Binary Search Tree

1. Lowest Common Ancestor of a Binary Search Tree

Difficulty: Easy
Time: 20 mins

Problem Statement:

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST. According to the definition of LCA on a BST, the LCA is the node that satisfies the following conditions: 1. The LCA node is located in the path between the two nodes `p` and `q`. 2. The node is a descendant of both `p` and `q`.

Sample Input:

```
root = [6,2,8,0,4,7,9,null,null,3,5]
p = 2
q = 8
```

Sample Output:

```
6
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def lowestCommonAncestor(root: TreeNode, p: TreeNode, q: TreeNode)-> TreeNode:
    # Start from the root node of the BST
```

```
    current_node = root

    while current_node:
        # If both nodes p and q are in the right subtree
        # of the current node, go right
        if p.val > current_node.val and q.val > current_node.val:
            current_node = current_node.right
        # If both nodes p and q are in the left subtree
        # of the current node, go left
        elif p.val < current_node.val and q.val < current_node.val:
            current_node = current_node.left
        else:
            # We have found the split point, i.e. the LCA node.
            return current_node

# Sample Test Case
# Construct a simple BST
root = TreeNode(6)
root.left = TreeNode(2)
root.right = TreeNode(8)
root.left.left = TreeNode(0)
root.left.right = TreeNode(4)
root.right.left = TreeNode(7)
root.right.right = TreeNode(9)
root.left.right.left = TreeNode(3)
root.left.right.right = TreeNode(5)

p = root.left # Node with value 2
q = root.right # Node with value 8
# Output: LCA of 2 and 8 is 6
print(f"LCA of {p.val} and {q.val} is {lowestCommonAncestor(root, p, q).val}")
```

```
LCA of 2 and 8 is 6
```

Time Complexity: O(h), where h is the height of the tree, as we traverse from the root to a leaf or the split point.
Space Complexity: O(1), as no additional data structures are used, and the traversal is iterative.

2. Validate Binary Search Tree

Difficulty: Medium
Time: 20 mins

Problem Statement:

Given a binary tree, determine if it is a valid binary search tree (BST). A binary search tree is valid if: 1. The left subtree of a node contains only nodes with keys **less than** the node's key. 2. The right subtree of a node contains only nodes with keys **greater than** the node's key. 3. Both the left and right subtrees must also be binary search trees.

Sample Input:

```
root = [2,1,3]
```

Sample Output:

```
true
```

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def isValidBST(root: TreeNode, low=float('-inf'), high=float('inf')) -> bool:
    # Empty trees are valid BSTs
    if not root:
        return True

    # Check the current node value
    if root.val <= low or root.val >= high:
        return False

    # Check recursively for every node.
    # The right subtree must have all values > root.val
    # The left subtree must have all values < root.val
    return (isValidBST(root.left, low, root.val) and
            isValidBST(root.right, root.val, high))
```

```
# Sample Test Case
root = TreeNode(2)
root.left = TreeNode(1)
root.right = TreeNode(3)

print("Is valid BST:", isValidBST(root))  # Output: Is valid BST: True
```

```
Is valid BST: True
```

Time Complexity: O(n), where n is the number of nodes, as each node is visited once.
Space Complexity: O(h), where h is the height of the tree, due to the recursive call stack.

3. Kth Smallest Element in a BST

Difficulty: Medium
Time: 25 mins

Problem Statement:

Given the root of a binary search tree, and an integer `k`, return the `k`th smallest value (1-indexed) of all the values of the nodes in the BST.

Sample Input:

```
root = [3,1,4,null,2], k = 1
```

Sample Output:

```
1
```

```
class TreeNode:
    """
    Definition for a binary tree node.
    """
    def __init__(self, val=0, left=None, right=None):
        self.val = val  # Value of the node
        self.left = left  # Left child of the node
        self.right = right  # Right child of the node
```

```python
def kthSmallest(root: TreeNode, k: int) -> int:
    """
    Finds the kth smallest element in a Binary Search Tree (BST).
    This is achieved using an in-order traversal (left, root, right) since
    it visits nodes in ascending order in a BST.

    :param root: The root node of the BST
    :param k: The position (1-based) of the smallest element to find
    :return: The value of the kth smallest element
    """
    # Initialize a stack to simulate in-order traversal without recursion
    stack = []

    # Perform an iterative in-order traversal
    while True:
        # Traverse to the leftmost node
        while root:
            stack.append(root)  # Push the current node onto the stack
            root = root.left  # Move to the left child

        # Process the node on the top of the stack
        root = stack.pop()  # Pop the top node from the stack
        k -= 1  # Decrease k as we've found one of the smallest elements

        # If k becomes 0, we've found the kth smallest element
        if k == 0:
            return root.val  # Return the value of the current node

        # Move to the right child to continue the traversal
        root = root.right

# Sample Test Case
# Construct the following BST:
#       3
#      / \
#     1   4
#      \
#       2
root = TreeNode(3)  # Root node
root.left = TreeNode(1)  # Left child of root
root.right = TreeNode(4)  # Right child of root
root.left.right = TreeNode(2)  # Right child of the left child
```

```
k = 1  # Find the 1st smallest element
print(f"The {k}th smallest element is {kthSmallest(root, k)}")
```

```
The 1th smallest element is 1
```

Time Complexity: Best case O(k), Worst case O(N), where N is the number of nodes in the tree.
Space Complexity: O(log N) for a balanced tree, O(N) for a skewed tree due to stack usage.

4. Inorder Successor in BST

Difficulty: Medium
Time: 30 mins

Problem Statement:

Given a binary search tree and a node `p` in it, find the inorder successor of that node in the BST. The inorder successor of a node is the node with the smallest key greater than `p`'s key. If there is no such node, return `null`.

Sample Input:

```
root = [2,1,3], p = 1
```

Sample Output:

```
2
```

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        # Initialize a binary tree node with a value,
        # and optional left and right child nodes
        self.val = val
        self.left = left
        self.right = right

def inorderSuccessor(root: TreeNode, p: TreeNode) -> TreeNode:
```

```
    """
    Finds the inorder successor of a given node `p`
    in a binary search tree (BST).
    The inorder successor of a node is the node with the
    smallest value greater than the given node.
    """
    # Initialize the variable to keep track of the potential successor
    successor = None

    # Traverse the tree to find the inorder successor
    while root:
        if p.val >= root.val:
            # If the value of `p` is greater than or
            # equal to the current root's value,
            # move to the right subtree (inorder successor must be larger).
            root = root.right
        else:
            # If the value of `p` is less than the current root's value,
            # the current root is a potential successor. Save it.
            successor = root
            # Move to the left subtree to continue searching
            # for a smaller valid successor.
            root = root.left

    # Return the found successor, or None if no successor exists.
    return successor

# Sample Test Case
# Construct the BST:
#         5
#       /   \
#      3     6
#     / \
#    2   4
#   /
#  1
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(6)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.left.left.left = TreeNode(1)
```

```
# Define the target node `p` (node with value 3)
p = root.left  # Node with value 3

# Find the inorder successor of `p`
successor = inorderSuccessor(root, p)

# Print the result with an explanation
print(f"Successor of {p.val} is {successor.val if successor else 'None'}")
# Output: Successor of 3 is 4
```

```
Successor of 3 is 4
```

Time Complexity: O(h), where h is the height of the tree, as we traverse from the root to a leaf in the worst case.
Space Complexity: O(1), as no additional space is used other than a few variables.

5. Convert Sorted Array to Binary Search Tree

Difficulty: Easy
Time: 20 mins

Problem Statement:

Given an integer array `nums` where the elements are sorted in **ascending** order, convert it to a height-balanced binary search tree (BST).

Sample Input:

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

Sample Output:

```
[0,-3,9,-10,null,5]
```

```python
class TreeNode:
    # Definition of a binary tree node.
    def __init__(self, val=0, left=None, right=None):
        self.val = val  # Value of the node
        self.left = left  # Left child of the node
        self.right = right  # Right child of the node

def sortedArrayToBST(nums: list) -> TreeNode:
    """
    Converts a 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.

    Args:
        nums (list): A sorted (ascending) list of integers.

    Returns:
        TreeNode: The root of the height-balanced BST.
    """
    if not nums:  # Base case: If the array is empty, return None.
        return None

    # Find the middle index of the array. This ensures the tree is balanced.
    mid = len(nums) // 2

    # Create a new tree node with the value at the middle of the array.
    root = TreeNode(nums[mid])

    # Recursively build the left subtree using the left half of the array.
    root.left = sortedArrayToBST(nums[:mid])

    # Recursively build the right subtree using the right half of the array.
    root.right = sortedArrayToBST(nums[mid+1:])

    return root  # Return the root of the constructed BST.

# Sample test case
nums = [-10, -3, 0, 5, 9]  # Input sorted array
root = sortedArrayToBST(nums)  # Construct BST from the array
```

```python
def inOrderTraversal(root):
    """
    Performs an in-order traversal of a binary tree.

    In-order traversal visits nodes in ascending order for a BST:
    1. Visit the left subtree
    2. Visit the root node
    3. Visit the right subtree

    Args:
        root (TreeNode): The root of the binary tree.

    Returns:
        list: A list of values obtained from the in-order traversal.
    """
    if root is None:  # Base case: If the tree is empty, return an empty list.
        return []

    # Traverse left subtree, visit root, then traverse right subtree.
    return (
        inOrderTraversal(root.left) +  # Traverse the left subtree
        [root.val] +                   # Visit the root node
        inOrderTraversal(root.right)   # Traverse the right subtree
    )

# Print the in-order traversal of the constructed BST.
print("In-order traversal of the constructed BST:", inOrderTraversal(root))
# Output: In-order traversal of the constructed BST: [-10, -3, 0, 5, 9]
```

```
In-order traversal of the constructed BST: [-10, -3, 0, 5, 9]
```

Time Complexity: O(n), where n is the number of elements in the array, as each element is processed exactly once.
Space Complexity: O(log n) for the recursion stack, as the tree height for a balanced BST is log(n) in the worst case.