Complete Binary Search Tree (BST) Mastery Guide

Table of Contents

Introduction & Definition

What is a Binary Search Tree?

A Binary Search Tree (BST) is a tree data structure where each node has at most two children (left and right), and it follows a specific ordering property: for every node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value.

Formal Definition

For every node `N` in a BST:

- All nodes in the left subtree of N have values less than N.val
- All nodes in the right subtree of N have values greater than N.val
- Both left and right subtrees must also be BSTs (recursive property)

BST Node Structure

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;

    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode* left, TreeNode* right) : val(x), left(left), right(right) {}
};
```

Why BST?

Advantages:

- Efficient searching: O(log n) average case
- Maintains sorted order
- Dynamic structure (easy insertion/deletion)
- In-order traversal gives sorted sequence

Disadvantages:

- Can become unbalanced → O(n) worst case
- Requires more memory than arrays
- No constant-time access to elements

Properties of BST

Key Properties

1. **Ordering Property**: Left < Root < Right
2. **Uniqueness**: Typically, duplicate values are not allowed (or handled with specific rules)
3. **In-order Traversal**: Always produces sorted sequence
4. **Recursive Structure**: Each subtree is also a BST
5. **Min/Max Property**:
    - Minimum value: leftmost node
    - Maximum value: rightmost node

Time Complexity Summary

Operation	Average Case	Worst Case	Best Case
Search	O(log n)	O(n)	O(1)
Insert	O(log n)	O(n)	O(1)
Delete	O(log n)	O(n)	O(1)
Min/Max	O(log n)	O(n)	O(1)
Traversal	O(n)	O(n)	O(n)

Space Complexity: O(n) for storing n nodes, O(h) for recursion stack where h is height

Types of Binary Search Trees

1. Standard BST (Unbalanced)

- Basic BST without balancing guarantees
- Can degrade to O(n) operations in worst case
- Simple to implement

2. Balanced BST Variants

AVL Tree

- Height-balanced BST
- Balance factor: |height(left) - height(right)| ≤ 1
- Strictly balanced, faster lookups
- More rotations during insertion/deletion

Red-Black Tree

- Self-balancing using color properties
- Less strict balancing than AVL
- Faster insertion/deletion
- Used in many standard libraries (C++ map, Java TreeMap)

Splay Tree

- Self-adjusting BST
- Recently accessed elements move to root
- Good for non-uniform access patterns

3. Special BST Types

Complete Binary Search Tree

- All levels filled except possibly the last
- Last level filled from left to right

Perfect Binary Search Tree

- All internal nodes have two children
- All leaves at same level

Full Binary Search Tree

- Every node has 0 or 2 children

Basic Operations

1. Search Operation

Algorithm:

- Start from root
- If target < current node, go left
- If target > current node, go right
- If target == current node, found!
- If reach null, not found

Implementation:

```
// Iterative approach
TreeNode* searchBST(TreeNode* root, int target) {
    TreeNode* current = root;
    while (current != nullptr) {
        if (target == current->val) {
            return current;
        } else if (target < current->val) {
            current = current->left;
        } else {
            current = current->right;
        }
    }
    return nullptr;
}

// Recursive approach
TreeNode* searchBSTRecursive(TreeNode* root, int target) {
    if (root == nullptr || root->val == target) {
        return root;
    }

    if (target < root->val) {
        return searchBSTRecursive(root->left, target);
    }
    return searchBSTRecursive(root->right, target);
}
```

Time Complexity: O(log n) average, O(n) worst case
Space Complexity: O(1) iterative, O(h) recursive

2. Insert Operation

Algorithm:

- Find the correct position (similar to search)
- Insert new node as leaf

Implementation:

```cpp
// Recursive approach
TreeNode* insertBST(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    if (val < root->val) {
        root->left = insertBST(root->left, val);
    } else {
        root->right = insertBST(root->right, val);
    }

    return root;
}

// Iterative approach
TreeNode* insertBSTIterative(TreeNode* root, int val) {
    if (root == nullptr) {
        return new TreeNode(val);
    }

    TreeNode* current = root;
    while (true) {
        if (val < current->val) {
            if (current->left == nullptr) {
                current->left = new TreeNode(val);
                break;
            }
            current = current->left;
        } else {
            if (current->right == nullptr) {
                current->right = new TreeNode(val);
                break;
            }
            current = current->right;
        }
    }

    return root;
}
```

Time Complexity: O(log n) average, O(n) worst case
Space Complexity: O(1) iterative, O(h) recursive

3. Delete Operation

Three Cases:

Case 1: Node is a leaf (no children)

- Simply remove the node

Case 2: Node has one child

- Replace node with its child

Case 3: Node has two children

- Find inorder successor (smallest in right subtree) OR inorder predecessor (largest in left subtree)
- Replace node's value with successor/predecessor
- Delete the successor/predecessor

Implementation:

```
TreeNode* findMin(TreeNode* node) {
    while (node->left != nullptr) {
        node = node->left;
    }
    return node;
}

TreeNode* deleteBST(TreeNode* root, int key) {
    if (root == nullptr) {
        return nullptr;
    }

    // Find the node to delete
    if (key < root->val) {
        root->left = deleteBST(root->left, key);
    } else if (key > root->val) {
        root->right = deleteBST(root->right, key);
    } else {
        // Node found - handle three cases

        // Case 1: Leaf node or Case 2: One child
        if (root->left == nullptr) {
            TreeNode* temp = root->right;
            delete root;
            return temp;
        }
        if (root->right == nullptr) {
            TreeNode* temp = root->left;
            delete root;
            return temp;
        }

        // Case 3: Two children
        // Find inorder successor (min in right subtree)
        TreeNode* successor = findMin(root->right);
        root->val = successor->val;
        root->right = deleteBST(root->right, successor->val);
    }

    return root;
}
```

Time Complexity: O(log n) average, O(n) worst case
Space Complexity: O(h) for recursion

4. Find Minimum / Maximum

```
TreeNode* findMin(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}

TreeNode* findMax(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }
    while (root->right != nullptr) {
        root = root->right;
    }
    return root;
}
```

Time Complexity: O(log n) average, O(n) worst case

5. Validate BST

```
bool validate(TreeNode* node, long minVal, long maxVal) {
    if (node == nullptr) {
        return true;
    }

    if (node->val <= minVal || node->val >= maxVal) {
        return false;
    }

    return validate(node->left, minVal, node->val) &&
           validate(node->right, node->val, maxVal);
}

bool isValidBST(TreeNode* root) {
    return validate(root, LONG_MIN, LONG_MAX);
}
```

BST Traversals

1. Inorder Traversal (Left → Root → Right)

- **Result**: Sorted sequence
- **Use**: Get sorted elements, validate BST

```
void traverse(TreeNode* node, vector<int>& result) {
    if (node == nullptr) {
        return;
    }
    traverse(node->left, result);
    result.push_back(node->val);
    traverse(node->right, result);
}

vector<int> inorderTraversal(TreeNode* root) {
    vector<int> result;
    traverse(root, result);
    return result;
}
```

2. Preorder Traversal (Root → Left → Right)

- **Use**: Create copy of tree, serialize tree

```
void traverse(TreeNode* node, vector<int>& result) {
    if (node == nullptr) {
        return;
    }
    result.push_back(node->val);
    traverse(node->left, result);
    traverse(node->right, result);
}

vector<int> preorderTraversal(TreeNode* root) {
    vector<int> result;
    traverse(root, result);
    return result;
}
```

3. Postorder Traversal (Left → Right → Root)

- **Use**: Delete tree, bottom-up processing

```cpp
void traverse(TreeNode* node, vector<int>& result) {
    if (node == nullptr) {
        return;
    }
    traverse(node->left, result);
    traverse(node->right, result);
    result.push_back(node->val);
}

vector<int> postorderTraversal(TreeNode* root) {
    vector<int> result;
    traverse(root, result);
    return result;
}
```

4. Level Order Traversal

- **Use**: BFS, level-wise processing

```cpp
#include <queue>

vector<vector<int>> levelOrder(TreeNode* root) {
    if (root == nullptr) {
        return {};
    }

    vector<vector<int>> result;
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
        int levelSize = q.size();
        vector<int> level;

        for (int i = 0; i < levelSize; i++) {
            TreeNode* node = q.front();
            q.pop();
            level.push_back(node->val);

            if (node->left != nullptr) {
                q.push(node->left);
            }
            if (node->right != nullptr) {
                q.push(node->right);
            }
        }
        result.push_back(level);
    }

    return result;
}
```

Common Patterns in BST Problems

Pattern 1: BST Validation & Properties

Key Concept: Use BST properties (left < root < right) with range checking

Template:

```
bool validate(TreeNode* node, long minVal, long maxVal) {
    if (node == nullptr) {
        return true;
    }
    if (node->val <= minVal || node->val >= maxVal) {
        return false;
    }
    return validate(node->left, minVal, node->val) &&
           validate(node->right, node->val, maxVal);
}
```

When to Use:

- Validating BST
- Finding if value exists in range
- Checking BST properties

Pattern 2: Inorder Traversal for Sorted Access

Key Concept: Inorder traversal of BST gives sorted order

Template:

```
def inorder_template(root):
    result = []
    def inorder(node):
        if not node:
            return
        inorder(node.left)
        # Process current node
        result.append(node.val)
        inorder(node.right)
    inorder(root)
    return result
```

When to Use:

- Finding kth smallest/largest
- Getting sorted elements

- Finding pairs with target sum
- Range queries

Pattern 3: BST to Sorted Array & Back

Key Concept: Convert BST to sorted array, process, and rebuild

Template:

```
def bst_to_array(root):
    result = []
    def inorder(node):
        if not node:
            return
        inorder(node.left)
        result.append(node.val)
        inorder(node.right)
    inorder(root)
    return result

def sorted_array_to_bst(nums):
    if not nums:
        return None
    mid = len(nums) // 2
    root = TreeNode(nums[mid])
    root.left = sorted_array_to_bst(nums[:mid])
    root.right = sorted_array_to_bst(nums[mid+1:])
    return root
```

When to Use:

- Balancing BST
- Converting between structures
- Modifying BST structure

Pattern 4: Search with Optimization

Key Concept: Use BST property to eliminate half the tree

Template:

```
def search_optimized(root, target):
    while root:
        if target == root.val:
            return root
        elif target < root.val:
            root = root.left
        else:
            root = root.right
    return None
```

When to Use:

- Finding specific values
- Range searches
- Closest value problems

Pattern 5: Predecessor & Successor

Key Concept: Navigate tree structure to find adjacent elements

Template:

```
def inorder_predecessor(root, node):
    # If left subtree exists, pred is max in left subtree
    if node.left:
        return find_max(node.left)

    # Otherwise, pred is ancestor where node is in right subtree
    pred = None
    while root:
        if node.val > root.val:
            pred = root
            root = root.right
        elif node.val < root.val:
            root = root.left
        else:
            break
    return pred
```

When to Use:

- Finding adjacent elements

- Delete operations
- Range queries

Pattern 6: Two Pointer (Inorder)

Key Concept: Use inorder traversal with two pointers

Template:

```
def two_pointer_inorder(root, target):
    nums = []
    def inorder(node):
        if not node:
            return
        inorder(node.left)
        nums.append(node.val)
        inorder(node.right)

    inorder(root)
    left, right = 0, len(nums) - 1
    # Use two pointers on sorted array
```

When to Use:

- Two sum problems
- Finding pairs
- Range queries with conditions

Pattern 7: Path Sum & Root-to-Leaf

Key Concept: Track path from root to current node

Template:

```
def path_problems(root):
    def dfs(node, path_sum, path):
        if not node:
            return

        path_sum += node.val
        path.append(node.val)

        # Process leaf node
        if not node.left and not node.right:
            # Do something with path_sum or path
            pass

        dfs(node.left, path_sum, path)
        dfs(node.right, path_sum, path)
        path.pop()  # Backtrack

    dfs(root, 0, [])
```

When to Use:

- Path sum problems
- Root to leaf paths
- Ancestor problems

Pattern 8: BST Construction & Modification

Key Concept: Build or modify BST structure

Template:

```
def construct_bst(values):
    root = None
    for val in values:
        root = insert(root, val)
    return root

def modify_bst(root):
    # Perform inorder and modify
    def inorder(node):
        if not node:
            return
        inorder(node.left)
        # Modify node here
        inorder(node.right)
    inorder(root)
    return root
```

When to Use:

- Building BST from array
- Converting structures
- Modifying values

Pattern 9: LCA (Lowest Common Ancestor)

Key Concept: Use BST property to find split point

Template:

```
def lca_bst(root, p, q):
    while root:
        if p.val < root.val and q.val < root.val:
            root = root.left
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            return root
```

When to Use:

- Finding common ancestors

- Path between nodes
- Distance problems

Pattern 10: Morris Traversal (Space Optimized)

Key Concept: Inorder traversal with O(1) space

Template:

```
def morris_inorder(root):
    current = root
    result = []

    while current:
        if not current.left:
            result.append(current.val)
            current = current.right
        else:
            # Find predecessor
            pred = current.left
            while pred.right and pred.right != current:
                pred = pred.right

            if not pred.right:
                pred.right = current
                current = current.left
            else:
                pred.right = None
                result.append(current.val)
                current = current.right

    return result
```

When to Use:

- Space-constrained problems
- Iterative inorder without stack

LeetCode Problems by Pattern

Easy Problems (Foundation Building)

Pattern: Basic Operations & Validation

1. Search in BST (LC 700) - Easy

- Pattern: Basic Search
- Concept: Navigate left/right based on comparison
- Key: Use BST property to eliminate subtrees

2. Insert into BST (LC 701) - Medium

- Pattern: Basic Insert
- Concept: Find position and insert as leaf
- Key: Recursive or iterative insertion

3. Delete Node in BST (LC 450) - Medium

- Pattern: Basic Delete
- Concept: Handle three cases (0, 1, 2 children)
- Key: Find inorder successor for two children case

4. Validate Binary Search Tree (LC 98) - Medium

- Pattern: BST Validation
- Concept: Check range constraints recursively
- Key: Pass min/max bounds through recursion

5. Minimum Absolute Difference in BST (LC 530) - Easy

- Pattern: Inorder Traversal
- Concept: Min difference is between consecutive inorder elements
- Key: Track previous node during inorder

6. Range Sum of BST (LC 938) - Easy

- Pattern: Range Query
- Concept: Prune tree based on range
- Key: Skip subtrees outside range

Medium Problems (Pattern Mastery)

Pattern: Inorder & Sorted Access

7. Kth Smallest Element in BST (LC 230) - Medium

- Pattern: Inorder Kth Element
- Concept: Inorder gives sorted order
- Key: Stop at kth element or use iterative with counter

8. Binary Search Tree Iterator (LC 173) - Medium

- Pattern: Controlled Inorder
- Concept: Simulate inorder with stack
- Key: Maintain stack of left spine

9. Two Sum IV - Input is BST (LC 653) - Easy

- Pattern: Two Pointer on Inorder
- Concept: Get sorted array, use two pointers
- Key: Or use set while traversing

10. Convert Sorted Array to BST (LC 108) - Easy

- Pattern: Array to BST Construction
- Concept: Middle element as root for balance
- Key: Recursively build left and right subtrees

11. Convert Sorted List to BST (LC 109) - Medium

- Pattern: List to BST Construction
- Concept: Find middle using slow/fast pointers
- Key: Or convert to array first

12. Recover Binary Search Tree (LC 99) - Medium

- Pattern: Inorder Anomaly Detection
- Concept: Find two swapped nodes in inorder
- Key: Track first and second violation

Pattern: BST Properties & Range

13. Trim a Binary Search Tree (LC 669) - Medium

- Pattern: Range Filtering
- Concept: Remove nodes outside range
- Key: Return null or appropriate subtree

14. Closest Binary Search Tree Value (LC 270) - Easy

- Pattern: Optimized Search
- Concept: Navigate toward target
- Key: Track closest seen so far

15. Closest Binary Search Tree Value II (LC 272) - Hard

- Pattern: K Closest with Inorder
- Concept: Use inorder and sliding window
- Key: Or use priority queue

16. Count Complete Tree Nodes (LC 222) - Medium

- Pattern: Complete Tree Property
- Concept: Binary search on completeness
- Key: Check left/right height for perfect subtree

Pattern: Construction & Conversion

17. Construct BST from Preorder (LC 1008) - Medium

- Pattern: Preorder Construction
- Concept: First element is root
- Key: Use range bounds to determine subtree

18. Construct BST from Inorder and Preorder (LC 105) - Medium

- Pattern: Multiple Traversal Construction
- Concept: Preorder gives roots, inorder gives split
- Key: Use hashmap for inorder indices

19. Flatten BST to Linked List (LC 114) - Medium

- Pattern: Structure Conversion
- Concept: Right-skewed tree (preorder)
- Key: Use Morris or recursion with prev pointer

20. Increasing Order Search Tree (LC 897) - Easy

- Pattern: Structure Conversion
- Concept: Right-skewed tree (inorder)
- Key: Build new tree during inorder

21. Balance a BST (LC 1382) - Medium

- Pattern: BST to Array to Balanced BST
- Concept: Inorder → sorted array → build balanced
- Key: Use middle element recursively

Pattern: LCA & Path

22. Lowest Common Ancestor of BST (LC 235) - Easy

- Pattern: BST LCA
- Concept: Find split point
- Key: Both nodes on different sides

23. Path Sum (LC 112) - Easy

- Pattern: Root to Leaf Path
- Concept: DFS with sum tracking
- Key: Check at leaf nodes

24. Path Sum II (LC 113) - Medium

- Pattern: Root to Leaf Paths Collection
- Concept: DFS with path tracking and backtracking
- Key: Copy path when adding to result

25. Sum Root to Leaf Numbers (LC 129) - Medium

- Pattern: Path Number Calculation
- Concept: Build number during traversal
- Key: number = number * 10 + node.val

Hard Problems (Advanced Mastery)

26. Serialize and Deserialize BST (LC 449) - Medium

- Pattern: BST Serialization
- Concept: Use preorder (or level order)
- Key: BST property allows reconstruction without nulls

27. Count of Smaller Numbers After Self (LC 315) - Hard

- Pattern: BST with Count
- Concept: Build BST right to left, count smaller on insert
- Key: Augment tree with subtree sizes

28. Contains Duplicate III (LC 220) - Hard

- Pattern: BST for Range Query
- Concept: Maintain sliding window in BST
- Key: Use TreeSet/TreeMap for ceiling/floor

29. Maximum Sum BST in Binary Tree (LC 1373) - Hard

- Pattern: Bottom-up BST Validation + Sum
- Concept: Return min, max, sum, isBST from subtrees
- Key: Update global max when valid BST found

30. Count of Range Sum (LC 327) - Hard

- Pattern: BST with Prefix Sums
- Concept: Use merge sort or BST for range counting
- Key: Count prefix sums in range

Special Patterns

31. Unique BSTs (LC 96) - Medium

- Pattern: Catalan Number / DP
- Concept: Count structurally unique BSTs
- Key: dp[i] = sum(dp[j] * dp[i-j-1]) for each root

32. Unique BSTs II (LC 95) - Medium

- Pattern: Generate All BSTs
- Concept: Generate all structurally unique BSTs
- Key: For each root, combine all left and right subtrees

33. Inorder Successor in BST (LC 285) - Medium

- Pattern: Successor Finding
- Concept: If right child exists, return min of right subtree
- Key: Else find ancestor where node is in left subtree

34. Inorder Successor in BST II (LC 510) - Medium

- Pattern: Successor with Parent Pointer
- Concept: Navigate using parent pointers
- Key: Go up until you're in left subtree

35. Convert BST to Greater Tree (LC 538) - Medium

- Pattern: Reverse Inorder with Accumulation
- Concept: Traverse right to left, accumulate sums
- Key: right → root (update) → left

36. Binary Search Tree to Greater Sum Tree (LC 1038) - Medium

- Pattern: Same as LC 538
- Concept: Reverse inorder accumulation
- Key: Maintain running sum

Advanced Topics

37. Verify Preorder Sequence in BST (LC 255) - Medium

- Pattern: Stack-based Validation
- Concept: Simulate BST construction
- Key: Use stack and lower bound

38. Find Mode in BST (LC 501) - Easy

- Pattern: Inorder with Frequency Tracking
- Concept: Count consecutive values in inorder
- Key: Track current value count

39. Minimum Distance Between BST Nodes (LC 783) - Easy

- Pattern: Inorder Consecutive Difference
- Concept: Same as LC 530
- Key: Track previous value

40. All Elements in Two BSTs (LC 1305) - Medium

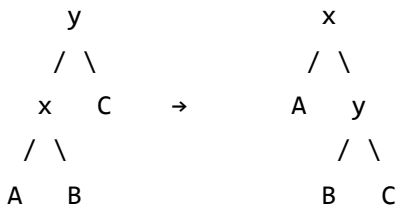
- Pattern: Merge Two Sorted Lists
- Concept: Inorder both trees, merge
- Key: Can optimize with iterators

Advanced Concepts

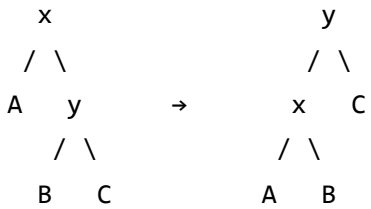
1. Self-Balancing BSTs

AVL Tree Rotations

Single Right Rotation (LL Case):



Single Left Rotation (RR Case):



Red-Black Tree Properties

1. Every node is red or black
2. Root is always black
3. All leaves (NIL) are black
4. Red nodes have black children
5. All paths from root to leaves have same number of black nodes

2. Threaded Binary Trees

Concept: Use null pointers to store predecessor/successor

- Right-threaded: null right pointers point to inorder successor
- Left-threaded: null left pointers point to inorder predecessor
- Advantage: O(1) space traversal

3. Augmented BST

Idea: Store additional information at each node

- **Order Statistics Tree**: Store subtree size → find kth element in O(log n)
- **Interval Tree**: Store max endpoint → find overlapping intervals
- **Range Tree**: Store range sum → answer range queries

4. Persistent BST

Concept: Maintain all versions of tree after modifications

- Path copying technique
- Each modification creates new version
- Space: O(log n) per modification

5. BST Optimization Techniques

Splay Operations

- Recently accessed elements moved to root
- Amortized O(log n) operations

Weight Balanced Trees

- Balance based on subtree sizes rather than heights
- Good for persistent structures

Practice Roadmap

Week 1-2: Foundation (Easy)

Focus: Basic operations and traversals

1. LC 700 - Search in BST
2. LC 938 - Range Sum of BST
3. LC 530 - Minimum Absolute Difference in BST
4. LC 653 - Two Sum IV - Input is BST
5. LC 108 - Convert Sorted Array to BST

6. LC 897 - Increasing Order Search Tree
7. LC 235 - Lowest Common Ancestor of BST
8. LC 501 - Find Mode in BST
9. LC 783 - Minimum Distance Between BST Nodes

Goal: Comfortable with BST traversal and basic operations

Week 3-4: Pattern Recognition (Easy-Medium)

Focus: Common patterns and validation

10. LC 98 - Validate Binary Search Tree
11. LC 230 - Kth Smallest Element in BST
12. LC 669 - Trim a Binary Search Tree
13. LC 270 - Closest Binary Search Tree Value
14. LC 112 - Path Sum
15. LC 113 - Path Sum II
16. LC 129 - Sum Root to Leaf Numbers
17. LC 1305 - All Elements in Two BSTs

Goal: Recognize and apply common BST patterns

Week 5-6: Advanced Operations (Medium)

Focus: Modification and construction

18. LC 701 - Insert into BST
19. LC 450 - Delete Node in BST
20. LC 99 - Recover Binary Search Tree
21. LC 173 - Binary Search Tree Iterator
22. LC 109 - Convert Sorted List to BST
23. LC 1008 - Construct BST from Preorder
24. LC 1382 - Balance a BST
25. LC 538 - Convert BST to Greater Tree

Goal: Master BST construction and modification

Week 7-8: Complex Problems (Medium-Hard)

Focus: Multiple patterns and optimizations

26. LC 449 - Serialize and Deserialize BST
27. LC 96 - Unique Binary Search Trees
28. LC 95 - Unique Binary Search Trees II
29. LC 285 - Inorder Successor in BST
30. LC 272 - Closest Binary Search Tree Value II
31. LC 315 - Count of Smaller Numbers After Self
32. LC 1373 - Maximum Sum BST in Binary Tree

Goal: Solve complex problems combining multiple concepts

Daily Practice Tips

1. **Morning (30 min)**: Review one pattern/concept
2. **Afternoon (1 hour)**: Solve 2-3 problems
3. **Evening (30 min)**: Review solutions and note patterns

Key Success Metrics

After completing this roadmap, you should be able to:

- Solve Easy BST problems in under 10 minutes
- Solve Medium BST problems in under 25 minutes
- Recognize patterns immediately
- Implement BST operations without reference
- Optimize solutions using BST properties

Quick Reference Cheat Sheet

When to Use Each Traversal

Traversal	Use Case	Result
Inorder	Get sorted elements, validate BST	Sorted sequence
Preorder	Copy tree, serialize, prefix expression	Root-first
Postorder	Delete tree, postfix expression	Children-first
Level Order	BFS, level processing	Level by level

Common BST Time Complexities

Operation	Balanced	Unbalanced
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)
Min/Max	O(log n)	O(n)
Successor	O(log n)	O(n)
Range Query	O(log n + k)	O(n)

Pattern Recognition Flowchart

1. **Need sorted order?** → Use inorder traversal
2. **Finding kth element?** → Inorder or augmented BST
3. **Range query?** → Prune using BST property
4. **Validating BST?** → Use min/max bounds
5. **Path problem?** → DFS with tracking
6. **Two nodes relationship?** → Find LCA
7. **Building BST?** → Use middle element or range bounds
8. **Modifying structure?** → Consider balanced BST

Common Mistakes to Avoid

1. Forgetting to check null nodes
2. Not maintaining BST property during modifications
3. Using wrong traversal for the problem
4. Not considering edge cases (single node, skewed tree)
5. Forgetting to handle duplicate values
6. Not optimizing using BST property (treating as regular tree)
7. Incorrect range bounds in validation
8. Not backtracking in path problems

Conclusion

Mastering BST requires:

1. **Understanding**: Core properties and operations
2. **Pattern Recognition**: Identify which pattern to apply
3. **Practice**: Solve problems systematically
4. **Optimization**: Use BST properties to optimize solutions

Remember: Every BST problem can be solved by recognizing it as a variation of one of the 10 core patterns. Focus on understanding why each pattern works, not just memorizing solutions.

Next Steps:

1. Complete the 8-week roadmap systematically
2. Revisit difficult problems after 2-3 days
3. Implement operations from scratch without reference
4. Move to balanced BST variants (AVL, Red-Black) after mastering basic BST
5. Practice explaining solutions to solidify understanding

Good luck on your journey to BST mastery!