Binary Search Tree

1. Validate Binary Search Tree

Pattern: Tree Traversal + Range Validation (In-Order / DFS with Bounds)

Problem Statement

Given the `root` of a binary tree, determine if it is a valid binary search tree (BST).

A **valid BST** is defined as follows:
- 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.

Note: Duplicate values are **not allowed** in a BST per this problem.

Sample Input & Output

```
Input: root = [2,1,3]
Output: true
Explanation: 2 is root; left=1 (<2), right=3 (>2); both subtrees valid.
```

```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: Root=5; right child=4 (<5 ).
Also, 4's left=3 is <5 but appears in right subtree → violates BST.
```

```
Input: root = [1]
Output: true
Explanation: Single node is always a valid BST.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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

class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        # STEP 1: Initialize recursive helper with bounds
        #   - Use -inf and +inf as initial valid range for root
        #   - Each recursive call tightens the allowed range

        def validate(node, low, high):
            # STEP 2: Base case - empty node is valid
            if not node:
                return True

            # STEP 3: Check current node against bounds
            #   - Must satisfy: low < node.val < high
            if node.val <= low or node.val >= high:
                return False

            # STEP 4: Recurse left and right with updated bounds
```

```python
            #   - Left subtree: upper bound becomes node.val
            #   - Right subtree: lower bound becomes node.val
            return (validate(node.left, low, node.val) and
                    validate(node.right, node.val, high))

        # STEP 5: Start validation from root with full range
        return validate(root, float('-inf'), float('inf'))

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - valid BST
    root1 = TreeNode(2, TreeNode(1), TreeNode(3))
    print("Test 1:", sol.isValidBST(root1))  # Expected: True

    #  Test 2: Edge case - single node
    root2 = TreeNode(1)
    print("Test 2:", sol.isValidBST(root2))  # Expected: True

    #  Test 3: Tricky case - invalid BST (right child too small)
    #       5
    #      / \
    #     1   4
    #        / \
    #       3   6
    root3 = TreeNode(5)
    root3.left = TreeNode(1)
    root3.right = TreeNode(4)
    root3.right.left = TreeNode(3)
    root3.right.right = TreeNode(6)
    print("Test 3:", sol.isValidBST(root3))  # Expected: False
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3** (`[5,1,4,null,null,3,6]`) step by step.

1. **Call `isValidBST(root3)`**
    - `root3.val = 5`
    - Calls `validate(root3, -inf, +inf)`
2. **Inside `validate(node=5, low=-inf, high=+inf)`**
    - Node exists → continue
    - Check: `5 <= -inf`? No. `5 >= +inf`? No. → OK
    - Now recurse:
        - Left: `validate(1, -inf, 5)`
        - Right: `validate(4, 5, +inf)`
3. **Process left subtree: `validate(1, -inf, 5)`**
    - `1` is between `-inf` and `5` → OK
    - Left child = `None` → returns `True`
    - Right child = `None` → returns `True`
    - Returns `True`
4. **Process right subtree: `validate(4, 5, +inf)`**
    - Check: `4 <= 5`? **Yes** → but condition is `node.val <= low` → `4 <= 5` is true, but wait!
        - **Correction**: `low = 5`, so `4 <= 5` → **true**, which triggers `return False`
        - Because in right subtree of 5, all values **must be > 5**, but 4 is **not > 5**
    - So: `4 >= high`? No (`high = inf`)
      But `4 <= low (5)` → **yes** → **invalid!**
    - Returns `False`
5. **Final result**: `True and False` → `False`

Output: `False` — correctly identifies invalid BST.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We visit every node exactly once in the worst case (skewed tree or fully valid BST).

- **Space Complexity**: `O(h)`, where `h` is height of tree

  Due to recursion stack depth. In worst case (skewed tree), `h = n`; in balanced tree, `h = log n`.

2. Convert Sorted Array to Binary Search Tree

Pattern: Divide and Conquer / Binary Search Tree Construction

---

Problem Statement

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

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.

Sample Input & Output

```
Input: nums = [-10, -3, 0, 5, 9]
Output: [0, -3, 9, -10, null, 5]
Explanation: One possible answer is [0, -3, 9, -10, null, 5],
which represents the following height-balanced BST:
      0
     / \
   -3   9
   /   /
 -10  5
```

```
Input: nums = [1, 3]
Output: [3, 1] or [1, null, 3]
Explanation: Both are valid height-balanced BSTs.
```

```
Input: nums = [0]
Output: [0]
Explanation: Single-node tree is trivially balanced.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List, Optional

# 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

class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        # STEP 1: Base case - empty subarray
        #   - Return None to terminate recursion
        if not nums:
            return None

        # STEP 2: Choose middle element as root
        #   - Ensures left/right subtrees differ by  1 in size
        mid = len(nums) // 2
        root = TreeNode(nums[mid])

        # STEP 3: Recursively build left and right subtrees
        #   - Left: elements before mid (guaranteed < root.val)
        #   - Right: elements after mid (guaranteed > root.val)
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid + 1:])
```

```python
        # STEP 4: Return constructed subtree root
        #   - Base case handles empty slices automatically
        return root

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    # Helper to serialize tree (preorder with None markers)
    def serialize(root):
        if not root:
            return [None]
        return [root.val] + serialize(root.left) + serialize(root.right)

    #   Test 1: Normal case
    tree1 = sol.sortedArrayToBST([-10, -3, 0, 5, 9])
    ser1 = serialize(tree1)
    # Trim trailing Nones for cleaner comparison
    while ser1 and ser1[-1] is None:
        ser1.pop()
    print("Test 1:", ser1)  # Expect: [0, -3, -10, None, None, None, 9, 5]

    #   Test 2: Edge case - two elements
    tree2 = sol.sortedArrayToBST([1, 3])
    ser2 = serialize(tree2)
    while ser2 and ser2[-1] is None:
        ser2.pop()
    print("Test 2:", ser2)  # Expect: [3, 1] or [1, None, 3]

    #   Test 3: Tricky/negative - single element
    tree3 = sol.sortedArrayToBST([0])
    ser3 = serialize(tree3)
    while ser3 and ser3[-1] is None:
        ser3.pop()
    print("Test 3:", ser3)  # Expect: [0]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `nums = [-10, -3, 0, 5, 9]`.

1. **Initial Call**: `sortedArrayToBST([-10, -3, 0, 5, 9])`
   - `nums` is not empty → proceed.
   - `mid = 5 // 2 = 2` → `root.val = nums[2] = 0`.
   - Create `TreeNode(0)`.
2. **Build Left Subtree**: `sortedArrayToBST([-10, -3])`
   - `mid = 2 // 2 = 1` → `root.val = -3`.
   - Left: `sortedArrayToBST([-10])` → returns `TreeNode(-10)`.
   - Right: `sortedArrayToBST([])` → returns `None`.
   - So left subtree of `0` is `-3` with left child `-10`.
3. **Build Right Subtree**: `sortedArrayToBST([5, 9])`
   - `mid = 2 // 2 = 1` → `root.val = 9`.
   - Left: `sortedArrayToBST([5])` → returns `TreeNode(5)`.
   - Right: `sortedArrayToBST([])` → `None`.
   - So right subtree of `0` is `9` with left child `5`.
4. **Final Tree**:

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

5. **Serialization (preorder)**:
   `[0, -3, -10, None, None, None, 9, 5, None, None, None]`
   After trimming trailing `None`s:
   `[0, -3, -10, None, None, None, 9, 5]`

Each recursive call builds a balanced subtree by always picking the middle element — this guarantees minimal height.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each element is visited exactly once to create a `TreeNode`. Slicing creates new lists, but total work across all levels is still linear (like merge sort's merge step).

- **Space Complexity**: `O(log n)`

  Recursion depth is `log n` (height of balanced tree). However, **slicing** creates new sublists, leading to **`O(n log n)` auxiliary space** in this implementation. *Note: A more space-efficient version would pass indices instead of slicing — but this version prioritizes clarity for learning.*

3. Kth Smallest Element in a BST

Pattern: In-Order Traversal (Tree DFS)

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 tree.

You may assume `k` is always valid (`1` `k` number of nodes).

Sample Input & Output

```
Input: root = [3,1,4,null,2], k = 1
Output: 1
Explanation: In-order traversal gives [1,2,3,4]; 1st smallest is 1.
```

```
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3
Explanation: In-order = [1,2,3,4,5,6]; 3rd smallest is 3.
```

```
Input: root = [1], k = 1
Output: 1
Explanation: Only one node - it's the 1st smallest.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

# 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

class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        # STEP 1: Initialize structures
        #   - Use in-order traversal (left → root → right)
        #   - BST property ensures ascending order

        self.count = 0      # Tracks how many nodes visited
        self.result = None  # Stores kth smallest once found

        # STEP 2: Main loop / recursion
        #   - Recurse left first (smallest values)
```

```
        #   - Visit current node → increment count
        #   - Stop early if result found (optimization)
        self._inorder(root, k)

        # STEP 4: Return result
        #   - Guaranteed to be set since k is valid
        return self.result

    def _inorder(self, node: Optional[TreeNode], k: int):
        if not node or self.result is not None:
            return

        # Traverse left subtree
        self._inorder(node.left, k)

        # Visit current node
        self.count += 1
        if self.count == k:
            self.result = node.val
            return  # Early exit - no need to go further

        # Traverse right subtree
        self._inorder(node.right, k)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    # Tree: [3,1,4,null,2]
    root1 = TreeNode(3)
    root1.left = TreeNode(1)
    root1.right = TreeNode(4)
    root1.left.right = TreeNode(2)
    assert sol.kthSmallest(root1, 1) == 1

    #  Test 2: Edge case - single node
    root2 = TreeNode(1)
    assert sol.kthSmallest(root2, 1) == 1

    #  Test 3: Tricky - deeper tree, k=3
    # Tree: [5,3,6,2,4,null,null,1]
```

```python
root3 = TreeNode(5)
root3.left = TreeNode(3)
root3.right = TreeNode(6)
root3.left.left = TreeNode(2)
root3.left.right = TreeNode(4)
root3.left.left.left = TreeNode(1)
assert sol.kthSmallest(root3, 3) == 3

print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 3** (`root = [5,3,6,2,4,null,null,1]`, `k = 3`):

1. **Start**: `count = 0`, `result = None`
   Call `_inorder(root=5, k=3)`
2. **Go left to 3** → then to **2** → then to **1** (leftmost)
3. **Visit node 1**:
   - `count` becomes 1
   - Not equal to `k=3` → continue
4. **Backtrack to node 2**:
   - `count` becomes 2
   - Still not 3 → continue
5. **Backtrack to node 3**:
   - `count` becomes **3**
   - Match! Set `result = 3`
   - Return immediately (skip right subtree of 3 and entire right of 5)
6. **Final state**: `result = 3` → returned

   The in-order traversal naturally visits nodes in **sorted order** due to BST structure. We stop as soon as we hit the `k`th node — no need to traverse the whole tree.

Complexity Analysis

- **Time Complexity**: `O(H + k)`

  In the worst case, we traverse from root to the leftmost leaf (`H` = height), then visit `k` nodes. For balanced BST, `H = log n`; for skewed, `H = n`. So worst-case `O(n)`, but average `O(log n + k)`.

- **Space Complexity**: `O(H)`

  Due to recursion stack depth, which equals tree height `H`. No extra data structures beyond a few variables.

4. Inorder Successor in BST

Pattern: Binary Search Tree (BST) Traversal + Successor Logic

---

Problem Statement

> Given the `root` of a binary search tree and a node `p` in it, return the **inorder successor** of that node in the BST. If the given node has no inorder successor in the tree, return `null`.
>
> The **inorder successor** of a node `p` is the node with the smallest key **greater than** `p.val`.
>
> You will be given the tree as a root node and a reference to a node `p`, **not its value**.

Sample Input & Output

```
Input: root = [2,1,3], p = 1
Output: 2
Explanation: The inorder traversal is [1,2,3]. The successor of 1 is 2.
```

```
Input: root = [5,3,6,2,4,null,null,1], p = 6
Output: null
Explanation: 6 is the largest node; no node has a greater value.
```

```
Input: root = [2,1,3], p = 3
Output: null
Explanation: 3 is the rightmost node; no successor exists.
```

LeetCode Editorial Solution + Inline Tests

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

class Solution:
    def inorderSuccessor(
        self, root: TreeNode, p: TreeNode
    ) -> TreeNode | None:
        # STEP 1: Initialize successor as None
        #   - We'll update it only when we find a node > p.val
        successor = None

        # STEP 2: Traverse using BST property
        #   - If current node > p.val, it's a candidate
        #   - Then go left to find smaller valid candidate
        #   - Else, go right to find larger values
        current = root
```

```python
        while current:
            if current.val > p.val:
                successor = current      # valid candidate
                current = current.left   # try to find smaller one
            else:
                current = current.right  # need larger values

        # STEP 3: Return successor (could be None)
        #   - Handles edge case where p is max node
        return successor

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case - successor exists
    # Tree: [2,1,3], p = node(1)
    root1 = TreeNode(2)
    root1.left = TreeNode(1)
    root1.right = TreeNode(3)
    p1 = root1.left
    result1 = sol.inorderSuccessor(root1, p1)
    print("Test 1:", result1.val if result1 else None)  # Expected: 2

    #  Test 2: Edge case - p is max node
    # Tree: [2,1,3], p = node(3)
    p2 = root1.right
    result2 = sol.inorderSuccessor(root1, p2)
    print("Test 2:", result2.val if result2 else None)  # Expected: None

    #  Test 3: Tricky case - deep tree, successor is ancestor
    # Tree: [5,3,6,2,4,null,null,1], p = node(4)
    root3 = TreeNode(5)
    root3.left = TreeNode(3)
    root3.right = TreeNode(6)
    root3.left.left = TreeNode(2)
    root3.left.right = TreeNode(4)
    root3.left.left.left = TreeNode(1)
    p3 = root3.left.right  # node(4)
    result3 = sol.inorderSuccessor(root3, p3)
    print("Test 3:", result3.val if result3 else None)  # Expected: 5
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3** step by step:
- **Tree**:
`5         / \        3    6        / \       2    4     /    1` - **Target `p`**: node with `val = 4`
- **Goal**: Find smallest node with value > 4 → should be `5`.

Initial state:
- `successor = None`
- `current = root (5)`

Step 1: `current.val = 5`, `p.val = 4`
- Since `5 > 4` → **candidate!**
- Set `successor = node(5)`
- Move `current = current.left` → now at node(3)

State:
- `successor = 5`
- `current = 3`

Step 2: `current.val = 3`, `p.val = 4`
- `3 <= 4` → not a candidate
- Move `current = current.right` → now at node(4)

State:
- `successor = 5`
- `current = 4`

Step 3: `current.val = 4`, `p.val = 4`
- `4 <= 4` → not greater → not a candidate
- Move `current = current.right` → now `None`

State:
- `successor = 5`
- `current = None` → loop ends

Return: `successor = node(5)` → output `5`

This works because:
- We **never go left unless we've found a valid candidate**, ensuring we don't miss the smallest greater value.
- The BST property lets us **eliminate half the tree** at each step.

Complexity Analysis

- **Time Complexity**: `O(h)`

  `h` = height of tree. In worst case (skewed tree), `h = n`. In balanced BST, `h = log n`. We traverse one path from root to leaf.

- **Space Complexity**: `O(1)`

  Only using a few pointers (`successor`, `current`). No recursion or extra data structures that scale with input.

5. Lowest Common Ancestor of a Binary Search Tree

Pattern: Binary Search Tree (BST) Traversal

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 Wikipedia: "The lowest common ancestor is defined between two nodes `p` and `q` as the lowest node in `T` that has both `p` and `q` as descendants (where we allow a node to be a descendant of itself)."

Sample Input & Output

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: Nodes 2 and 8 are in left and right subtrees of 6 → LCA is 6.
```

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: Both 2 and 4 are in left subtree; 2 is ancestor of 4 → LCA is 2.
```

```
Input: root = [2,1], p = 2, q = 1
Output: 2
Explanation: One node is the root itself → LCA is root.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import Optional

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

```python
class Solution:
    def lowestCommonAncestor(
        self,
        root: 'TreeNode',
        p: 'TreeNode',
        q: 'TreeNode'
    ) -> 'TreeNode':
        # STEP 1: Initialize current node as root
        #   - We traverse from root downward using BST property

        curr = root

        # STEP 2: Main loop - exploit BST ordering
        #   - In BST: left < root < right
        #   - If both p and q are < curr → LCA in left subtree
        #   - If both p and q are > curr → LCA in right subtree
        #   - Otherwise, curr splits p and q → curr is LCA

        while curr:
            if p.val < curr.val and q.val < curr.val:
                curr = curr.left
            elif p.val > curr.val and q.val > curr.val:
                curr = curr.right
            else:
                # p and q are on different sides (or one is curr)
                return curr

        # STEP 3: Return result
        #   - Loop always returns inside; this line never reached
        return None

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    # Helper to build minimal tree for testing
    def build_tree():
        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)
    return root

tree = build_tree()

#  Test 1: Normal case - p=2, q=8 → LCA=6
p1 = tree.left          # val=2
q1 = tree.right         # val=8
assert sol.lowestCommonAncestor(tree, p1, q1).val == 6

#  Test 2: Edge case - p=2, q=4 → LCA=2
p2 = tree.left          # val=2
q2 = tree.left.right    # val=4
assert sol.lowestCommonAncestor(tree, p2, q2).val == 2

#  Test 3: Tricky/negative - p=root, q=1 (in small tree)
small = TreeNode(2)
small.left = TreeNode(1)
p3 = small              # val=2
q3 = small.left         # val=1
assert sol.lowestCommonAncestor(small, p3, q3).val == 2

print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `p = node(2)`, `q = node(8)`, `root = node(6)`.

1. **Start**: `curr = root` → `curr.val = 6`
   - Check: Is `2 < 6` and `8 < 6`? → No (`8 > 6`)
   - Check: Is `2 > 6` and `8 > 6`? → No (`2 < 6`)

- So, `else` branch triggers → **return `curr` (node 6)**.

That's it! Because `p` is in left subtree and `q` in right, the root **splits** them → it's the LCA.

Now **Test 2**: `p = node(2)`, `q = node(4)`

1. `curr = 6`
   - Both `2 < 6` and `4 < 6` → go left → `curr = node(2)`
2. Now `curr.val = 2`
   - Check: `2 < 2` and `4 < 2`? → No
   - Check: `2 > 2` and `4 > 2`? → `2 > 2` is false
   - So, `else` → **return node(2)**
   - Why? Because `p` is the current node, and `q` is in its right subtree → LCA is `p`.

Key Insight: In a BST, we **never need to search both subtrees**. The ordering tells us exactly where to go — making this O(h) instead of O(n).

Complexity Analysis

- **Time Complexity**: `O(h)`

  `h` = height of BST. At each step, we go one level deeper. In balanced BST, `h = log n`; worst-case (skewed), `h = n`.

- **Space Complexity**: `O(1)`

  Only using a constant number of pointers (`curr`). No recursion stack or extra data structures.