1902. Depth of BST Given Insertion Order Medium Tree **Binary Search Tree Binary Tree Ordered Set** 

## **Problem Description** You are given an array order which is a permutation of integers from 1 to n. This array represents the sequence in which nodes are

inserted into a binary search tree (BST). The properties of a BST are such that: • The left subtree of a node contains only nodes with keys less than the node's key.

**Leetcode Link** 

- Both the left and right subtrees must also be binary search trees.
- The first element of order is the root of the BST. All subsequent elements are inserted one by one, at the correct position to maintain

• The right subtree of a node contains only nodes with keys greater than the node's key.

the BST property. You need to find the depth of this binary search tree, which is the number of nodes along the longest path from the root node down to the farthest leaf node.

the BST. To achieve this efficiently, the solution uses a SortedDict structure, which is a Python data structure that maintains the

keys in sorted order and allows binary search operations.

Intuition The intuition behind the solution is to track the depth of the tree while iterating through the order array and inserting elements into

Here's how we use SortedDict to solve the problem: • Initialize the SortedDict with the first element of the order array as the root with a depth of 1 and two sentinel values 0 and

infinity with depths 0. These sentinel values help in finding the lower and higher elements (parent candidates in BST) for the new element to be inserted. As elements from the order array are processed, determine the depth of the new node by finding the elements just less than

• The depth of the current element will be one more than the maximum depth of the lower or higher neighbors, since it will be the child of one of these neighbors. Update the maximum depth (ans) if the depth of the current node is greater.

(lower) and just greater than (higher) the current element in the sorted keys of the SortedDict.

- Insert the current element with its computed depth into the SortedDict. Repeat this process for each element in the order array.
- After processing all elements, ans will contain the depth of the binary search tree. This efficient approach avoids having to build the actual BST and traverse it to find the depth, and instead calculates the depth on-
- the-fly during insertion. Solution Approach
- To understand the implementation of the solution, let's go through the key components of the code:
- **Data Structure Used** SortedDict: A SortedDict is a data structure in the sortedcontainers library of Python that keeps the keys sorted and allows for

1. Initialization: We initiate the SortedDict with a mapping from 0 to depth 0, infinity to depth 0, and the first element of the

order array (which is the root node) to depth 1. Both 0 and infinity are sentinel values that assist in determining the depth of

2. Iterate through order: After the root node is added, we iterate through the remaining numbers in order array. For each number

## 1 sd = SortedDict({0: 0, inf: 0, order[0]: 1})

Steps of the Algorithm

the subsequent nodes.

(node) v, we want to find where it fits in the tree and calculate its depth.

1 for v in order[1:]: # Skip the first element as it is the root

binary search. In this scenario, it's being used to keep track of the depth at each inserted node.

# 3. Binary Search for Depth: For each number, we perform a binary search (bisect\_left(v)) on the SortedDict keys to find the

lower = sd.bisect\_left(v) - 1

ans = max(ans, depth)

2 sd[v] = depth

BST.

2 higher = lower + 14. Calculate the Depth of the Node: The depth of the new node we are inserting is one more than the maximum depth between its potential parents (lower and higher), this is because in the BST the new node will become the child of one of these nodes.

5. Update the SortedDict and ans: Insert the new node v with its calculated depth into the SortedDict. Also, update ans with the

6. Return the Maximum Depth: Once we have inserted all the elements, ans will hold the maximum depth of the tree, which we

In this implementation, SortedDict plays a vital role by keeping track of nodes in sorted order and their associated depth, using

binary search for neighbor lookup which results in efficient on-the-fly depth computation of the BST while simulating the insertion

closest smaller value (the predecessor or lower) and the direct successor (higher). These are the two possible parents in the

```
then return.
1 return ans
```

represents the sequence in which nodes will be inserted into the BST.

maximum depth of these two nodes, which is 1 + 1 = 2.

2 ans = 1 # The depth of the tree

1 sd = SortedDict({0: 0, 2: 2, 3: 1, inf: 0})

2. Process Second Element (2):

Update our SortedDict:

Update our SortedDict:

4. Process Fourth Element (1):

5. Process Fifth Element (4):

2 ans = 3

of this BST.

**Python Solution** 

class Solution:

9

10

11

13

14

15

16

17

19

20

21

22

24

26

27

28

29

30

9

16

18

19

20

21

22

23

24

25

26

27

28

30

17

18

19

20

23

24

25

26

17

18

19

20

21

22

23

24

25

26

27

28

29

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

from typing import List

1 from sortedcontainers import SortedDict

return max\_depth

Java Solution

2 ans = 2

1 depth = 1 + max(sd.values()[lower], sd.values()[higher])

new depth if it is greater than the current maximum depth found.

each of the n elements. Example Walkthrough

To illustrate the solution approach, let's use a small example with an order array. Suppose our order array is [3, 2, 5, 1, 4]. This

The time complexity of this approach is  $0(n \log n)$ , where n is the number of elements in the order array. This is because, for each

insertion into the SortedDict, which internally maintains a sorted order, the operations perform in O(log n) time, and we do this for

```
1. Initialization: We start with our SortedDict initialized as follows:
   1 sd = SortedDict({0: 0, inf: 0, 3: 1}) # Root node '3' with depth '1'
```

process.

**Time Complexity** 

3. Process Third Element (5):

∘ The depth of '4' is calculated as 1 + 2 = 3 (since both '3' and '5' have a depth of '1' and '2' respectively, and we take the

6. Return the Maximum Depth: After processing all elements, the maximum depth ans is already updated to 3, which is the depth

Following this step-by-step illustration, it's clear how the SortedDict and the algorithm work together to insert nodes and compute

Since the depth of '0' (a sentry) is '0', and the depth of '3' (root node) is '1', the depth of '2' will be one more than the

```
ans remains 2 as we did not find a deeper depth.
```

1 sd = SortedDict({0: 0, 1: 3, 2: 2, 3: 1, 5: 2, inf: 0})

Insert '1'. The closest smaller is '0', and the closest larger is '2'.

1 sd = SortedDict({0: 0, 2: 2, 3: 1, 5: 2, inf: 0})

 $\circ$  The depth of '5' becomes 1 + 1 = 2 (since the depth of '3' is '1').

```
\circ The depth of '1' will then be 1 + 2 = 3 (since the depth of '2' is '2').
Update our SortedDict:
```

Insert '4' into the BST. The closest smaller value is '3', and the closest larger is '5'.

∘ Insert '2' into the BST. The closest smaller value to '2' is '0', and the closest larger value is '3'.

o Insert '5' into the BST. The closest smaller value to '5' is '3', and the closest larger is inf.

```
max).
Update our SortedDict:
```

1 sd = SortedDict({0: 0, 1: 3, 2: 2, 3: 1, 4: 3, 5: 2, inf: 0})

the depth of the BST efficiently without constructing the actual tree.

def maxDepthBST(self, order: List[int]) -> int:

higher\_index = lower\_index + 1

// Dummy nodes to handle edge cases

depthMap.put(order[0], 1);

# the root of BST with depth 1

ans remains 3 since the depth of '4' matches the current deepest level.

Update ans since we found a deeper depth.

# Initialize answer with the depth of the first element (root), which is 1  $max_depth = 1$ # Iterate over the remaining elements in the order list starting from the second for value in order[1:]:

# Find the keys that would be immediate predecessors and successors of the value

# The new depth is calculated by 1 plus the max depth of immediate lower and higher keys

depth = 1 + max(sorted\_dict.peekitem(lower\_index)[1], sorted\_dict.peekitem(higher\_index)[1])

# Create a SortedDict initialized with border elements with depth 0 and

sorted\_dict = SortedDict({0: 0, float('inf'): 0, order[0]: 1})

lower\_index = sorted\_dict.bisect\_left(value) - 1

# Finally, return the maximum depth of the binary search tree

depthMap.put(0, 0); // Represents the leftmost boundary

// Initial case: the root node has a depth of 1

// Update the maximum depth found so far

depthMap.put(currentValue, currentDepth);

// Return the maximum depth found

int maxDepthBST(std::vector<int>& order) {

// Dummy nodes to handle edge cases

std::map<int, int> depthMap;

return maxDepth;

maxDepth = Math.max(maxDepth, currentDepth);

depthMap.put(Integer.MAX\_VALUE, 0); // Represents the rightmost boundary

// Find the immediate lower and higher entries in the TreeMap

// Insert the current value and its depth into the TreeMap

// TreeMap to store the nodes with their corresponding depths

// Iterate over the remaining nodes in the "order" vector

// Find the immediate lower and higher entries in the map

int currentDepth = 1 + std::max(lowerEntry->second, higherEntry->second);

auto lowerEntry = --depthMap.lower\_bound(currentValue);

auto higherEntry = depthMap.upper\_bound(currentValue);

// Global method to calculate the maximum depth of the binary search tree.

// The starting maximum depth is the depth of the root node, which is 1.

const currentDepth: number = 1 + Math.max(lowerDepth, higherDepth);

// Insert the current value and its depth into the TreeMap-like structure.

depthMap.set(Number.MAX\_SAFE\_INTEGER, 0); // Represents the right-most boundary.

// Fetch the immediate lower and higher entries in the TreeMap-like structure.

const lowerEntry: [number, number] | undefined = depthMap.lowerEntry(currentValue);

const higherEntry: [number, number] | undefined = depthMap.higherEntry(currentValue);

// Compute the depth of the current node as one plus the maximum of the depths of lower and higher entries.

// Iterate over the remaining nodes in the "order" array starting from the second element.

depthMap.set(0, 0); // Represents the left-most boundary.

// If either entry is undefined, use zero as their depth.

const lowerDepth: number = lowerEntry?.[1] ?? 0;

// Update the maximum depth found so far.

depthMap.set(currentValue, currentDepth);

maxDepth = Math.max(maxDepth, currentDepth);

const higherDepth: number = higherEntry?.[1] ?? 0;

function maxDepthBST(order: number[]): number {

// Insert dummy nodes to handle edge cases.

// The root node has an initial depth of 1.

for (let i = 1; i < order.length; ++i) {</pre>

let currentValue: number = order[i];

depthMap.set(order[0], 1);

let maxDepth: number = 1;

for (size\_t i = 1; i < order.size(); ++i) {</pre>

// Update the maximum depth found so far

maxDepth = std::max(maxDepth, currentDepth);

int currentValue = order[i];

Map.Entry<Integer, Integer> lowerEntry = depthMap.lowerEntry(currentValue);

Map.Entry<Integer, Integer> higherEntry = depthMap.higherEntry(currentValue);

int currentDepth = 1 + Math.max(lowerEntry.getValue(), higherEntry.getValue());

- # Update the answer if the new calculated depth is greater than the previous max depth max\_depth = max(max\_depth, depth) # Insert the current value with its calculated depth into the sorted dictionary sorted\_dict[value] = depth
- class Solution { public int maxDepthBST(int[] order) { // TreeMap to store the nodes with their corresponding depths TreeMap<Integer, Integer> depthMap = new TreeMap<>();
- // The starting maximum depth is the depth of the root, which is 1 int maxDepth = 1; 11 12 13 // Iterate over the remaining nodes in the "order" array for (int i = 1; i < order.length; ++i) {</pre> 14 int currentValue = order[i]; 15

// Compute the depth of the current node as one plus the maximum of the depths of lower and higher entry

### depthMap[0] = 0; // Represents the leftmost boundary 11 12 depthMap[INT\_MAX] = 0; // Represents the rightmost boundary 13 // Initial case: the root node has a depth of 1 depthMap[order[0]] = 1; 14 // The starting maximum depth is the depth of the root, which is 1 int maxDepth = 1; 16

C++ Solution

#include <map>

2 #include <vector>

class Solution {

public:

#include <algorithm>

```
28
               // Insert the current value and its depth into the map
29
               depthMap[currentValue] = currentDepth;
30
           // Return the maximum depth found
31
           return maxDepth;
34 };
35
Typescript Solution
  1 // TreeMap-like structure using a SortedMap interface from external libraries could be used.
  2 // For the purpose of this question, assuming such an interface exists in the environment.
  3 // If not, one would need to implement it or use a workaround.
    interface SortedMap<K, V> {
       get(key: K): V | undefined;
       set(key: K, value: V): void;
       lowerEntry(key: K): [K, V] | undefined;
       higherEntry(key: K): [K, V] | undefined;
  9
 10
 11 // Initialize a TreeMap-like structure to store nodes with their corresponding depths.
    // Assume that an appropriate SortedMap implementation or a suitable external module is available.
    const depthMap: SortedMap<number, number> = new SortedMapImpl();
 14
```

// Compute the depth of the current node as one plus the maximum of the depths of lower and higher entry

### 47 // Return the maximum depth found. 48 return maxDepth; 49 50

Time and Space Complexity

### elements in order. • Within the loop, the bisect\_left operation performs a binary search which is 0(log k) where k is the number of elements in the SortedDict at that point.

**Time Complexity** 

• The indexing operations to access sd.values()[lower] and sd.values()[higher] are 0(1) as SortedDict maintains a list-like values view. The max and bisect\_left operations within the loop are also 0(log k).

The time complexity of the maxDepthBST function can be analyzed as follows:

Considering the loop iteration and the operations that occur within each iteration, the total time complexity is  $0((n - 1) * \log k)$ . Since k can be at most n as the loop progresses, this simplifies to  $O(n \log n)$ .

• The update operation sd[v] = depth is O(log k) since it might require rebalancing the tree structure of the SortedDict.

• The construction of the initial SortedDict is O(log n) since it involves inserting the first element of order into the data structure.

• The main for loop runs for every element in the order list except the first one, which is n - 1 iterations where n is the number of

- **Space Complexity** The space complexity can be analyzed as follows:
- The SortedDict can hold up to n + 2 elements (including the added 0 and inf keys). The depth and ans variables use a constant amount of space. Therefore, the space complexity is O(n) due to the space required to store the elements in the SortedDict.

In conclusion, the time complexity of the code is  $O(n \log n)$  and the space complexity is O(n).