Topic Introduction

Today, we're diving into a classic data structure: the **Binary Search Tree (BST)**. Whether you're a beginner or brushing up for interviews, mastering BSTs is a must.

A **BST** is a special kind of binary tree where, for every node:

- All values in the left subtree are **less** than the node's value.
- All values in the right subtree are **greater** than the node's value.

This property makes searching, insertion, and deletion efficient — often taking O(log n) time for balanced trees. In interviews, BSTs are popular because they test your understanding of recursion, tree traversals, and the ability to maintain invariants (rules that must always hold).

How does it work?
Imagine you want to find a number in a sorted list. Instead of checking every number, you keep splitting the list in half — classic binary search! A BST brings that same idea to a tree structure.

Example (not from today's problems):
Suppose you insert 5, 3, 7, 2, 4 into a BST:

```
      5
     / \
    3   7
   / \
  2   4
```

Notice how all left-descendants are less, and right-descendants are greater than their parent? That's the BST property.

Let's look at three classic BST problems that will solidify your understanding:

- **Validate Binary Search Tree** (LeetCode 98): Check if a given binary tree is a valid BST.
- **Recover Binary Search Tree** (LeetCode 99): Two nodes in a BST are swapped by mistake; restore the tree.
- **Convert Sorted Array to Binary Search Tree** (LeetCode 108): Build a height-balanced BST from a sorted array.

Why group these together?
All three are about BST validation and construction. The first checks if a tree **is** a BST (using inorder traversal). The second **fixes** a broken BST (also via inorder). The third **builds** a BST from sorted data (using divide and conquer). You'll see how similar traversal and recursive logic appears in all three, but applied in different ways.

Problem 1: Validate Binary Search Tree

LeetCode 98

Problem Statement (in my own words):
Given a binary tree, determine if it satisfies the BST property: for every node, all left descendants are less than the node, and all right descendants are greater.

Example Input/Output:

Input:

```
    2
   / \
  1   3
```

Output: `True` (This is a valid BST.)

Walkthrough:

- Node 2: left child is 1 (<2), right child is 3 (>2).
- Node 1 and 3 have no children, so nothing to check.
- All checks pass, so it's a valid BST.

Another test case for you to try:

```
    5
   / \
  1   4
     / \
    3   6
```

Is this a valid BST?

Brute-force approach:

For each node, check that all nodes in the left subtree are less, and all in the right are greater.

- This requires scanning whole subtrees for every node: O(n^2) time!

Optimal approach:

Use **inorder traversal**. In a BST, inorder traversal yields nodes in ascending order.

Core pattern:

- As you traverse the tree inorder (left, root, right), keep track of the previously visited value.
- If you ever find a current value <= previous, it's not a BST.

Step-by-step logic:

- Initialize `prev` to None.
- Traverse the tree in inorder.
- At each node:
    - If `prev` exists and node.val <= prev, return False.
    - Update `prev` to node.val.
- If traversal finishes with no issue, return True.

Python Solution:

```python
class Solution:
    def isValidBST(self, root):
        self.prev = None  # Stores the value of the previous node in inorder

        def inorder(node):
            if not node:
                return True
            if not inorder(node.left):
                return False
            if self.prev is not None and node.val <= self.prev:
                return False
            self.prev = node.val
            return inorder(node.right)

        return inorder(root)
```

Time Complexity: O(n), where n is the number of nodes (each visited once).

Space Complexity: O(h), where h is the tree height (for the recursion stack).

Explanation:

- `self.prev` holds the last value seen.
- The `inorder` function recursively checks the left subtree, then the current node, then the right subtree.
- If any value is not greater than the previous, the function returns False.

Trace Example:

Let’s trace this tree:

```
   2
  / \
 1   3
```

- Inorder visits: 1, 2, 3
- `prev` starts as None.
- Visit 1: prev=None, set prev=1
- Visit 2: prev=1, 2>1, set prev=2
- Visit 3: prev=2, 3>2, set prev=3
- All checks pass!

Try this test case:

```
   10
   / \
  5  15
     / \
    6  20
```

Is this a valid BST? Walk through it step by step!

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Recover Binary Search Tree

LeetCode 99

Problem Statement (in my own words):

A BST had two of its nodes swapped by mistake. Recover the tree (swap them back), modifying the tree in place.

How is this similar/different from Problem 1?

Both use BST properties and inorder traversal. But here, instead of just checking validity, you must **find and fix** the two nodes that are out of place.

Example Input/Output:

Input:

```
    3
   / \
  1   4
     /
    2
```

(Here, nodes 2 and 3 are swapped.)

Output:

```
    2
   / \
  1   4
     /
    3
```

(The tree is restored to a valid BST.)

Another test case for you:

```
    1
   /
  3
   \
    2
```

Swapped nodes? What should the fixed tree look like?

Brute-force approach:

Collect all node values, sort them, and reassign to the nodes in inorder.

- Time: O(n log n), Space: O(n)

Optimal approach:

Again, use inorder traversal. In a BST, inorder should give a sorted sequence. If nodes are swapped, you'll see two places where the order is violated.

Core pattern:

- As you traverse inorder, keep track of the previous node.
- When you detect a node where the current value < previous value, record the two nodes.
    - The first violation gives you the first node (prev) and possibly the second node (current).
    - If a second violation occurs, update the second node (current).

- After traversal, swap the values of the two nodes.

Step-by-step logic:

- Initialize `first`, `second`, `prev` as None.
- Perform inorder traversal.
- When you find prev.val > current.val:
    - If first is None, set first=prev and second=current.
    - If first is set, set second=current.
- After traversal, swap first.val and second.val.

Pseudocode:

```
prev = None
first = None
second = None

function inorder(node):
    if node is null:
        return
    inorder(node.left)
    if prev is not null and prev.val > node.val:
        if first is null:
            first = prev
            second = node
        else:
            second = node
    prev = node
    inorder(node.right)

inorder(root)
swap first.val and second.val
```

Example Walkthrough:

For the tree:

```
    3
   / \
  1   4
     /
    2
```

Inorder traversal: 1, 3, 2, 4

- prev=1, node=3 (ok)
- prev=3, node=2 (violation: 3>2, first=3, second=2)
- prev=2, node=4 (ok)

Swap 3 and 2. Tree is fixed!

Try this test case:

```
    2
   / \
  3   1
```

Which nodes are swapped? What should the fixed tree look like?

Time Complexity: O(n)
Space Complexity: O(h) (recursion stack)

Problem 3: Convert Sorted Array to Binary Search Tree

LeetCode 108

Problem Statement (in my own words):
Given a sorted array, convert it into a height-balanced BST. Each node's left subtree should be less than the node, and right subtree greater.

What's different here?
Now, we're **building** a BST from scratch using a sorted array, rather than traversing or fixing an existing tree. But the BST property is still central.

Example Input/Output:

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

Output:

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

(Other valid balanced BSTs are possible.)

Another test case for you to try:
`nums = [1, 2, 3, 4]`
What BSTs could you build?

Brute-force approach:
Insert elements one by one into a new BST.

- This can make the tree unbalanced (degenerate into a linked list).

Optimal approach:
Use **divide and conquer**:

- Always pick the middle element as the root to ensure balance.
- Recursively build left and right subtrees from left and right halves.

Pseudocode:

```
function buildBST(nums, left, right):
    if left > right:
        return null
    mid = (left + right) // 2
    node = TreeNode(nums[mid])
    node.left = buildBST(nums, left, mid-1)
    node.right = buildBST(nums, mid+1, right)
    return node

return buildBST(nums, 0, len(nums)-1)
```

Example Walkthrough:

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

- mid=2, node=0
    - Left: [-10, -3] -> mid=0, node=-10
        - Right: [-3] -> node=-3
    - Right: [5, 9] -> mid=3, node=5
        - Right: [9] -> node=9

Time Complexity: O(n) (visit each node once)
Space Complexity: O(h) (recursion stack, h=log n for balanced BST)

Try this test case:
`nums = [1, 2, 3]`
Draw the BST to check if it's balanced!

Reflect:
Notice how divide and conquer ensures balance. Could you do this iteratively? Or what about for a linked list input?

Summary and Next Steps

These three problems circle around the core ideas of **BST invariants**, **inorder traversal**, and **recursive construction**. You should now feel more comfortable:

- Recognizing and verifying the BST property.
- Using inorder traversal to spot errors or validate structure.
- Building balanced BSTs from sorted data.

Key patterns to remember:

- Inorder traversal yields sorted order for BSTs.
- To **validate** a BST, check the inorder sequence.
- To **fix** a BST, spot the misplaced nodes during inorder.
- To **build** a balanced BST, use divide and conquer with the middle element.

Common mistakes:

- Forgetting that left and right subtree values must be strictly less/greater, not just the immediate children.

- Not handling the recursion stack space in large trees.
- Failing to ensure balance when constructing BSTs from sorted arrays.

Action List:

- Try solving all three problems on your own, even the one with code above!
- For Problem 2 and 3, try a different approach (e.g., iterative traversal).
- Explore other BST-related problems, like insertion, deletion, or finding the kth smallest element.
- Compare your solutions with others — pay attention to how edge cases are handled.
- Don’t stress if you get stuck. Practice is the path to mastery!

Happy coding!