BST Problems (5 Problems)

Problem 1: Lowest Common Ancestor of a BST

Problem Summary

Given a binary search tree (BST) and two nodes `p` and `q`, find their lowest common ancestor (LCA). The LCA is the deepest node that has both `p` and `q` as descendants.

Pattern(s)

- **BST Property + Two Pointers (Ancestor Search)**
  → Exploits the BST invariant: left < root < right

Interview Template: BST Property + Two Pointers (Ancestor Search)

Aspect	Explanation
How to recognize	- Problem involves finding an ancestor in a BST- Input includes two nodes, and you're to return their shared ancestor- No need for extra space; can traverse using BST ordering
Step-by-step thinking process	1. Start at root2. If both `p` and `q` are smaller than current node → go left3. If both are larger → go right4. If split (one smaller, one larger) → current node is LCA5. Return when split condition met
Common pitfalls & edge cases	- `p == q` → return `p`- One node is the ancestor of the other → return the ancestor- Empty tree → not applicable (assumed valid input)- Incorrect direction logic (e.g., going wrong way due to equality check)

Solution with Inline Comments

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

```python
def lowestCommonAncestor(root: TreeNode, p: TreeNode, q: TreeNode)-> TreeNode:
    # We'll use the BST property: left < root < right
    # So we can decide which subtree to explore based on values

    while root:
        # If both p and q are smaller than root, LCA must be in left subtree
        if p.val < root.val and q.val < root.val:
            root = root.left
        # If both are greater, LCA must be in right subtree
        elif p.val > root.val and q.val > root.val:
            root = root.right
        # Otherwise, they are on different sides (or one is root),
        # so root is LCA
        else:
            break  # This is our answer

    return root  # At this point, root is the LCA


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
    # Expected Output: 6 (since 6 is the LCA of 2 and 8)

    # Build the tree
    #        6
    #       / \
    #      2   8
    #     / \ / \
    #    0  4 7  9
    #      / \
    #     3   5
    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)
```

```python
p = root.left       # Node with val = 2
q = root.right      # Node with val = 8

# Call function
lca = lowestCommonAncestor(root, p, q)

# Output should be 6
print("Output:", lca.val)  # Output: 6
```

Step-by-Step Walkthrough (Example)

- Start at `root = 6`
- `p.val = 2 < 6`, `q.val = 8 > 6` → split → return `6`
- Done.

Matches expected output.

Complexity

- **Time**: O(h), where h is height of BST → O(log n) average, O(n) worst case
- **Space**: O(1) — iterative, no recursion stack

Problem 2: [Validate Binary Search Tree]

Problem Summary

Given a binary tree, determine if it is a valid BST. A valid BST satisfies: - Left subtree contains only nodes with values < root - Right subtree contains only nodes with values > root - Both subtrees are also valid BSTs

Pattern(s)

- **Divide & Conquer with Range Bounds (DFS)**
  → Use min/max bounds to validate each node's value during traversal

Interview Template: Divide & Conquer with Range Bounds

Aspect	Explanation
How to recognize	- You're validating structure based on ordering constraints- Need to track valid range per node- Often recursive, but can be iterative with stack
Step-by-step thinking process	1. Define valid range `(min_val, max_val)` for current node2. Root must satisfy `min_val < root.val < max_val`3. Recursively validate left subtree with new upper bound = root.val4. Recursively validate right subtree with new lower bound = root.val5. Base case: null node is valid
Common pitfalls & edge cases	- Using `int` limits without proper type handling (use `float('-inf')`, `float('inf')`)- Not updating bounds correctly (left child gets updated max, right gets updated min)- Assuming all nodes in left subtree are less than root → need global constraint

---

Solution with Inline Comments

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

def isValidBST(root: TreeNode) -> bool:
    # Helper function to validate BST using bounds
    # Returns True if subtree rooted at 'node' is
    # valid given min and max constraints
    def validate(node, min_val, max_val):
        # Null node is valid
        if not node:
            return True

        # Check if current node violates the range
        if node.val <= min_val or node.val >= max_val:
            return False

        # Recursively validate left and right subtrees with updated bounds
        # Left subtree: must be < node.val → new max = node.val
```

```
        # Right subtree: must be > node.val → new min = node.val
        return (validate(node.left, min_val, node.val) and
                validate(node.right, node.val, max_val))

    # Start validation with unbounded range
    return validate(root, float('-inf'), float('inf'))


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [5,1,4,null,null,3,6]
    # Output: false (since 4 is in right subtree of 5, but 4 < 5 → invalid)

    root = TreeNode(5)
    root.left = TreeNode(1)
    root.right = TreeNode(4)
    root.right.left = TreeNode(3)
    root.right.right = TreeNode(6)

    result = isValidBST(root)
    print("Output:", result)  # Output: False
```

Step-by-Step Walkthrough (Example)

- Start: `validate(root=5, min=-inf, max=inf)`
  - 5   (-inf, inf)? Yes.
  - Validate left: `validate(1, -inf, 5)` → OK
    - 1   (-inf, 5)? Yes.
    - Left/right null → valid
  - Validate right: `validate(4, 5, inf)`
    - 4   5 → violates `val > min` → returns `False`
- Entire tree invalid → returns `False`

Correctly identifies invalid BST.

Complexity

- **Time**: O(n) — visit every node once
- **Space**: O(h) — recursion depth (stack space); h = height

---

Problem 3: Kth Smallest Element in a BST

Problem Summary

Given a BST, return the k-th smallest element (1-indexed).

Pattern(s)

- **Inorder Traversal (Order Statistics)**
  → Inorder traversal of BST gives sorted order

Interview Template: Inorder Traversal (Order Statistics)

Aspect	Explanation
How to recognize	- Asked for "k-th smallest", "sorted order", or "rank"- BST is involved- Can use inorder traversal to get elements in increasing order
Step-by-step thinking process	1. Perform inorder traversal (left → root → right)2. Keep a counter of visited nodes3. When counter reaches k, return current node's value4. Stop early (optimize): don't traverse entire tree if k is small
Common pitfalls & edge cases	- Forgetting that k is 1-indexed- Doing full traversal even when k is small- Not stopping early → inefficient- Recursive depth overflow for deep trees

---

Solution with Inline Comments

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

def kthSmallest(root: TreeNode, k: int) -> int:
    # Use inorder traversal with early termination
    # Stack-based (iterative) to avoid recursion depth issues

    stack = []
```

```python
    curr = root
    count = 0  # number of nodes processed

    while stack or curr:
        # Go to leftmost node
        while curr:
            stack.append(curr)
            curr = curr.left

        # Pop and process
        curr = stack.pop()
        count += 1

        # If we've reached k-th node, return its value
        if count == k:
            return curr.val

        # Move to right subtree
        curr = curr.right

    # Should never reach here if k is valid
    raise ValueError("k is out of bounds")


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [3,1,4,null,2], k = 1
    # Expected Output: 1 (smallest element)

    root = TreeNode(3)
    root.left = TreeNode(1)
    root.right = TreeNode(4)
    root.left.right = TreeNode(2)

    result = kthSmallest(root, 1)
    print("Output:", result)  # Output: 1
```

Step-by-Step Walkthrough (Example)

- Start: `curr = 3`, stack=[], count=0
- Traverse left: push 3 → 1 → null
- Pop 1 → count=1 → k=1 → return 1

Correct.

Complexity

- **Time**: O(H + k), where H is height → best case O(k), worst O(n)
- **Space**: O(H) — stack size (max depth)

Problem 4: Inorder Successor in BST

Problem Summary

Given a node in a BST, find its inorder successor (next node in inorder traversal). Assume parent pointers are **not** available.

Pattern(s)

- **BST Search with Candidate Tracking**
  → Track potential successor while traversing

Interview Template: BST Search with Candidate Tracking

Aspect	Explanation
How to recognize	- You're asked for next element in sorted order- Given a single node, not root- No parent links → must search from root
Step-by-step thinking process	1. If node has right child → successor is leftmost node in right subtree2. Else → successor is the first ancestor where node is in left subtree3. Use binary search-like path: if node < current → candidate = current, move left; else move right
Common pitfalls & edge cases	- Forgetting that successor might be ancestor- Not tracking candidate properly- Assuming successor always exists (should handle case where none exists)

Solution with Inline Comments

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

def inorderSuccessor(root: TreeNode, p: TreeNode) -> TreeNode:
    successor = None

    while root:
        if p.val < root.val:
            # Current node could be successor
            successor = root
            root = root.left  # Look for smaller candidates
        elif p.val > root.val:
            root = root.right  # Continue searching in right subtree
        else:
            # p == root → successor is leftmost in right subtree
            if root.right:
                root = root.right
                while root.left:
                    root = root.left
                return root
            else:
                # No right subtree → successor is stored in 'successor'
                return successor

    return successor  # Could be None if no successor exists


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: root = [2,1,3], p = 1
    # Expected Output: 2 (inorder: 1,2,3 → successor of 1 is 2)

    root = TreeNode(2)
    root.left = TreeNode(1)
    root.right = TreeNode(3)

    p = root.left  # node with val=1
```

```
succ = inorderSuccessor(root, p)
print("Output:", succ.val)  # Output: 2
```

Step-by-Step Walkthrough (Example)

- Start: `root = 2`, `successor = None`
- `p.val=1 < 2` → set `successor = 2`, go left
- Now `root = 1`, `p.val == 1` → enter `else`
  - `root.right` is `None` → return `successor = 2`

Correct.

Complexity

- **Time**: O(h) — height of tree
- **Space**: O(1)

Problem 5: Convert Sorted Array to BST

Problem Summary

Given a sorted array, construct a height-balanced BST (i.e., difference in heights between left and right subtrees 1).

Pattern(s)

- **Divide & Conquer (Construction)**
  → Pick middle element as root → recursively build left/right subtrees

Aspect	Explanation

Interview Template: Divide & Conquer (Construction)

Aspect	Explanation
How to recognize	- Input is sorted (implying order)- Task is to build balanced tree- Can pick midpoint as root → ensures balance
Step-by-step thinking process	1. Choose middle element as root2. Recursively build left subtree from left half3. Recursively build right subtree from right half4. Base case: empty subarray → return None
Common pitfalls & edge cases	- Not choosing mid properly (off-by-one errors)- Creating unbalanced tree by picking wrong pivot- Returning wrong node types (must return TreeNode)

Solution with Inline Comments

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

def sortedArrayToBST(nums):
    # Helper function to build BST from nums[left:right+1]
    def build(left, right):
        # Base case: no elements
        if left > right:
            return None

        # Choose middle element as root (ensures balance)
        mid = (left + right) // 2
        root = TreeNode(nums[mid])

        # Recursively build left and right subtrees
        root.left = build(left, mid - 1)
```

```python
        root.right = build(mid + 1, right)

        return root

    return build(0, len(nums) - 1)


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: nums = [-10,-3,0,5,9]
    # Expected Output: [0,-3,9,-10,null,5] → any height-balanced BST is acceptable

    nums = [-10, -3, 0, 5, 9]
    root = sortedArrayToBST(nums)

    # Print level-order traversal (for verification)
    from collections import deque
    result = []
    queue = deque([root])

    while queue:
        node = queue.popleft()
        if node:
            result.append(node.val)
            queue.append(node.left)
            queue.append(node.right)
        else:
            result.append(None)

    # Trim trailing Nones
    while result and result[-1] is None:
        result.pop()

    print("Output:", result)  # Output: [0, -3, 9, -10, None, 5]
```

Step-by-Step Walkthrough (Example)

- `nums = [-10,-3,0,5,9]`, `left=0`, `right=4`
- `mid = 2`, `root = 0`
- Left: `build(0,1)` → `mid=0` → `-10`, then `build(1,1)` → `-3`
- Right: `build(3,4)` → `mid=3` → `5`, then `build(4,4)` → `9`
- Result: balanced tree with 0 as root

Valid height-balanced BST.

Complexity

- **Time**: O(n) — each element visited once
- **Space**: O(log n) — recursion depth (stack space); also O(n) for output tree

---