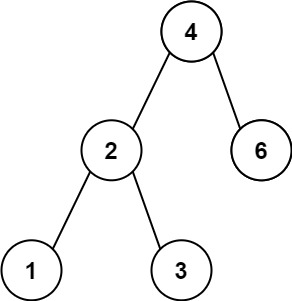
<https://leetcode.com/problems/minimum-absolute-difference-in-bst/description/>

Given the root of a Binary Search Tree (BST), return **the minimum absolute difference between the values of any two different nodes in the tree**.

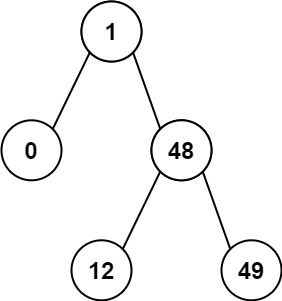
**Example 1:**



**Input:** root = [4,2,6,1,3]

**Output:** 1

**Example 2:**



**Input:** root = [1,0,48,null,null,12,49]

**Output:** 1

**Constraints:**

The number of nodes in the tree is in the range [2, 10^4].

0 <= Node.val <= 10^5

**Note:** This question is the same as 783: <https://leetcode.com/problems/minimum-distance-between-bst-nodes/>

**Attempt 1: 2024-01-14**

**Solution 1: Inorder Traversal (60 min)**

**Style 1: Need extra space (10 min)**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public int getMinimumDifference(TreeNode root) {

        List<Integer> list = new ArrayList<>();

        helper(root, list);

        int diff = (int)1e6 + 1;

        int prev = list.get(0);

        for(int i = 1; i < list.size(); i++) {

            diff = Math.min(diff, list.get(i) - prev);

            prev = list.get(i);

        }

        return diff;

    }

    private void helper(TreeNode root, List<Integer> list) {

        if(root == null) {

            return;

        }

        helper(root.left, list);

        list.add(root.val);

        helper(root.right, list);

    }

}

Time Complexity: O(N)

Space Compelxity: O(N)

**Style 2: No need extra space (60 min)**

**Wrong Solution**

**Test out by**

root = [236,104,701,null,227,null,911]

236

/ \

104 701

\ \

227 911

Expect: 9

Output: 123

**错误的原因是认为只有直接父子关系的节点能获得最小绝对值差，比如[104,227], [104,236], [237,701], [701,911]这些构成了直接父子关系，但是这些关系中的绝对值差最小只能是[104,227]这一对获得，227 - 104 = 123，但实际上真正的最小绝对值差来自非直接父子关系的一对，而是来自BST inorder traversal中位置上前后相邻的两个节点[227,236]，236 - 227 = 9，所以计算构造上有问题，不能只计算和比较直接父子关系的节点的绝对值差，而是需要利用BST的性质，对于每一个root节点，需要将该root节点左子树的所有节点与该root节点比较，同理也需要将该root节点右子树的所有节点与该root节点比较，比如该例子中对于root节点236，它左边所有节点是104和227，不能只像错误方法中只拿直接父子关系的236和104作比较，而是应该拿236同时与104和227比较，这样才不会漏掉[236,227]这样一对**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    public int getMinimumDifference(TreeNode root) {

        return helper(root);

    }

    private int helper(TreeNode root) {

        if(root == null) {

            return (int)1e6 + 1;

        }

        int diff = (int)1e6 + 1;

        if(root.left != null) {

            diff = Math.min(diff, root.val - root.left.val);

        }

        if(root.right != null) {

            diff = Math.min(diff, root.right.val - root.val);

        }

        int leftResult = helper(root.left);

        int rightResult = helper(root.right);

        return Math.min(diff, Math.min(leftResult, rightResult));

    }

}

**Correct Solution**

/\*\*

 \* Definition for a binary tree node.

 \* public class TreeNode {

 \*     int val;

 \*     TreeNode left;

 \*     TreeNode right;

 \*     TreeNode() {}

 \*     TreeNode(int val) { this.val = val; }

 \*     TreeNode(int val, TreeNode left, TreeNode right) {

 \*         this.val = val;

 \*         this.left = left;

 \*         this.right = right;

 \*     }

 \* }

 \*/

class Solution {

    TreeNode prev;

    int diff = (int)1e6 + 1;

    public int getMinimumDifference(TreeNode root) {

        prev = null;

        helper(root);

        return diff;

    }

    private void helper(TreeNode root) {

        if(root == null) {

            return;

        }

        helper(root.left);

        // During inorder traversal, after traverse left

        // child branch, we process current node:

        // 1. Update 'diff' by 'root.val - prev.val', and

        // since BST inorder traversal, no need root.val

        // always larger than prev.val, no need Math.abs()

        // 2. Update 'prev' to current 'root' before we

        // continue on right child branch

        if(prev != null) {

            diff = Math.min(diff, root.val - prev.val);

        }

        prev = root;

        helper(root.right);

    }

}

Time Complexity: O(N)

Space Complexity: O(h), where h is the height of the binary tree.

**Test code**

**关键debug步骤：**

**1.遍历从BST root = 236节点开始，根据inorder traversal，首先进入root节点236的左子结点分支，即root = 104，在root = 104节点继续进入其左子结点分支，此时因为root = 104节点左子结点为null，根据root == null的base condition直接返回root = 104节点，并根据inorder traversal开始在root = 104节点做当前节点处理，在处理root = 104节点的过程中我们使用prev = root将prev = null更新为prev = 104**

**2.然后进入root = 104节点的右子节点分支，到达节点root = 227，同样，对于root = 227节点我们也需要先进入其左子结点分支，此时因为root = 227节点左子结点为null，根据root == null的base condition直接返回root = 227节点，并根据inorder traversal开始在root = 227节点做当前节点处理，随后因为此时prev不再是null，我们更新diff：diff = Math.min((int)1e6 + 1, 227 - 104) = 123，并且在处理root = 227节点的过程中我们使用prev = root将prev = 104更新为prev = 227，最后进入root = 227的右子节点，此时因为root = 227节点右子结点为null，根据root == null的base condition直接返回root = 227节点，然后因为完成了所有语句，即完成了对root = 236节点的左子节点分支的遍历，自动返回上递归一层即root = 236节点这一层**

**3.此时我们拥有prev = 227，当前root = 236，根据inorder traversal的定义，在之前步骤1,2处理完root = 236节点的左子分支后，开始在root = 236节点做当前节点处理，随后因为此时prev不再是null，我们更新diff：diff = Math.min(123, 236 - 227) = 9，并且在处理root = 236节点的过程中我们使用prev = root将prev = 227更新为prev = 236，最后进入root = 236的右子节点.... etc**

**这里完美解释了如何通过inorder traversal配合prev = root的做法获得236 - 227 = 9这个关键值的过程，避免了错误做法中无法将非直接父子关系的节点放在一起比较的问题，所以本题的关键在于用prev来追踪inorder traversal中紧邻当前节点的前一个节点，他们可以不是直接父子关系，但是因为是BST上的inorder traversal，这两个节点即便在BST树形结构上不是父子节点也必定是除开树形结构排序后的相邻两节点**

public class TreeSolution {

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

TreeNode prev;

int diff = (int)1e6 + 1;

public int getMinimumDifference(TreeNode root) {

prev = null;

helper(root);

return diff;

}

private void helper(TreeNode root) {

if(root == null) {

return;

}

helper(root.left);

if(prev != null) {

diff = Math.min(diff, root.val - prev.val);

}

prev = root;

helper(root.right);

}

public static void main(String[] args) {

/\*\*

\* 236

\* / \

\* 104 701

\* \ \

\* 227 911

\*/

TreeSolution so = new TreeSolution();

TreeNode a = so.new TreeNode(236);

TreeNode b = so.new TreeNode(104);

TreeNode c = so.new TreeNode(701);

TreeNode d = so.new TreeNode(227);

TreeNode e = so.new TreeNode(911);

a.left = b;

a.right = c;

b.right = d;

c.right = e;

int result = so.getMinimumDifference(a);

System.out.println(result);

}

}

**Refer to**

<https://algo.monster/liteproblems/530>

**Problem Description**

This LeetCode problem asks us to find the minimum absolute difference between the values of any two different nodes in a Binary Search Tree (BST). A BST is a tree data structure where each node has at most two children, referred to as the left child and the right child. For any node in a BST, the value of all the nodes in the left subtree are less than the node's value, and the value of all the nodes in the right subtree are greater than the node's value.

**The "minimum absolute difference" refers to the smallest difference in value between any pair of nodes in the tree.** An important characteristic of a BST is that when it is traversed in-order (left node, current node, right node), it will yield the values in a sorted manner. This property will be crucial for finding the minimum absolute difference without having to compare every pair of nodes in the tree.

**Intuition**

To arrive at the solution, we leverage the in-order traversal property of the BST mentioned above. The idea is to perform an in-order traversal of the BST, which effectively means we will be processing the nodes in ascending order of their values.

During the traversal, we keep track of the value of the previously processed node.

At each current node, we calculate the absolute difference between the current node's value and the previously visited node's value.

We keep an ongoing count of the minimum absolute difference encountered so far.

By the end of the traversal, the minimum absolute difference will have been found.

**The intuition behind this approach is based on the fact that the smallest difference between any two values in a sorted list is always between adjacent values. Since an in-order traversal of a BST gives us a sorted list of values, we only need to check adjacent nodes to determine the minimum absolute difference in the entire tree.**

We use a helper function dfs (Depth-First Search) that performs the in-order traversal recursively. The nonlocal keyword is used to update the ans and prev variables defined in the outer scope of the dfs function.

**Solution Approach**

The provided Python solution makes use of recursion to implement the in-order traversal pattern for navigating the BST. Here's a step by step breakdown of how the implementation works:

A nested function named dfs (short for "Depth-First Search") is defined within the getMinimumDifference method. This dfs function is responsible for performing the in-order traversal of the BST.

The dfs function does nothing if it encounters a None node (the base case of the recursion), which means that it has reached a leaf node's child.

When the dfs function visits a node, it first recursively calls itself on the left child of the current node to ensure that the nodes are visited in ascending order.

**After visiting the left child, the function then processes the current node by calculating the absolute difference between the current node's value and the previously visited node's value, and keeps track of this absolute difference if it's the smallest one seen so far. This is done using the ans variable, which is initialized with infinity (inf). ans represents the minimum absolute difference found during the traversal.**

**The prev variable holds the value of the previously visited node in the traversal. Initially, prev is also initialized with inf, and it is updated to the current node's value before moving to the right child.**

Once the current node has been processed, the dfs function recursively calls itself on the right child to complete the in-order visitation pattern.

The nonlocal keyword is used for ans and prev to allow the nested dfs function to modify these variables that are defined in the enclosing getMinimumDifference method's scope.

The getMinimumDifference method initializes ans and prev with inf and begins the in-order traversal by calling the dfs function on the root node of the BST.

After the in-order traversal is complete, ans holds the minimum absolute difference between the values of any two nodes in the BST. This value is returned as the final result.

Through the use of in-order traversal, the algorithm ensures that each node is compared only with its immediate predecessor in terms of value, resulting in an efficient way to find the minimum absolute difference.

Here's the algorithm represented in pseudocode:

    def in\_order\_traversal(node):

        if node is not None:

            in\_order\_traversal(node.left)

            process(node)

            in\_order\_traversal(node.right)

    def process(node):

        update\_minimum\_difference(previous\_node\_value, node.value)

        previous\_node\_value = node.value

In the pseudocode, process(node) represents the steps of comparing the current node's value with the previous node's value and updating the minimum difference accordingly.

**Example Walkthrough**

Consider the following BST for this example:

    4

   / \

  2   6

 / \

1   3

Let's apply the in-order traversal to this BST and keep track of the previous node to calculate the differences and find the minimum absolute difference.

Start at the root (4), then traverse to the left child (2). Since 2 has a left child (1), continue traversing left until reaching a leaf node.

Process node 1: This is the first node, so there's no previous node to compare with. Set prev to 1.

Traverse up and to the right, now to node 2. Calculate the absolute difference with prev: |2 - 1| = 1, and since prev was set to 1 earlier, set ans to 1 (this is our first comparison, so it's also the smallest so far). Update prev to 2.

Process node 2: Node 2 is considered again, but the comparison has already been made with its left side. No left child traversal needed now.

Traverse to the right child of node 2, which is node 3. Calculate the absolute difference with prev: |3 - 2| = 1. Since ans is already 1 and |3 - 2| is also 1, ans remains unchanged. Update prev to 3.

Move up to node 4, the root. Calculate the absolute difference with prev: |4 - 3| = 1. The ans is still 1, so there's no change. Update prev to 4.

Finally, move to the right child of the root, which is node 6. Calculate the absolute difference with prev: |6 - 4| = 2. Since ans is 1, and |6 - 4| is greater than 1, there's no change to ans.

By the end of the traversal, ans holds the value of 1, which is the minimum absolute difference that was found between the values of adjacent nodes during the in-order traversal of the BST. Therefore, the minimum absolute difference between any two nodes in this BST is 1.

**Java Solution**

class Solution {

private int minDifference;

private int previousValue;

private static final int INFINITY = Integer.MAX\_VALUE; // Use a static final constant for infinity

/\*\*

\* Find the minimum absolute difference between values of any two nodes.

\*

\* @param root The root of the binary search tree.

\* @return The minimum absolute difference.

\*/

public int getMinimumDifference(TreeNode root) {

minDifference = INFINITY; // Initialize minimum difference to the largest value possible

previousValue = INFINITY; // Initialize previous value to the largest value possible for the start

inOrderTraversal(root); // Perform in-order traversal to compare node values

return minDifference; // Return the smallest difference found

}

/\*\*

\* Perform in-order traversal on BST to find minimum absolute difference.

\*

\* @param node The current node being visited.

\*/

private void inOrderTraversal(TreeNode node) {

if (node == null) {

return; // Base case: if node is null, return to stop the traversal

}

inOrderTraversal(node.left); // Visit left subtree

// Compute the minimum difference with the previous value (if not first node)

if (previousValue != INFINITY) {

minDifference = Math.min(minDifference, Math.abs(node.val - previousValue));

}

previousValue = node.val; // Update the previous value to the current node's value

inOrderTraversal(node.right); // Visit right subtree

}

}

/\*\*

\* Definition for a binary tree node.

\*/

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) {

this.val = val;

}

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

**Time and Space Complexity**

The time complexity of the code is O(n), where n is the number of nodes in the binary tree. This time complexity arises because the depth-first search (DFS) in the function dfs(root) visits each node exactly once. The actions performed on each node—including updating the 'ans' and 'prev' variables—are all O(1) operations, so they do not add to the overall complexity beyond that imposed by the traversal.

The space complexity of the code is O(h), where h is the height of the binary tree. This space complexity is due to the recursive call stack that will grow to the height of the tree in the worst case. For a balanced tree, this would be O(log n), but for a skewed tree (e.g., a tree where each node only has a left or a right child), this could degrade to O(n).