth First Search

不管二叉树的题型如何变化
考点都是基于树的深度优先搜索



将递归和非递归理解为算法的一种实现方式而不是算法

第一类考察形态

二叉树上求值，求路径

Maximum / Minimum / Average / Sum / Paths

Minimum Subtree

<http://www.lintcode.com/problem/minimum-subtree/>

<http://www.jiuzhang.com/solutions/minimum-subtree/>

求和最小的子树

一棵二叉树有多少棵子树

A: $O(n)$

B: $O(2^n)$

C: $O(n^2)$

D: $O(n)$ - 叶子节点个数

代码 - 使用了全局变量的分治法

```
public class Solution {
    private int minSum;
    private TreeNode minRoot;

    public TreeNode findSubtree(TreeNode root) {
        minSum = Integer.MAX_VALUE;
        minRoot = null;
        getSum(root);
        return minRoot;
    }

    private int getSum(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int sum = getSum(root.left) + getSum(root.right) + root.val;
        if (sum < minSum) {
            minSum = sum;
            minRoot = root;
        }

        return sum;
    }
}
```

```
class Solution:
    def findSubtree(self, root):
        self.minimum_weight = float('inf')
        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
    def getTreeSum(self, root):
        if root is None:
            return 0

        left_weight = self.getTreeSum(root.left)
        right_weight = self.getTreeSum(root.right)
        root_weight = left_weight + right_weight + root.val

        if root_weight < self.minimum_weight:
            self.minimum_weight = root_weight
            self.minimum_subtree_root = root

        return root_weight
```

全局变量的坏处

函数不“纯粹”，容易出 Bug

不利于多线程化，对共享变量加锁带来效率下降


```
class Solution:

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

    def helper(self, root):
        if root is None:
            return sys.maxsize, None, 0

        left_minimum, left_subtree, left_sum = self.helper(root.left)
        right_minimum, right_subtree, right_sum = self.helper(root.right)

        sum_of_root = left_sum + right_sum + root.val
        if left_minimum == min(left_minimum, right_minimum, sum_of_root):
            return left_minimum, left_subtree, sum_of_root
        if right_minimum == min(left_minimum, right_minimum, sum_of_root):
            return right_minimum, right_subtree, sum_of_root

        return sum_of_root, root, sum_of_root
```



```
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);

    ResultType result = new ResultType(
        node,
        leftResult.sum + rightResult.sum + node.val,
        leftResult.sum + rightResult.sum + node.val
    );

    if (leftResult.minSum <= result.minSum) {
        result.minSum = leftResult.minSum;
        result.minSubtree = leftResult.minSubtree;
    }

    if (rightResult.minSum <= result.minSum) {
        result.minSum = rightResult.minSum;
        result.minSubtree = rightResult.minSubtree;
    }

    return result;
}
```

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

```
class ResultType {
    public TreeNode minSubtree;
    public int sum, minSum;
    public ResultType(TreeNode minSubtree, int minSum, int sum) {
        this.minSubtree = minSubtree;
        this.minSum = minSum;
        this.sum = sum;
    }
}
```

Lowest Common Ancestor

<http://www.lintcode.com/problem/lowest-common-ancestor/>

<http://www.jiuzhang.com/solutions/lowest-common-ancestor/>

with parent pointer vs no parent pointer

follow up: LCA II & III

问法1：如果有父指针

<http://www.lintcode.com/problem/lowest-common-ancestor-ii/>

<http://www.jiuzhang.com/solutions/lowest-common-ancestor-ii/>

使用 **HashSet** 记录从 **A** 到根的所有点
访问从 **B** 到根的所有点，第一个出现在 **HashSet** 中的就是

```
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;
    }
}
```



```
def lowestCommonAncestorII(self, root, A, B):
    parentSet = set()
    # 把A的祖先节点都加入到哈希表中
    curr = A
    while (curr is not None):
        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 is not None):
        if (curr in parentSet):
            return curr
        curr = curr.parent
    return None
```

问法2：两个节点都在树里

<http://www.lintcode.com/problem/lowest-common-ancestor/>

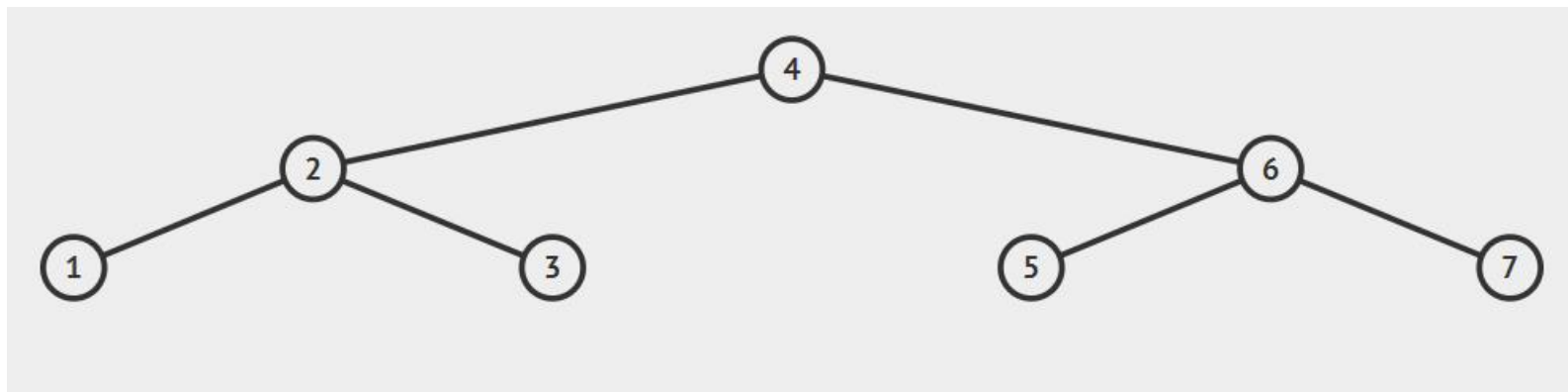
<http://www.jiuzhang.com/solutions/lowest-common-ancestor/>

给你 root, A, B 三个点的信息

A和B保证都在 root 的下面

方法一：直接遍历的方法

找出两个节点遍历之间深度最小的节点



遍历顺序													
node	4	2	1	2	3	2	4	6	5	6	7	6	4
depth	1	2	3	2	3	2	1	2	3	2	3	2	1

方法2： 分治法

定义返回值：

A,B 都存在 \rightarrow return LCA(A,B)

只有A \rightarrow return A

只有B \rightarrow return B

A,B 都不存在 \rightarrow return null

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode A, TreeNode B) {  
    if (root == null){  
        return null;  
    }  
    if (root == A || root == B){  
        return root;  
    }  
  
    TreeNode left = lowestCommonAncestor(root.left, A, B);  
    TreeNode right = lowestCommonAncestor(root.right, A, B);  
    if (left != null && right != null){  
        return root;  
    }  
    if (left != null){  
        return left;  
    }  
    if (right != null){  
        return right;  
    }  
    return null;  
}
```

```
def lowestCommonAncestor(self, root, A, B):  
    if root is None:  
        return None  
    if root == A or root == B:  
        return root  
    left = self.lowestCommonAncestor(root.left, A, B)  
    right = self.lowestCommonAncestor(root.right, A, B)  
    if left and right:  
        return root  
    if left:  
        return left  
    if right:  
        return right  
    return None
```

问法3：两个节点不一定都在树里

<http://www.lintcode.com/problem/lowest-common-ancestor-iii/>

<http://www.jiuzhang.com/solutions/lowest-common-ancestor-iii/>

root, p, q

但是不保证 root 里一定有 p 和 q

```
public TreeNode lowestCommonAncestor3(TreeNode root,
                                     TreeNode A,
                                     TreeNode B) {
    ResultType rt = helper(root, A, B);
    if (rt.a_exist && rt.b_exist)
        return rt.node;
    else
        return null;
}

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 ResultType helper(TreeNode root, TreeNode A, TreeNode B) {
    if (root == null)
        return new ResultType(false, false, null);

    ResultType left_rt = helper(root.left, A, B);
    ResultType right_rt = helper(root.right, A, B);

    boolean a_exist = left_rt.a_exist || right_rt.a_exist || root == A;
    boolean b_exist = left_rt.b_exist || right_rt.b_exist || root == B;

    if (root == A || root == B)
        return new ResultType(a_exist, b_exist, root);

    if (left_rt.node != null && right_rt.node != null)
        return new ResultType(a_exist, b_exist, root);
    if (left_rt.node != null)
        return new ResultType(a_exist, b_exist, left_rt.node);
    if (right_rt.node != null)
        return new ResultType(a_exist, b_exist, right_rt.node);

    return new ResultType(a_exist, b_exist, null);
}
```

```
import copy
class Solution:

    def lowestCommonAncestor3(self, root, A, B):
        a, b, lca = self.helper(root, A, B)
        if a and b:
            return lca
        else:
            return None
```

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

    left_a, left_b, left_node = self.helper(root.left, A, B)
    right_a, right_b, right_node = self.helper(root.right, A, B)

    a = left_a or right_a or root == A
    b = left_b or right_b or root == B

    if root == A or root == B:
        return a, b, root

    if left_node is not None and right_node is not None:
        return a, b, root
    if left_node is not None:
        return a, b, left_node
    if right_node is not None:
        return a, b, right_node

    return a, b, None
```

TAKE A BREAK

第二类考察形态

二叉树结构变化

Flatten Binary Tree to Linked List

<http://www.lintcode.com/problem/flatten-binary-tree-to-linked-list/>

<http://www.jiuzhang.com/solutions/flatten-binary-tree-to-linked-list/>

将二叉树拆成链表

进行前序遍历，将上一个节点的右指针指向当前节点


```
public class Solution {
    TreeNode prevNode = null;
    public void flatten(TreeNode root) {
        if (root == null) {
            return;
        }

        if (prevNode != null) {
            prevNode.left = null;
            prevNode.right = root;
        }

        prevNode = root;
        TreeNode right = root.right;
        flatten(root.left);
        flatten(right);
    }
}
```

```
class Solution:
    prev_node = None
    def flatten(self, root):
        if root is None:
            return

        if self.prev_node is not None:
            self.prev_node.left = None
            self.prev_node.right = root

        self.prev_node = root
        right = root.right
        self.flatten(root.left)
        self.flatten(right)
```

尽可能避免使用全局变量

容易写出 BUG

可以把需要修改的变量作为参数传入到函数里
或者是放在 `return value` 里

```
public class Solution {
    public void flatten(TreeNode root) {
        flatten_and_return_last_node(root);
    }

    private TreeNode flatten_and_return_last_node(TreeNode root) {
        if (root == null) {
            return null;
        }

        TreeNode left_last = flatten_and_return_last_node(root.left);
        TreeNode right_last = flatten_and_return_last_node(root.right);

        // connect
        if (left_last != null) {
            left_last.right = root.right;
            root.right = root.left;
            root.left = null;
        }

        if (right_last != null) {
            return right_last;
        }
        if (left_last != null) {
            return left_last;
        }
        if (root != null) {
            return root;
        }

        return null;
    }
}
```

```
class Solution:
    def flatten(self, root):
        self.flatten_and_return_last_node(root)

    def flatten_and_return_last_node(self, root):
        if root is None:
            return None

        left_last = self.flatten_and_return_last_node(root.left)
        right_last = self.flatten_and_return_last_node(root.right)

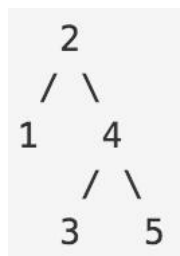
        # connect
        if left_last is not None:
            left_last.right = root.right
            root.right = root.left
            root.left = None

        return right_last or left_last or root
```

第三类考察形态

二叉搜索树
Binary Search Tree

- 从定义出发：
 - 左子树都比根节点小
 - 右子树都不小于根节点
- 从效果出发：
 - 中序遍历 in-order traversal 是“不下降”序列
 - 如图，中序遍历为 1 2 3 4 5



- 性质：
 - 如果一棵二叉树的中序遍历不是“不下降”序列，则一定不是BST
 - 如果一棵二叉树的中序遍历是不下降，也未必是BST
 - 比如下面这棵树就不是 BST，但是它的中序遍历是不下降序列。
 - 1
 - / \
 - 1 1

BST的高度是多少

A: $O(n)$

B: $O(\log n)$

C: $O(h)$

BST的高度是多少

同样是最坏 $O(n)$ 最好 $O(\log n)$

用 $O(h)$ 表示更合适

只有 **Balanced Binary Tree** (平衡二叉树) 才是 $O(\log n)$

红黑树 Red-black Tree

红黑树是一种 **Balanced BST**

C++/Java中有一个数据结构的实现用的是 **Balanced BST**

你知道是什么吗？

Java: TreeMap / TreeSet

C++: map / set

Python: 不好意思我没有

有一一定要用 TreeMap 来做的面试题么？——没有

关于红黑树，你需要掌握的是

- 知道他是一个 **Balanced BST**
- 知道他能干嘛
 - $O(\log N)$ 的时间内实现增删查改
 - $O(\log N)$ 的时间内实现找最大找最小
 - $O(\log N)$ 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(**lowerBound**)
- 知道他的工程应用价值
 - Java 1.8 中的 **HashMap** 的实现里同时用到了 **TreeMap** 和 **LinkedList**

就面试而言，你不需要掌握他的代码实现，因为写出来**200+**行，没人能在面试的时候写完

Kth Smallest Element in BST

<https://www.lintcode.com/problem/kth-smallest-element-in-a-bst/>

<https://www.jiuzhang.com/solution/kth-smallest-element-in-a-bst/>

时间复杂度如何分析？



```
public class Solution {
    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<>();

        while (root != null) {
            stack.push(root);
            root = root.left;
        }

        for (int i = 0; i < k - 1; i++) {
            TreeNode node = stack.peek();

            if (node.right == null) {
                node = stack.pop();
                while (!stack.isEmpty() && stack.peek().right == node) {
                    node = stack.pop();
                }
            } else {
                node = node.right;
                while (node != null) {
                    stack.push(node);
                    node = node.left;
                }
            }
        }

        return stack.peek().val;
    }
}
```

```
class Solution:
    def kthSmallest(self, root, k):
        # use binary tree iterator
        dummy = TreeNode(0)
        dummy.right = root
        stack = [dummy]

        for i in range(k):
            node = stack.pop()
            if node.right:
                node = node.right
                while node:
                    stack.append(node)
                    node = node.left

            if not stack:
                return None

        return stack[-1].val
```

时间复杂度分析

$$O(k + h)$$

当 k 是 1 的时候 $\implies O(h)$

当 k 是 n 的时候 $\implies O(n)$ 

k 和 h 两者取大值

Follow up: 二叉树经常被修改

如何优化 `kthSmallest` 这个操作？

在 `TreeNode` 中增加一个 `counter`，代表整个树的节点个数

也可以用一个 `HashMap<TreeNode, Integer>` 来存储某个节点为代表的子树的节点个数

在增删查改的过程中记录不断更新受影响节点的 `counter`

在 `kthSmallest` 的实现中用类似 `Quick Select` 的算法去找到 `kth smallest element`

时间复杂度为 $O(h)$ ， h 为树的高度。

Strong Hire: 能够答出 `Follow Up` 的算法，并写出 `kthSmallest` 核心代码（不需要写增删查改，45分钟写不完的）， `bug free or minor bug`，不需要提示

Hire / Weak Hire : 能够答出 `Follow up` 的算法，大致写出 `kthSmallest` 核心代码，存在一定bug，或者需要提示

No Hire: 答不出 `follow up`

Strong No: 连第一问的 `Inorder traversal` 都不会写

Closest Binary Search Tree Value

<https://www.lintcode.com/problem/closest-binary-search-tree-value/>

<http://www.jiuzhang.com/solution/closest-binary-search-tree-value/>

如果使用中序遍历，时间复杂度是多少？

如果使用 `lowerBound` / `upperBound` 的方法，时间复杂度是多少？

Follow up: 寻找 k 个最接近的值

<https://www.lintcode.com/problem/closest-binary-search-tree-value-ii/>

<https://www.jiuzhang.com/solution/closest-binary-search-tree-value-ii/>

如果是用中序遍历得到从小到大的所有值，接下来的问题相当于之前学过的哪个题？

有没有更快的办法？

方法1 暴力做法

先用 inorder traversal 求出中序遍历

找到第一个 $\geq \text{target}$ 的位置 index

从 $\text{index}-1$ 和 index 出发，设置两根指针一左一右，获得最近的 k 个整数

```
public List<Integer> closestKValues(TreeNode root, double target, int k) {
    List<Integer> values = new ArrayList<>();

    traverse(root, values);

    int i = 0, n = values.size();
    for (; i < n; i++) {
        if (values.get(i) >= target) {
            break;
        }
    }

    if (i >= n) {
        return values.subList(n - k, n);
    }

    int left = i - 1, right = i;
    List<Integer> result = new ArrayList<>();
    for (i = 0; i < k; i++) {
        if (left >= 0 && (right >= n || target - values.get(left) < values.get(right) -
            target)) {
            result.add(values.get(left));
            left--;
        } else {
            result.add(values.get(right));
            right++;
        }
    }

    return result;
}

private void traverse(TreeNode root, List<Integer> values) {
    if (root == null) {
        return;
    }

    traverse(root.left, values);
    values.add(root.val);
    traverse(root.right, values);
}
```

```
def closestKValues(self, root, target, k):
    if root is None or k == 0:
        return []

    nums = self.get_inorder(root)
    left = self.find_lower_index(nums, target)
    right = left + 1
    results = []
    for _ in range(k):
        if self.is_left_closer(nums, left, right, target):
            results.append(nums[left])
            left += 1
        else:
            results.append(nums[right])
            right += 1
    return results
```

```
def get_inorder(self, root):
    dummy = TreeNode(0)
    dummy.right = root
    stack = [dummy]
    inorder = []

    while stack:
        node = stack.pop()
        if node.right:
            node = node.right
            while node:
                stack.append(node)
                node = node.left
        if stack:
            inorder.append(stack[-1].val)

    return inorder
```

```
def find_lower_index(self, nums, target):
    start, end = 0, len(nums) - 1
    while start + 1 < end:
        mid = (start + end) // 2
        if nums[mid] < target:
            start = mid
        else:
            end = mid

    if nums[end] < target:
        return end

    if nums[start] < target:
        return start

    return -1

def is_left_closer(self, nums, left, right, target):
    if left < 0:
        return False
    if right >= len(nums):
        return True
    return target - nums[left] < nums[right] - target
```

方法2 使用两个 Iterator

一个 iterator move forward

另一个iterator move backward

每次 $i++$ 的时候根据 `stack`, 挪动到 `next node`

每次 $i--$ 的时候根据 `stack`, 挪动到 `prev node`

```
public List<Integer> closestKValues(TreeNode root, double target, int k) {
    List<Integer> values = new ArrayList<>();

    if (k == 0 || root == null) {
        return values;
    }

    Stack<TreeNode> lowerStack = getStack(root, target);
    Stack<TreeNode> upperStack = new Stack<>();
    upperStack.addAll(lowerStack);
    if (target < lowerStack.peek().val) {
        moveLower(lowerStack);
    } else {
        moveUpper(upperStack);
    }

    for (int i = 0; i < k; i++) {
        if (lowerStack.isEmpty() ||
            !upperStack.isEmpty() && target - lowerStack.peek().val > upperStack
                .peek().val - target) {
            values.add(upperStack.peek().val);
            moveUpper(upperStack);
        } else {
            values.add(lowerStack.peek().val);
            moveLower(lowerStack);
        }
    }

    return values;
}

private Stack<TreeNode> getStack(TreeNode root, double target) {
    Stack<TreeNode> stack = new Stack<>();

    while (root != null) {
        stack.push(root);

        if (target < root.val) {
            root = root.left;
        } else {
            root = root.right;
        }
    }

    return stack;
}
```



```
public void moveUpper(Stack<TreeNode> stack) {
    TreeNode node = stack.peek();
    if (node.right == null) {
        node = stack.pop();
        while (!stack.isEmpty() && stack.peek().right == node) {
            node = stack.pop();
        }
        return;
    }

    node = node.right;
    while (node != null) {
        stack.push(node);
        node = node.left;
    }
}

public void moveLower(Stack<TreeNode> stack) {
    TreeNode node = stack.peek();
    if (node.left == null) {
        node = stack.pop();
        while (!stack.isEmpty() && stack.peek().left == node) {
            node = stack.pop();
        }
        return;
    }

    node = node.left;
    while (node != null) {
        stack.push(node);
        node = node.right;
    }
}
```



```
def closestKValues(self, root, target, k):
    if root is None or k == 0:
        return []

    lower_stack = self.get_stack(root, target)
    upper_stack = list(lower_stack)
    if lower_stack[-1].val < target:
        self.move_upper(upper_stack)
    else:
        self.move_lower(lower_stack)

    result = []
    for i in range(k):
        if self.is_lower_closer(lower_stack, upper_stack, target):
            result.append(lower_stack[-1].val)
            self.move_lower(lower_stack)
        else:
            result.append(upper_stack[-1].val)
            self.move_upper(upper_stack)

    return result

def get_stack(self, root, target):
    stack = []
    while root:
        stack.append(root)
        if target < root.val:
            root = root.left
        else:
            root = root.right

    return stack
```

```
def move_upper(self, stack):
    if stack[-1].right:
        node = stack[-1].right
        while node:
            stack.append(node)
            node = node.left
    else:
        node = stack.pop()
        while stack and stack[-1].right == node:
            node = stack.pop()

def move_lower(self, stack):
    if stack[-1].left:
        node = stack[-1].left
        while node:
            stack.append(node)
            node = node.right
    else:
        node = stack.pop()
        while stack and stack[-1].left == node:
            node = stack.pop()

def is_lower_closer(self, lower_stack, upper_stack, target):
    if not lower_stack:
        return False

    if not upper_stack:
        return True

    return target - lower_stack[-1].val < upper_stack[-1].val - target
```

Strong Hire

找1个点和找k个点都答出来，且找 k 个点的能用 $O(k + \log n)$ 的时间复杂度

Hire

找1个点和找k个点都答出来，且找 k 个点的能用 $O(k \log n)$ 的时间复杂度完成，少 bug，无需提示

Weak Hire:

找1个点和找k个点都答出来，且找 k 个点的能分别用 $O(k \log n)$ 和 $O(n)$ 的时间复杂度完成，bug 多，需要提示

No Hire

答出1个点，答不出 k 个点非 $O(n)$ 的算法

Strong No Hire

啥都答不出来

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