Binary Trees

A Practical Exercise Guide

10 Progressive Exercises with Hints & Complexity Analysis

Table of Contents

Overview

Binary trees are hierarchical data structures where each node has at most two children, referred to as left and right. They form the foundation for many advanced data structures including binary search trees (BSTs), heaps, and balanced trees like AVL and Red-Black trees.

A **Binary Search Tree (BST)** adds an ordering property: for any node, all values in its left subtree are smaller, and all values in its right subtree are larger. This enables efficient O(log n) search, insert, and delete operations when the tree is balanced.

Sample BST used throughout exercises:

```
        5
      /   \
     3     8
    / \   / \
   1   4 7   9
```

Exercise 1: TreeNode Class & Count Nodes

Objective: Implement the fundamental building block of a binary tree.

Task:

Create a Node class with three attributes: value, left_child, and right_child. Then write a count_nodes() method that returns the total number of nodes in the tree.

Test Your Solution:

Build this tree manually and verify count_nodes() returns 4:

```
     5
    / \
   3   8
  /
 1
```

Hints:

- ■ *Hint 1:* The __init__ method needs 'self' as the first parameter and should accept a 'value' parameter.
- ■ *Hint 2:* Initialize left_child and right_child to None.
- ■ *Hint 3:* For count_nodes, think recursively: count = 1 + count(left) + count(right).
- ■ *Hint 4:* Base case: what should you return when a node is None? (Think: how many nodes in an empty tree?)

Time Complexity: O(n) — must visit every node once

Space Complexity: O(h) — recursive call stack depth equals tree height

Exercise 2: Inorder Traversal

Objective: Learn depth-first traversal that visits nodes in sorted order for BSTs.

Task:

Implement inorder_traversal() that prints node values in the order: left subtree → current node → right subtree.

Test Your Solution:

For the sample BST, output should be: 1, 3, 4, 5, 7, 8, 9 (sorted order!)

Hints:

- *Hint 1:* The order of operations matters! Process left child BEFORE printing current value.
- *Hint 2:* Check if child exists before recursing to avoid errors.
- *Hint 3:* The structure is: recurse left → print self.value → recurse right
- *Hint 4:* Remember: Inorder traversal of a BST always produces sorted output. Use this to verify correctness.

Time Complexity: O(n) — visits every node exactly once

Space Complexity: O(h) — call stack depth

Key Insight:

Inorder traversal of a BST always produces values in ascending sorted order. This property is used to verify BST validity and for efficient sorted iteration.

Exercise 3: Preorder & Postorder Traversal

Objective: Complete the three classic DFS traversal patterns.

Task:

Implement both traversal methods:

• **preorder_traversal():** current → left → right
• **postorder_traversal():** left → right → current

Test Your Solution:

For the sample BST:
• Preorder output: 5, 3, 1, 4, 8, 7, 9
• Postorder output: 1, 4, 3, 7, 9, 8, 5

Hints:

■ *Hint 1:* The only difference between traversals is WHEN you print the current value.

■ *Hint 2:* Pre = print before recursing, In = print between, Post = print after both recurse calls.

■ *Hint 3:* Make sure recursive calls use the SAME method name (don't accidentally call inorder from preorder).

■ *Hint 4:* Preorder visits root first (good for copying). Postorder visits root last (good for deleting).

Time Complexity: O(n) for both

Use Cases:

• **Preorder:** Creating a copy of the tree, serialization
• **Postorder:** Deleting a tree (children before parent), expression evaluation
• **Inorder:** Sorted traversal of BST

Exercise 4: Tree Height

Objective: Calculate the longest path from root to any leaf.

Task:

Implement height() that returns the number of edges on the longest root-to-leaf path. A single node with no children has height 0.

Test Your Solution:

The sample BST has height 2 (path: 5 → 3 → 1 or 5 → 3 → 4, etc.)

Hints:

- *Hint 1:* Height = 1 + max(left_height, right_height)
- *Hint 2:* Initialize both left_height and right_height to 0 before checking if children exist.
- *Hint 3:* A leaf node (no children) should return 0 — both children are None, so max(0, 0) = 0.
- *Hint 4:* Only add 1 when recursing into an existing child, not for None children.

Time Complexity: O(n) — must check all nodes to find deepest

Space Complexity: O(h) — recursive depth

Why Height Matters:

Tree height determines the worst-case time complexity of operations. A balanced tree has height O(log n), giving O(log n) operations. A skewed tree has height O(n), degrading to O(n) operations.

Exercise 5: BST Insert

Objective: Build a BST by inserting values in their correct positions.

Task:

Implement insert(value) that adds a new node while maintaining BST property: left subtree values < node value < right subtree values.

Test Your Solution:

```
tree = Node(5)
tree.insert(3)
tree.insert(8)
tree.insert(1)
tree.insert(4)
tree.insert(7)
tree.insert(9)
tree.inorder_traversal()  # Should print: 1,3,4,5,7,8,9
```

Hints:

- ■ *Hint 1:* Compare value with self.value to decide direction (left if smaller, right if larger).
- ■ *Hint 2:* If the appropriate child is None, create a new Node there.
- ■ *Hint 3:* If the appropriate child exists, recursively call insert on that child.
- ■ *Hint 4:* Verify correctness by running inorder_traversal — values should print in sorted order.

Time Complexity: O(h) — follows one path from root to insertion point

Best case (balanced): O(log n) | **Worst case (skewed):** O(n)

Exercise 6: BST Search

Objective: Efficiently find values using BST property.

Task:

Implement search(value) that returns True if value exists, False otherwise. Use BST property to eliminate half the tree at each step.

Test Your Solution:

```
print(tree.search(4))   # True
print(tree.search(7))   # True
print(tree.search(10))  # False
print(tree.search(2))   # False
```

Hints:

- *Hint 1:* If value == self.value, you found it — return True.
- *Hint 2:* If value < self.value, search left; otherwise search right.
- *Hint 3:* Check if child is None before recursing — if so, the value doesn't exist, return False.
- *Hint 4:* Don't forget to RETURN the result of recursive calls! Just calling search() without return loses the result.

Time Complexity: O(h) — same as insert

Comparison to Linear Search: Array search is O(n). BST search is O(log n) when balanced.

Exercise 7: Level-Order Traversal (BFS)

Objective: Visit nodes breadth-first using a queue.

Task:

Implement level_order_traversal() that prints nodes level by level, left to right.

Test Your Solution:

For the sample BST, output should be: 5, 3, 8, 1, 4, 7, 9

Hints:

- *Hint 1:* This CANNOT be done with simple recursion — you need a queue data structure.
- *Hint 2:* Algorithm: Start with root in queue. While queue not empty: dequeue front node, print it, enqueue its children.
- *Hint 3:* Use a Python list as queue: list.append() to add to back, list.pop(0) to remove from front.
- *Hint 4:* This is an iterative algorithm using a while loop, not recursive.

Time Complexity: O(n) — visits every node once

Space Complexity: O(w) where w is maximum width of tree (up to n/2 at last level)

DFS vs BFS:

• DFS (recursion/stack): Goes deep before wide. Uses O(h) space.
• BFS (queue): Goes wide before deep. Uses O(w) space.
Choose based on tree shape and what you're searching for.

Exercise 8: Find Min & Max

Objective: Exploit BST structure for O(h) min/max finding.

Task:

Implement find_min() and find_max() using the BST property. Don't traverse the whole tree — use the structure!

Test Your Solution:

```
print(tree.find_min())  # 1
print(tree.find_max())  # 9
```

Hints:

- *Hint 1:* In a BST, the minimum is always the LEFTMOST node.
- *Hint 2:* The maximum is always the RIGHTMOST node.
- *Hint 3:* Keep going in one direction until you can't anymore (child is None).
- *Hint 4:* Can be done iteratively (while loop) or recursively. For recursive: if no left child, return self.value; else recurse left.

Time Complexity: O(h) — just follows one path down the edge

Note: A naive approach traversing all nodes would be O(n). The BST property gives us O(h).

Exercise 9: Validate BST

Objective: Verify a tree satisfies the BST property throughout.

Task:

Implement is_valid_bst() that returns True if the tree is a valid BST, False otherwise. Remember: ALL nodes in left subtree must be less than current, not just the immediate child.

Test Your Solution:

```
print(tree.is_valid_bst())  # True
tree.left_child.right_child.value = 100  # break it
print(tree.is_valid_bst())  # False
```

Hints:

■ *Hint 1:* Checking only immediate children is NOT enough! A node deep in left subtree could be larger than root.

■ *Hint 2:* Pass down min/max bounds as you recurse. Each node must be within (min_bound, max_bound).

■ *Hint 3:* Use a helper function: helper(node, min_val, max_val) that returns True/False.

■ *Hint 4:* Start with bounds of (-infinity, +infinity). Going left tightens upper bound; going right tightens lower bound.

Time Complexity: O(n) — must check every node

Common Mistake:

Only checking node.left.value < node.value is WRONG. Consider: root=5, root.left=3, root.left.right=10. The 10 passes the local check (10 > 3) but violates the BST property (10 > 5, but it's in 5's left subtree). The bounds approach catches this.

Exercise 10: Delete Node

Objective: The most complex BST operation — remove a node while maintaining order.

Task:

Implement delete(value) that removes a node and returns the modified tree root. Handle three cases.

The Three Cases:

Case 1 - Leaf (no children): Simply remove it by returning None.

Case 2 - One child: Replace node with its only child (return that child).

Case 3 - Two children: Find the inorder successor (smallest value in right subtree), copy its value to current node, then delete the successor from the right subtree.

Test Your Solution:

```
tree.inorder_traversal()  # 1,3,4,5,7,8,9
tree = tree.delete(3)
tree.inorder_traversal()  # 1,4,5,7,8,9
```

Hints:

- *Hint 1:* Structure: if value < self.value, recurse left and update self.left_child with result. Same for right.
- *Hint 2:* When you find the node (value == self.value), handle the three cases.
- *Hint 3:* For two children: the inorder successor is found by going right once, then left as far as possible.
- *Hint 4:* After copying successor's value to current node, delete the successor from right subtree recursively.
- *Hint 5:* The successor is guaranteed to have at most one child (no left child), so deleting it is Case 1 or 2.

Time Complexity: O(h) — find node + find successor + delete successor

Summary & Review

Core Concepts Covered:

1. **Tree Structure:** Nodes with value, left, and right pointers
2. **Traversals:** Inorder, Preorder, Postorder (DFS) and Level-order (BFS)
3. **BST Property:** Left < Node < Right for all subtrees
4. **BST Operations:** Insert, Search, Delete, Find Min/Max
5. **Validation:** Using bounds to verify BST property

Traversal Memory Aid:

• **PRE**order: **P**rocess current, then children (root first)
• **IN**order: Left, process current **IN** middle, Right (sorted for BST)
• **POST**order: Children first, process current **POST** (root last)

When to Use Each Traversal:

• **Inorder:** Get sorted sequence, BST validation
• **Preorder:** Copy tree, serialize tree structure
• **Postorder:** Delete tree, evaluate expressions
• **Level-order:** Find shortest path, level-by-level processing

Key Patterns:

1. **Recursive tree functions:** Handle base case (None), then recurse on children
2. **BST navigation:** Compare value, go left if smaller, right if larger
3. **Passing state down:** Use helper functions with additional parameters (like bounds)
4. **BFS with queue:** Initialize with root, process front, add children to back

Time Complexity Reference

Operation	Average Case	Worst Case	Notes
count_nodes()	O(n)	O(n)	Must visit all nodes
Traversals (all)	O(n)	O(n)	Visit all nodes once
height()	O(n)	O(n)	Must find deepest path
insert()	O(log n)	O(n)	Follows one path down
search()	O(log n)	O(n)	Follows one path down
find_min/max()	O(log n)	O(n)	Follows left/right edge
is_valid_bst()	O(n)	O(n)	Must check all nodes
delete()	O(log n)	O(n)	Find + successor ops

Why Worst Case is O(n):

The worst case occurs when the tree degenerates into a linked list (completely unbalanced). This happens when you insert sorted data: [1, 2, 3, 4, 5] creates a right-skewed tree with height n-1.

Achieving O(log n) Consistently:

Use self-balancing trees (AVL, Red-Black) that maintain height ≈ log(n) after every operation. These guarantee O(log n) for insert, search, and delete.

Space Complexity Notes:

• Recursive operations: O(h) space for call stack
• Level-order traversal: O(w) space for queue, where w is max width
• For balanced tree: h = O(log n), w = O(n/2) at bottom level
• For skewed tree: h = O(n), w = O(1)