108. Convert Sorted Array to Binary Search Tree

Easy | Tree | Binary Search Tree | Array | Divide and Conquer | Binary Tree

Problem Description

The problem gives us an integer array called `nums` which is sorted in ascending order. Our task is to convert this sorted array into a height-balanced binary search tree (BST). A height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differs by more than one.

Intuition

The solution to the problem lies in the properties of a binary search tree and the characteristics of a sorted array. A binary search tree is a node-based binary tree where each node has the following properties:

- 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.
- The left and right subtrees must also be binary search trees.

The challenge is to ensure that the BST is height-balanced. To achieve this, we need to pick the middle element from the sorted array and make it a root, so that the elements to the left, which are lesser, form the left subtree, and the elements to the right, which are greater, form the right subtree.

To construct the BST, we apply a recursive strategy:

1. Find the middle element of the current segment of the array.
2. Make this element the root of the current subtree.
3. Recursively perform the same action on the left half of the array to create the left subtree.
4. Recursively perform the same action on the right half of the array to create the right subtree.

This divide and conquer approach will ensure that the tree remains height-balanced, as each subtree is constructed from segments of the array that are roughly half the size of the original segment.

By carefully selecting the middle element of the array as the root for each recursive step, the resulting tree satisfies the conditions of a binary search tree as well as being height-balanced.

Solution Approach

To implement the solution using the given Python code, the following steps are taken, which relate to the concepts of depth-first search (DFS) and recursion:

1. A recursive helper function called `dfs` is defined, which takes two parameters, `l` and `r`, which represent the left and right bounds of the segment of the `nums` array that we are currently considering. This function is responsible for creating nodes in the BST.

2. The base case for the recursive function is checked: if `l > r`, it means we have considered all elements in this segment, and there is nothing to create a node with. Therefore, `None` is returned, indicating the absence of a subtree.

3. The middle index of the current segment is found using the expression `(l + r) >> 1`, which is equivalent to finding the average of `l` and `r` and then taking the floor of the result. The bitwise right shift operator `>>` is used here to efficiently divide the sum by 2.

4. The middle element of `nums`, located at index `mid`, is the value of the root node for the current segment of the array. A `TreeNode` is created with this value.

5. To build a binary search tree, we need to recursively build the left and right subtrees. The recursive `dfs` call for the left subtree uses the bounds `l` to `mid - 1`, while the recursive `dfs` call for the right subtree uses the bounds `mid + 1` to `r`.

6. After the left and right subtrees are created via recursion, a new `TreeNode` with `nums[mid]` as its value and the created left and right subtrees as its children is returned.

7. The outer function `sortedArrayToBST` initializes the construction process by calling `dfs(0, len(nums) - 1)` with the full range of the input array `nums` as the bounds for the entire tree.

This approach efficiently builds the binary search tree by always dividing the array into two halves, ensuring the tree is balanced. The tree's properties as a binary search tree are maintained because we construct the left and right subtrees from elements that are strictly less than or greater than the root of any subtree, thus satisfying all the properties of the BST.

The choice of using a recursive function encapsulates the logic for both creating TreeNode instances and ensuring that we honor the bounds of the current array segment at each step, simplifying the complexity of the task at each recursive call.

Example Walkthrough

Let's use a small example to illustrate the solution approach. We are given the following sorted array:

```
nums = [-10, -3, 0, 5, 9]
```

Using the steps outlined in the solution approach, we would construct a height-balanced BST as follows:

1. We first call the `dfs` function with the initial bounds set to `l = 0` and `r = 4` (assuming zero-based array indexing).

2. We find the middle element of `nums` between indices 0 and 4. The midpoint calculation would be `(0 + 4) >> 1`, which simplifies to `2`. Hence, the middle element is `nums[2]`, which is `0`.

3. We create a `TreeNode` with a value of `0`. This is the root of our BST.

4. Now we need to build the left subtree. We make a recursive `dfs` call with the new bounds, `l = 0` and `r = 1` (`mid - 1`), focusing on the subarray `[-10, -3]`.
   - For this subarray, we take the middle element as the root for the left subtree. The midpoint is `(0 + 1) >> 1`, simplifying to `0`. The element at that index in `nums` is `-10`, so we create a `TreeNode` with value `-10`.
   - For the left child of `-10`, the `dfs` call with `l = 0` and `r = -1` returns `None` since `l > r`.
   - For the right child, we have `nums[mid + 1]` where `mid` is `0`. We make a recursive call with `l = 1` and `r = 1`. `mid` for this subarray is `1` and the element at that index is `-3`, so `-3` becomes the right child node of `-10`.
5. Returning to our root `0`, we build the right subtree with a new `dfs` call using the bounds `l = 3` (`mid + 1`) and `r = 4`, focusing on the subarray `[5, 9]`.
   - The midpoint here is `(3 + 4) >> 1`, which is `3`. The element at index `3` in `nums` is `5`. So, `5` becomes the left child node of the right subtree of `0`.
   - The left child of `5` would be `None` since a recursive call with bounds `l = 3` and `r = 2` returns `None`.
   - For the right child, the new midpoint is `4` (`mid + 1` where `mid = 3`), leading to the element `9` being selected. Hence, `9` is the right child node of `5`.
6. At this point, we have successfully constructed all parts of the BST, which would look like this:

```
        0
       / \
     -3   5
     /     \
  -10       9
```

By following these steps, we have successfully transformed the sorted array `nums` into a height-balanced BST. The recursive process ensures that each subtree is constructed from the middle of the subarray it refers to, making the tree balanced and maintaining the binary search tree property.

Python Solution

```python
class TreeNode:
    # Definition for a binary tree node.
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class Solution:
    def sortedArrayToBST(self, nums):
        """
        Converts a sorted array into a height-balanced binary search tree (BST).
        A height-balanced binary tree is a binary tree in which the depth of the two
        subtrees of every node never differ by more than one.

        Args:
        nums : List[int]
            An integer array which is sorted in non-decreasing order.

        Returns:
        TreeNode
            The root of the otherwise height-balanced binary search tree.
        """

        def convert_to_bst(left_index, right_index):
            """
            Recursively construct BST from nums array using the divide and conquer approach.

            Args:
            left_index : int
                The starting index of the subarray to be processed.
            right_index : int
                The ending index of the subarray to be processed.

            Returns:
            TreeNode
                The constructed BST node (root for the subarray).
            """
            # Base case: If the left index is greater than the right index,
            # the subarray is empty, and there is no tree to construct.
            if left_index > right_index:
                return None

            # Choosing the middle element of the subarray to be the root of the BST.
            middle_index = (left_index + right_index) // 2

            # Recursively constructing the left subtree.
            left_subtree = convert_to_bst(left_index, middle_index - 1)

            # Recursively constructing the right subtree.
            right_subtree = convert_to_bst(middle_index + 1, right_index)

            # Creating the root node with the middle element, passing the
            # left and right subtrees as its children.
            return TreeNode(nums[middle_index], left_subtree, right_subtree)

        # Initiating the recursive function with the whole array range.
        return convert_to_bst(0, len(nums) - 1)

```

Java Solution

```java
/**
 * Definition for a binary tree node.
 */
class TreeNode {
    int val; // Node value
    TreeNode left; // Left child
    TreeNode right; // Right child

    // Constructor
    TreeNode() {}

    // Constructor with value
    TreeNode(int val) { this.val = val; }

    // Constructor with value, left child, and right child
    TreeNode(int val, TreeNode left, TreeNode right) {
        this.val = val;
        this.left = left;
        this.right = right;
    }
}

class Solution {
    private int[] nums;

    /**
     * Convert a sorted array into a height-balanced binary search tree.
     *
     * @param nums Sorted array of integers
     * @return Root of the height-balanced binary search tree
     */
    public TreeNode sortedArrayToBST(int[] nums) {
        this.nums = nums;
        return constructBSTRecursive(0, nums.length - 1);
    }

    /**
     * Recursive helper method to construct BST from the sorted array.
     *
     * @param left  The starting index of the subarray
     * @param right The ending index of the subarray
     * @return The root of the BST subtree constructed from subarray
     */
    private TreeNode constructBSTRecursive(int left, int right) {
        // Base case: If left > right, the subarray is empty and should return null
        if (left > right) {
            return null;
        }

        // Find the middle element to maintain BST properties. Use `left + (right - left) / 2`
        // to avoid integer overflow
        int mid = left + (right - left) / 2;

        // Recursively construct the left subtree
        TreeNode leftSubtree = constructBSTRecursive(left, mid - 1);

        // Recursively construct the right subtree
        TreeNode rightSubtree = constructBSTRecursive(mid + 1, right);

        // Create a new TreeNode with the mid element and the previously constructed left and right subtrees
        TreeNode node = new TreeNode(nums[mid], leftSubtree, rightSubtree);

        return node;
    }
}

```

C++ Solution

```cpp
/**
 * Definition for a binary tree node.
 */
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;

    // Constructor for a tree node with a given value and optional left and right children.
    TreeNode(int x, TreeNode *left = nullptr, TreeNode *right = nullptr)
        : val(x), left(left), right(right) {}
};

class Solution {
public:
    /**
     * Converts a sorted array into a binary search tree with minimal height.
     * @param nums The vector of integers sorted in non-decreasing order.
     * @return A pointer to the root of the constructed binary search tree.
     */
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        // Recursive function that converts a subarray into a BST.
        function<TreeNode*(int, int)> convertToBST = [&](int leftIndex, int rightIndex) -> TreeNode* {
            // Base case: if left index is greater than right, the subarray is empty and returns nullptr.
            if (leftIndex > rightIndex) {
                return nullptr;
            }

            // Choose the middle element as the root to maintain minimal height.
            int mid = leftIndex + (rightIndex - leftIndex) / 2;

            // Recursively construct the left subtree.
            auto leftChild = convertToBST(leftIndex, mid - 1);

            // Recursively construct the right subtree.
            auto rightChild = convertToBST(mid + 1, rightIndex);

            // Create and return the current tree node with its left and right children.
            return new TreeNode(nums[mid], leftChild, rightChild);
        };

        // Start the recursive process with the full array range.
        return convertToBST(0, nums.size() - 1);
    }
};

```

Typescript Solution

```typescript
interface TreeNode {
    val: number;
    left: TreeNode | null;
    right: TreeNode | null;
}

/**
 * Converts a sorted array into a height-balanced binary search tree.
 * @param {number[]} nums - A sorted array of numbers.
 * @return {TreeNode | null} - The root node of the constructed BST, or null if the array is empty.
 */
function sortedArrayToBST(nums: number[]): TreeNode | null {
    // Determine the length of the array.
    const length = nums.length;

    // Base Case: If array is empty, return null.
    if (length === 0) {
        return null;
    }

    // Find the middle index of the array.
    const middleIndex = Math.floor(length / 2);

    // Recursively construct the BST by choosing the middle element of the
    // current subsection as the root, and the left and right subsections as the left and right subtrees.
    const rootNode = {
        val: nums[middleIndex],
        left: sortedArrayToBST(nums.slice(0, middleIndex)),
        right: sortedArrayToBST(nums.slice(middleIndex + 1))
    };

    // Return the root node of the BST.
    return rootNode;
}

// Note: No class has been defined as per the instructions.
// TreeNode and sortedArrayToBST are both defined in the global scope.

```

Time and Space Complexity

The time complexity of the given code is `O(n)`, where `n` is the number of elements in the input list `nums`. This is because each element of the array is visited exactly once to construct each node of the resulting binary search tree (BST).

The space complexity of the code is also `O(n)` if we consider the space required for the output BST. However, if we only consider the auxiliary space (excluding the space taken by the output), it is `O(log n)` in the best case (which is the height of a balanced BST). The worst-case space complexity would be `O(n)` if the binary tree is degenerated (i.e., every node only has one child) which would be the case if the input list is sorted in strictly increasing or decreasing order.