1902. Depth of BST Given Insertion Order

Medium Tree Binary Search Tree Binary Tree Ordered Set

Leetcode Link

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

Intuition

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

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 (lower) and just greater than (higher) the current element in the sorted keys of the `SortedDict`.
- 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.
- 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 binary search. In this scenario, it's being used to keep track of the depth at each inserted node.

Steps of the Algorithm

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 the subsequent nodes.

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

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

3. **Binary Search for Depth**: For each number, we perform a binary search (`bisect_left(v)`) on the `SortedDict` keys to find the closest smaller value (the predecessor or `lower`) and the direct successor (`higher`). These are the two possible parents in the BST.

```
1 lower = sd.bisect_left(v) - 1
2 higher = lower + 1
```

4. **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.

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

5. **Update the `SortedDict` and `ans`**: Insert the new node `v` with its calculated depth into the `SortedDict`. Also, update `ans` with the new depth if it is greater than the current maximum depth found.

```
1 ans = max(ans, depth)
2 sd[v] = depth
```

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

```
1 return ans
```

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 process.

Time Complexity

The time complexity of this approach is $O(n \log n)$, 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 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 represents the sequence in which nodes will be inserted into the BST.

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'
2 ans = 1  # The depth of the tree
```

2. **Process Second Element (`2`)**:
   - Insert '2' into the BST. The closest smaller value to '2' is '0', and the closest larger value is '3'.
   - 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 maximum depth of these two nodes, which is `1 + 1 = 2`.

   Update our `SortedDict`:

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

3. **Process Third Element (`5`)**:
   - Insert '5' into the BST. The closest smaller value to '5' is '3', and the closest larger is `inf`.
   - The depth of '5' becomes `1 + 1 = 2` (since the depth of '3' is '1').

   Update our `SortedDict`:

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

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

4. **Process Fourth Element (`1`)**:
   - Insert '1'. The closest smaller is '0', and the closest larger is '2'.
   - The depth of '1' will then be `1 + 2 = 3` (since the depth of '2' is '2').

   Update our `SortedDict`:

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

   Update `ans` since we found a deeper depth.

5. **Process Fifth Element (`4`)**:
   - Insert '4' into the BST. The closest smaller value is '3', and the closest larger is '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 max).

   Update our `SortedDict`:

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

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

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

Following this step-by-step illustration, it's clear how the `SortedDict` and the algorithm work together to insert nodes and compute the depth of the BST efficiently without constructing the actual tree.

Python Solution

```
from sortedcontainers import SortedDict
from typing import List

class Solution:
    def maxDepthBST(self, order: List[int]) -> int:
        # Create a SortedDict initialized with border elements with depth 0 and
        # the root of BST with depth 1
        sorted_dict = SortedDict({0: 0, float('inf'): 0, order[0]: 1})

        # 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 index in range(1, len(order)):
            # Find the keys that would be immediate predecessors and successors of the value
            lower_index = sorted_dict.bisect_left(value) - 1
            higher_index = lower_index + 1

            # The new depth is calculated by 1 plus the max depth of immediate lower and higher keys
            depth = 1 + max(sorted_dict.values()[lower_index], sorted_dict.peekitem(higher_index)[1])

            # 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

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

Java Solution

```
class Solution {
    public int maxDepthBST(int[] order) {
        // TreeMap to store the nodes with their corresponding depths
        TreeMap<Integer, Integer> depthMap = new TreeMap<>();
        // Initialize the TreeMap with boundary values
        depthMap.put(0, 0); // Represents the leftmost boundary
        depthMap.put(INC_MAX + 0, 0); // Represents the rightmost boundary
        // Initial depth of the root node has a depth of 1
        depthMap.put(order[0], 1);
        // The starting maximum depth is the depth of the root, which is 1
        int maxDepth = 1;

        // Iterate over the remaining nodes in the "order" array
        for (int i = 1; i < order.length; ++i) {
            int currentValue = order[i];
            // Find the immediate lower and higher entries in the TreeMap
            int lowerEntry = depthMap.lowerEntry(currentValue).getKey();
            int higherEntry = depthMap.higherEntry(currentValue).getKey();
            // Compute the depth of the current node as one plus the maximum of the depths of lower and higher entry
            int currentDepth = 1 + Math.max(lowerEntry.getValue(), higherEntry.getValue());
            // Update the maximum depth found so far
            maxDepth = Math.max(maxDepth, currentDepth);
            // Insert the current value and its depth into the TreeMap
            depthMap.put(currentValue, currentDepth);
        }
        // Return the maximum depth found
        return maxDepth;
    }
}
```

C++ Solution

```
#include <map>
#include <vector>
#include <algorithm>

class Solution {
public:
    int maxDepthBST(std::vector<int>& order) {
        // TreeMap to store the nodes with their corresponding depths
        std::map<int, int> depthMap;
        // Dummy nodes to handle edge cases
        depthMap[0] = 0; // Represents the leftmost boundary
        depthMap[INC_MAX] = 0; // Represents the rightmost boundary
        // Initial depth of the root node has a depth of 1
        depthMap[order[0]] = 1;
        // The starting maximum depth is the depth of the root, which is 1
        int maxDepth = 1;

        // Iterate over the remaining nodes in the "order" vector
        for (size_t i = 1; i < order.size(); ++i) {
            int currentValue = order[i];
            // Find the immediate lower and higher entries in the map
            int lowerEntry = std::prev(depthMap.lower_bound(currentValue))->first;
            auto higherEntry = depthMap.upper_bound(currentValue);
            // Compute the depth of the current node as one plus the maximum of the depths of lower and higher entry
            int currentDepth = 1 + std::max(depthMap[lowerEntry], higherEntry->second);
            // Update the maximum depth found so far
            maxDepth = std::max(maxDepth, currentDepth);
            // Insert the current value and its depth into the map
            depthMap[currentValue] = currentDepth;
        }
        // Return the maximum depth found
        return maxDepth;
    }
};
```

Typescript Solution

```
// TreeMap-like structure using a SortedMap interface from external libraries could be used.
// For the purpose of this question, assuming such an interface exists in the environment.
// If not, one would need to implement it or use a workaround.
interface SortedMap<K, V> {
    set(key: K): V | undefined;
    set(key: K, value: V): void;
    lowerEntry(key: K): [K, V] | undefined;
    higherEntry(key: K): [K, V] | undefined;
}

// Initialize a TreeMap-like structure to store nodes with their corresponding depths.
// Assuming 'SortedMap' is a globally available interface for a suitable external module is available.
const depthMap: SortedMap<number, number> = new SortedMap<number>();

// Global method to calculate the maximum depth of the binary search tree.
function maxDepthBST(order: number[]): number {
    // Insert dummy nodes to handle edge cases.
    depthMap.set(0, 0); // Represents the left-most boundary.
    depthMap.set(MAX_SAFE_INTEGER, 0); // Represents the right-most boundary.

    // The root node has an initial depth of 1.
    depthMap.set(order[0], 1);
    // The starting maximum depth is the depth of the root node, which is 1.
    let maxDepth: number = 1;

    // Iterate over the remaining nodes in the "order" array starting from the second element.
    for (let i = 1; i < order.length; ++i) {
        const currentValue: number = order[i];
        // Find the immediate lower and higher entries in the TreeMap-like structure.
        const lowerEntry: number | undefined = depthMap.lowerEntry(currentValue)?.[0];
        const higherEntry: number | undefined = depthMap.higherEntry(currentValue)?.[0];

        // If either entry is undefined, use zero as their depth.
        const lowerDepth: number = lowerEntry? depthMap.get(lowerEntry) : 0;
        const higherDepth: number = higherEntry? depthMap.get(higherEntry) : 0;

        // Compute the depth of the current node as one plus the maximum of the depths of lower and higher entries.
        const currentDepth: number = 1 + Math.max(lowerDepth, higherDepth);

        // Update the maximum depth found so far.
        maxDepth = Math.max(maxDepth, currentDepth);

        // Insert the current value and its depth into the TreeMap-like structure.
        depthMap.set(currentValue, currentDepth);
    }

    // Return the maximum depth found.
    return maxDepth;
}
```

Time and Space Complexity

Time Complexity

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

- 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 elements in `order`.
- Within the loop, the `bisect_left` operation performs a binary search which is $O(\log k)$ where k is the number of elements in the `SortedDict` at that point.
- The indexing operations to access `sd.values()[lower]` and `sd.values()[higher]` are $O(1)$ as `SortedDict` maintains a list-like values view.
- The `max` and `bisect_left` operations within the loop are also $O(\log k)$.
- The update operation `sd[v] = depth` is $O(\log k)$ since it might require rebalancing the tree structure of the `SortedDict`.

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

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 $O(n)$.