Problem Description

contain duplicates and the goal is to identify the mode(s), which are the values that appear most frequently. As a reminder, a BST has the property where each node's left child has a value less than or equal to its own value, while the right child has a value greater than or equal to its own value. This property must hold true for every node's left and right subtrees as well. The output should be a list of the modes. If there's more than one mode, they can be returned in any order. What makes this interesting is that we must traverse the tree and keep track of the frequency of each element without knowing beforehand which

The problem is about finding the most frequently occurring elements in a Binary Search Tree (BST). The challenge is that a BST may

values appear in the BST. Intuition

To solve this problem, one effective approach is to perform an in-order traversal of the BST. An in-order traversal is a type of depth-

first traversal that visits the left subtree, then the root node, and finally the right subtree. This traversal order guarantees that the values are accessed in a sorted manner because of the BST property.

As we perform the traversal, we can track the current value, its count, and compare it with the maximum frequency (or mode) encountered so far. There are a few edge cases to consider: If the current value is the same as the previous value we saw, we increment the count for this value. If it's a new value (not the same as the previous one), we reset the count to 1.

 If the count for the current value is greater than the current maximum frequency, we update the maximum frequency and reset the list of modes to now only include this new value.

A nonlocal variable is used to maintain state between recursive calls. Variables like mx (short for 'maximum'), prev (short for 'previous value'), ans (short for 'answer'), and cnt (short for 'count') are updated as the in-order traversal proceeds. After the traversal is

This approach efficiently computes the mode(s) using the BST's inherent properties, all without needing additional data structures to

Solution Approach

• mx: This keeps track of the maximum frequency of any value encountered so far.

ans: This list stores the mode(s) — the values with the highest frequency seen so far.

Otherwise, cnt is set to 1 since we're seeing this value for the first time.

Recursive calls are made to visit the right subtree: dfs(root.right).

Let's illustrate the solution with a small example. Suppose our BST looks like this:

Here is a step-by-step explanation of the function dfs which is called with the root of the BST:

complete, the ans list will contain the mode(s) of the BST.

track frequency of all elements, thus saving space.

If the count matches the current maximum frequency, we add the current value to the list of modes.

The solution uses a Depth-First Search (DFS) in-order traversal algorithm to walk through the tree. Since it's a BST, an in-order traversal will access the nodes in ascending order. This property is critical because it allows us to track the mode(s) without the need for additional data structures like a hash table to count the frequencies of each value.

In the Python solution, a nested function dfs is defined, capturing the variables mx, prev, ans, and cnt from the enclosing findMode function scope using the nonlocal keyword. These captured variables serve as state throughout the traversal:

• prev: This holds the previous value processed in the in-order traversal. It's used to compare with the current node's value to

determine if the count should be increased or reset. cnt: This records the current count of consecutive nodes with the same value.

 The base case checks if the current node root is None, and if it is, the function returns because there's nothing to process. Recursive calls are made to visit the left subtree: dfs(root.left). When control returns from the left subtree, the current node is processed. If prev is the same as root.val, cnt is incremented.

a list containing just root.val. • If cnt equals mx, we append root.val to ans because it's a mode with the same frequency as the current maximum. prev is updated to the current node's value: prev = root.val.

• We then check if cnt is greater than mx. If it is, this is the new maximum frequency, and so mx is updated to cnt, and ans is set to

and space. Example Walkthrough

After the dfs function has completed the in-order traversal, the ans list will have the mode(s). The Solution class's findMode function

then returns ans. The result is that all modes in the BST are found in a single pass through the tree, which is an efficient use of time

The BST contains duplicates, and we want to find the mode. In this case, the mode is 2, since it occurs more frequently than any other number. Now, let's walk through how the solution processes this BST:

1. We start with an in-order traversal of the BST, visiting the left child first, then the current node, and finally the right child.

Since this is our first visit, we compare prev and root.val. They are different (prev is None at the start, and root.val is 1), so

As this is the first value we've seen, mx is also set to 1, and ans is updated to [1] because cnt equals mx at this point.

3. Up next, we backtrack and visit the root node which has a value 2. Now, prev is 1: o prev (1) does not match root.val (2), so we set cnt to 1.

space efficient.

Python Solution

class TreeNode:

10

13

14

15

16

17

20

21

22

23

24

26

27

28

29

30

35

36

37

38

39

40

41

43

44

45

46

48

49

50

51

52

53

54

55

56

57

58

59

60

61

62

63

64

66

65 }

C++ Solution

struct TreeNode {

int val;

13 class Solution {

/**

TreeNode *left;

TreeNode *right;

TreeNode* previous;

vector<int> modes;

/**

*/

6

8

9

11 };

14 public:

10

12

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

inorderTraversal(node.left);

modes.add(node.val);

// Traverse the right subtree

inorderTraversal(node.right);

* Definition for a binary tree node.

int maxFrequency, currentCount;

previous = nullptr;

inOrderTraversal(root);

maxFrequency = 0;

currentCount = 0;

modes.clear();

return modes;

* @param root The root node of the BST.

vector<int> findMode(TreeNode* root) {

TreeNode() : val(0), left(nullptr), right(nullptr) {}

TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

* @return A vector containing the mode(s) of the BST.

previousNode = node;

if (currentCount > maxFrequency) {

maxFrequency = currentCount;

else if (currentCount == maxFrequency) {

// Update the current count of the current value

modes = new ArrayList<>(Arrays.asList(node.val));

1 # Definition for a binary tree node.

we set cnt to 1.

We update prev to 2.

2. Begin the traversal. The dfs function runs:

The dfs function visits the leftmost node which is 1.

We update prev to 1 now that we have processed this node.

Currently, cnt does not exceed mx, so ans remains the same.

This time, prev matches root.val, so we increment cnt to 2.

In this example, the output is [2], which is the correct mode of the given BST.

4. Finally, we visit the right child which also has a value of 2:

 Now cnt (2) is greater than mx (1), so we have a new mode. Therefore, we update mx to 2, and reset ans to [2], the new mode.

5. The traversal is now complete, and since there are no more nodes to visit, ans contains the mode(s).

This walkthrough demonstrates the effectiveness of the algorithm as it leverages the BST property to perform an in-order traversal and identify the most frequently occurring element with the need for any additional data structures for frequency counting, making it

def __init__(self, val=0, left=None, right=None):

def in_order_traversal(node):

Traverse the left subtree

in_order_traversal(node.left)

if current_count > max_count:

max_count = current_count

elif current_count == max_count:

Traverse the right subtree

Initialize variables

in_order_traversal(node.right)

modes = [node.val]

if node is None:

return

Helper function to perform in-order traversal

Update the count for the current value

If the current count matches the max_count,

previous_value = None # Stores the previously seen value

add the current value to the list of modes

nonlocal max_count, previous_value, modes, current_count

If the current count exceeds the max_count found so far,

update max_count and reset modes with the current value

Otherwise, reset the count to 1 for the new value

If the current value matches the previous value, increment the count;

current_count = current_count + 1 if previous_value == node.val else 1

max_count = current_count = 0 # max_count is the maximum frequency of any value,

self.val = val self.left = left self.right = right class Solution: def findMode(self, root: TreeNode) -> List[int]:

modes.append(node.val) 31 32 33 # Update the previous_value to the current node's value 34 previous_value = node.val

```
42
                                           # current_count is the frequency of the current node value
43
           modes = [] # Stores the modes
44
           # Perform the in-order traversal starting from the root
45
            in_order_traversal(root)
46
47
           # Return the list of modes
48
49
           return modes
50
Java Solution
 1 import java.util.ArrayList;
 2 import java.util.Arrays;
   import java.util.List;
   // Definition for a binary tree node.
   class TreeNode {
       int val;
       TreeNode left;
       TreeNode right;
       TreeNode() {}
10
11
       TreeNode(int val) { this.val = val; }
12
       TreeNode(int val, TreeNode left, TreeNode right) {
13
           this.val = val;
           this.left = left;
14
           this.right = right;
15
16
17 }
18
   class Solution {
20
       private int maxFrequency;
       private int currentCount;
21
       private TreeNode previousNode;
23
       private List<Integer> modes;
24
25
       // Finds the mode(s) in a binary search tree.
26
       public int[] findMode(TreeNode root) {
            modes = new ArrayList<>();
27
28
            inorderTraversal(root);
            int[] result = new int[modes.size()];
29
30
            for (int i = 0; i < modes.size(); ++i) {</pre>
                result[i] = modes.get(i);
31
32
33
           return result;
34
35
36
       // Helper function to perform inorder traversal of the binary tree.
37
       private void inorderTraversal(TreeNode node) {
38
           if (node == null) {
39
                return;
41
42
            // Traverse the left subtree
```

// If it's the same as the previous node's value, increment the count; otherwise, reset it to 1

currentCount = (previousNode != null && previousNode.val == node.val) ? currentCount + 1 : 1;

// If current count is greater than max frequency, update max frequency and reset modes list

// If current count is equal to max frequency, add the current value to modes list

TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}

* Finds the mode(s) in a binary search tree (BST), which are the values with the highest frequency of occurrence.

// Update previousNode to the current node before traversing right subtree

42 43 44 45

```
34
 35
 36
         /**
 37
          * In-order depth-first search (DFS) traversal of the BST to find the mode(s).
 38
 39
          * @param node The current node being visited.
 40
         void inOrderTraversal(TreeNode* node) {
 41
             if (!node) return; // Base case: if the current node is null, do nothing.
             // Traverse the left subtree.
             inOrderTraversal(node->left);
 46
 47
             // Process the current node.
 48
             if (previous != nullptr && previous->val == node->val) {
 49
                 // If the value of the current node is the same as the previous node's value, increment the count.
 50
                 currentCount++;
 51
             } else {
                 // Otherwise, reset the count to 1.
                 currentCount = 1;
 54
 55
             if (currentCount > maxFrequency) {
 56
                 // If the current count exceeds the max frequency found so far,
 57
 58
                 // clear the current modes and add the new mode.
 59
                 modes.clear();
                 modes.push_back(node->val);
 60
 61
                 maxFrequency = currentCount;
 62
             } else if (currentCount == maxFrequency) {
 63
                 // If the current count equals the max frequency, add this node's value to the modes.
                 modes.push_back(node->val);
 64
 65
 66
 67
             // Update the previous node to the current one.
 68
             previous = node;
 69
             // Traverse the right subtree.
 70
 71
             inOrderTraversal(node->right);
 72
 73
    };
 74
Typescript Solution
  1 /**
     * Definition for a binary tree node.
     */
    class TreeNode {
         val: number;
         left: TreeNode | null;
         right: TreeNode | null;
  8
         constructor(val: number, left?: TreeNode, right?: TreeNode) {
 10
             this.val = val === undefined ? 0 : val;
 11
             this.left = left === undefined ? null : left;
             this.right = right === undefined ? null : right;
 13
 14 }
 15
 16 let previous: TreeNode | null = null;
 17 let maxFrequency: number = 0;
 18 let currentCount: number = 0;
    let modes: number[] = [];
 20
 21 /**
     * Finds the modes in a binary search tree (BST), which are the values with the highest frequency of occurrence.
     * @param root The root node of the BST.
     * @returns An array containing the modes of the BST.
 25
    function findMode(root: TreeNode | null): number[] {
         previous = null;
 27
        maxFrequency = 0;
 28
 29
        currentCount = 0;
        modes = [];
 30
 31
 32
         inOrderTraversal(root);
 33
 34
         return modes;
 35 }
 36
     * Performs an in-order traversal of the BST to find the modes.
     * @param node The current node in the traversal.
 40
     */
     function inOrderTraversal(node: TreeNode | null): void {
         if (!node) return; // Base case: If the current node is null, return.
 42
 43
 44
        // Traverse the left subtree.
```

68 69 // Traverse the right subtree. 70 inOrderTraversal(node.right); 71 } 72

Time and Space Complexity

previous = node;

inOrderTraversal(node.left);

// Process the current node.

currentCount++;

currentCount = 1;

modes = [node.val];

modes.push(node.val);

if (currentCount > maxFrequency) {

maxFrequency = currentCount;

} else {

if (previous && node.val === previous.val) {

// Otherwise, reset the count to 1.

} else if (currentCount === maxFrequency) {

// clear the modes array and add the new mode.

45

46

47

48

49

50

51

54

55

56

57

58

59

60

61

62

63

64

65

66

67

The time complexity of the provided code is O(N), where N is the number of nodes in the binary tree. This is because the in-order traversal performed by the dfs function visits each node exactly once. The space complexity of the code is O(H), where H is the height of the binary tree, due to the call stack during the recursive

// If the current node value is the same as the previous node's value, increment the count.

// If the current count is equal to the max frequency, add this node's value to the modes.

// If the current count is greater than the max frequency found so far,

// Update the previous node to the current node for the next iteration.

traversals. In the worst-case scenario of a skewed tree, the height H can be N, leading to a space complexity of O(N). Additionally, the space required to store the modes in ans does not exceed O(N) (in the case where all elements are the same and hence the mode list contains all elements), but it does not affect the worst-case space complexity which is dominated by the recursive stack.