The Ultimate Tree Data Structures & Algorithms Handbook

Complete Visual Guide for Technical Interviews & Competitive Programming

 Table of Contents

Chapter 1: Tree Fundamentals & Deep Theory

[201]

1.1 What Are Trees? The Mathematical Foundation

Formal Definition

A **tree** is a connected, acyclic undirected graph. More formally:

- **T = (V, E)** where V is a set of vertices (nodes) and E is a set of edges
- **|E| = |V| - 1** (exactly n-1 edges for n vertices)
- **Connected**: There exists a path between any two vertices
- **Acyclic**: Contains no cycles

Tree Properties (Theoretical)

Property 1: Unique Path
For any two nodes u and v in a tree, there exists exactly one simple path connecting them.

Property 2: Adding Edge Creates Cycle
Adding any edge to a tree creates exactly one cycle.

Property 3: Removal Disconnects
Removing any edge from a tree disconnects it into exactly two components.

Property 4: Minimal Connected Graph
A tree is the minimal connected graph - removing any edge disconnects it.

Tree Terminology (Complete)

[210]

Term	Definition	Mathematical Notation	Example
Root	Node with no parent	$r \in V$, $\text{parent}(r) = \varnothing$	Top node in hierarchy
Leaf	Node with no children	$v \in V$, $\text{children}(v) = \varnothing$	Terminal nodes
Height	Max distance from node to any leaf	$h(v) = \max\{d(v,u) : u \text{ is leaf}\}$	Longest path down
Depth	Distance from root to node	$\text{depth}(v) = d(\text{root}, v)$	Level of node
Degree	Number of children	$\deg(v) = \lvert\text{children}(v)\rvert$	Branching factor
Level	Set of nodes at same depth	$L_k = \{v : \text{depth}(v) = k\}$	Horizontal layer
Subtree	Tree rooted at node v	$T_v = (V', E')$ induced by v	Portion below node
Ancestor	Node on path from root	u ancestor of v if u on path(root,v)	Above in hierarchy
Descendant	Node in subtree	v descendant of u if $v \in T_u$	Below in hierarchy

Tree Invariants and Properties

Size-Height Relationship

For a tree with n nodes:

- **Minimum height**: $\lfloor \log_2(n) \rfloor$ (complete binary tree)
- **Maximum height**: n-1 (skewed tree)
- **Average height**: $O(\sqrt{n})$ for random trees

Mathematical Bounds

Theorem 1: Catalan Numbers
Number of structurally different binary trees with n nodes = C_n = (2n)!/(n!(n+1)!)

Theorem 2: Tree Traversal Count
For any tree with n nodes, each traversal visits exactly n nodes in O(n) time.

Chapter 2: Mathematical Foundation & Complexity Theory

2.1 Tree Mathematics

Recurrence Relations for Trees

Height Calculation:

```
h(node) = 1 + max(h(left), h(right))
h(null) = -1
```

Node Count:

```
count(node) = 1 + count(left) + count(right)
count(null) = 0
```

Perfect Binary Tree Properties:

- Nodes at level k: 2^k
- Total nodes in tree of height h: 2^(h+1) - 1
- Number of leaves: 2^h
- Number of internal nodes: 2^h - 1

Asymptotic Analysis

Tree Operation Complexities

Tree Type	Search	Insert	Delete	Space	Notes
Binary Tree	O(n)	O(n)	O(n)	O(n)	Worst case skewed
BST (Average)	O(log n)	O(log n)	O(log n)	O(n)	Balanced case
BST (Worst)	O(n)	O(n)	O(n)	O(n)	Skewed tree
AVL Tree	O(log n)	O(log n)	O(log n)	O(n)	Guaranteed balanced
Red-Black	O(log n)	O(log n)	O(log n)	O(n)	Guaranteed balanced
Complete Tree	O(n)	O(1)	O(log n)	O(n)	Array representation

Probability and Expected Values

Random BST Analysis:

- Expected height of random BST: O(log n)
- Probability of height > c log n: O(1/n^c)
- Expected search cost: O(log n)

Chapter 3: Complete Tree Classifications

[210]

3.1 Binary Tree Types (Visual Classification)

Full Binary Tree

Definition: Every node has either 0 or 2 children (never 1)

Properties:

- Number of leaves = Number of internal nodes + 1
- If L = leaves, I = internal nodes, then L = I + 1
- Total nodes = 2L - 1

Python Implementation:

```
def is_full_binary_tree(root):
    """Check if tree is full binary tree"""
    if not root:
        return True

    # If only one child exists, not full
    if bool(root.left) ^ bool(root.right):  # XOR
        return False

    return is_full_binary_tree(root.left) and is_full_binary_tree(root.right)
```

Complete Binary Tree

Definition: All levels filled except possibly the last, which is filled left-to-right

[202]

Properties:

- Height = $\lfloor \log_2(n) \rfloor$
- Can be efficiently stored in array
- Parent of node i: (i-1)/2
- Left child of node i: 2i+1
- Right child of node i: 2i+2

Array Representation:

```
class CompleteBinaryTree:
    def __init__(self):
        self.tree = []

    def parent(self, i):
        return (i - 1) // 2 if i &gt; 0 else None

    def left_child(self, i):
        left = 2 * i + 1
        return left if left &lt; len(self.tree) else None

    def right_child(self, i):
        right = 2 * i + 2
        return right if right &lt; len(self.tree) else None

    def insert(self, val):
        """Insert maintains complete tree property"""
        self.tree.append(val)

    def get_height(self):
        """Height calculation for complete tree"""
        n = len(self.tree)
        return int(math.log2(n)) if n &gt; 0 else -1
```

Perfect Binary Tree

Definition: All internal nodes have 2 children, all leaves at same level

Properties:

- Total nodes = 2^(h+1) - 1
- Leaves = 2^h
- Internal nodes = 2^h - 1
- Extremely rare in practice

```
def is_perfect_binary_tree(root):
    """Check if tree is perfect"""
    def get_depth(node):
        depth = 0
        while node:
            depth += 1
            node = node.left
        return depth

    def is_perfect_recursive(node, depth, current_depth=0):
        if not node:
            return current_depth == depth

        if not node.left and not node.right:
            return current_depth == depth - 1
```

```
        if not node.left or not node.right:
            return False

        return (is_perfect_recursive(node.left, depth, current_depth + 1) and
                is_perfect_recursive(node.right, depth, current_depth + 1))

    depth = get_depth(root)
    return is_perfect_recursive(root, depth)
```

Balanced Binary Tree

Definition: Height difference between left and right subtrees ≤ 1

[205]

Mathematical Definition:
For every node v: $|\text{height(left(v)) - height(right(v))}| \leq 1$

```
def is_balanced_detailed(root):
    """Detailed balance checking with height tracking"""
    def check_balance(node):
        if not node:
            return True, -1  # (is_balanced, height)

        left_balanced, left_height = check_balance(node.left)
        if not left_balanced:
            return False, 0

        right_balanced, right_height = check_balance(node.right)
        if not right_balanced:
            return False, 0

        current_height = 1 + max(left_height, right_height)
        is_balanced = abs(left_height - right_height) &lt;= 1

        return is_balanced, current_height

    balanced, _ = check_balance(root)
    return balanced
```

Degenerate (Skewed) Tree

Definition: Each parent has only one child (essentially a linked list)

Properties:

- Height = n - 1
- Performance degrades to O(n) for all operations
- Space efficiency poor due to pointer overhead

Chapter 4: Tree Traversal Algorithms (Complete Visual Guide)

[206]

4.1 Depth-First Search (DFS) Traversals

Inorder Traversal (Left → Root → Right)

Theory: Visits left subtree, then root, then right subtree

Key Applications:

- Gets sorted sequence from BST
- Expression evaluation (infix expressions)
- Tree flattening

Recursive Implementation (Detailed)

```
def inorder_traversal_detailed(root):
    """
    Inorder traversal with step-by-step explanation
    Time Complexity: O(n) - visits each node exactly once
    Space Complexity: O(h) - maximum recursion depth equals tree height
    """
    result = []
    call_stack = []  # For debugging/visualization

    def inorder_recursive(node, depth=0):
        call_stack.append(f"{'  ' * depth}Visiting node: {node.val if node else 'None'}")

        if not node:
            call_stack.append(f"{'  ' * depth}Base case: None node, returning")
            return

        call_stack.append(f"{'  ' * depth}Going left from {node.val}")
        inorder_recursive(node.left, depth + 1)

        call_stack.append(f"{'  ' * depth}Processing root: {node.val}")
        result.append(node.val)

        call_stack.append(f"{'  ' * depth}Going right from {node.val}")
        inorder_recursive(node.right, depth + 1)

        call_stack.append(f"{'  ' * depth}Finished with node: {node.val}")

    inorder_recursive(root)
    return result, call_stack
```

Iterative Implementation (Stack-Based)

```
def inorder_iterative_detailed(root):
    """
    Iterative inorder using explicit stack
    Mimics recursion call stack manually
    """
    result = []
    stack = []
    current = root
    steps = []  # For visualization

    while stack or current:
        # Go to leftmost node
        while current:
            steps.append(f"Pushing {current.val} to stack, going left")
            stack.append(current)
            current = current.left

        # Current is None, backtrack
        if stack:
            current = stack.pop()
            steps.append(f"Popped {current.val}, processing it")
            result.append(current.val)

            steps.append(f"Moving to right subtree of {current.val}")
            current = current.right

    return result, steps
```

Morris Inorder (O(1) Space)

```
def morris_inorder(root):
    """
    Morris traversal: O(1) space using threading
    Modifies tree temporarily by creating threads
    """
    result = []
    current = root
    steps = []

    while current:
        if not current.left:
            # No left subtree, process current and go right
            steps.append(f"No left child for {current.val}, processing it")
            result.append(current.val)
            current = current.right
        else:
            # Find inorder predecessor
            predecessor = current.left
            while predecessor.right and predecessor.right != current:
                predecessor = predecessor.right

            if not predecessor.right:
```

```
                # Create thread
                steps.append(f"Creating thread from {predecessor.val} to {current.val}")
                predecessor.right = current
                current = current.left
            else:
                # Remove thread and process current
                steps.append(f"Removing thread, processing {current.val}")
                predecessor.right = None
                result.append(current.val)
                current = current.right

    return result, steps
```

Preorder Traversal (Root → Left → Right)

Applications:

- Tree copying/cloning
- Prefix expression evaluation
- Tree serialization

```
def preorder_applications():
    """Demonstrate preorder applications"""

    def clone_tree(root):
        """Clone tree using preorder traversal"""
        if not root:
            return None

        # Process root first (preorder characteristic)
        new_node = TreeNode(root.val)
        new_node.left = clone_tree(root.left)
        new_node.right = clone_tree(root.right)
        return new_node

    def serialize_preorder(root):
        """Serialize tree using preorder"""
        if not root:
            return "null"

        # Root first, then left, then right
        return f"{root.val},{serialize_preorder(root.left)},{serialize_preorder(root.right)

    def evaluate_prefix_expression(expression):
        """Evaluate prefix expression using tree"""
        tokens = expression.split()
        index = [0]  # Use list for reference passing

        def build_expression_tree():
            token = tokens[index[0]]
            index[0] += 1

            if token in ['+', '-', '*', '/']:
                node = TreeNode(token)
```

```python
            node.left = build_expression_tree()  # Left operand
            node.right = build_expression_tree() # Right operand
            return node
        else:
            return TreeNode(int(token))

    def evaluate_tree(root):
        if not root:
            return 0

        if root.val not in ['+', '-', '*', '/']:
            return root.val

        left_val = evaluate_tree(root.left)
        right_val = evaluate_tree(root.right)

        if root.val == '+':
            return left_val + right_val
        elif root.val == '-':
            return left_val - right_val
        elif root.val == '*':
            return left_val * right_val
        elif root.val == '/':
            return left_val / right_val

    tree = build_expression_tree()
    return evaluate_tree(tree)

return clone_tree, serialize_preorder, evaluate_prefix_expression
```

Postorder Traversal (Left → Right → Root)

Applications:

- Tree deletion (delete children before parent)
- Postfix expression evaluation
- Directory size calculation
- Dependency resolution

```python
def postorder_applications():
    """Demonstrate postorder applications"""

    def delete_tree(root):
        """Safely delete entire tree using postorder"""
        if not root:
            return None

        # Delete children first
        delete_tree(root.left)
        delete_tree(root.right)

        # Then delete root
        print(f"Deleting node: {root.val}")
```

```
        # In actual implementation: free(root)
        return None

    def calculate_directory_size(root):
        """Calculate directory sizes using postorder"""
        if not root:
            return 0

        # Calculate sizes of subdirectories first
        left_size = calculate_directory_size(root.left)
        right_size = calculate_directory_size(root.right)

        # Then process current directory
        current_size = root.val  # Assume val is file size
        total_size = current_size + left_size + right_size

        print(f"Directory {root.val}: {total_size} bytes")
        return total_size

    def evaluate_postfix_expression_tree(root):
        """Evaluate postfix expression tree"""
        if not root:
            return 0

        # If leaf node (operand)
        if not root.left and not root.right:
            return root.val

        # Evaluate children first
        left_val = evaluate_postfix_expression_tree(root.left)
        right_val = evaluate_postfix_expression_tree(root.right)

        # Then apply operator
        if root.val == '+':
            return left_val + right_val
        elif root.val == '-':
            return left_val - right_val
        elif root.val == '*':
            return left_val * right_val
        elif root.val == '/':
            return left_val / right_val

    return delete_tree, calculate_directory_size, evaluate_postfix_expression_tree
```

4.2 Breadth-First Search (Level Order)

Theory: Visit nodes level by level, left to right

Applications:

- Find shortest path in unweighted tree
- Level-wise processing
- Tree width calculation
- Serialization for complete trees

Standard Level Order

```
def level_order_comprehensive(root):
    """
    Comprehensive level order implementation with multiple variants
    """
    from collections import deque

    if not root:
        return []

    # Variant 1: Simple level order
    def simple_level_order():
        result = []
        queue = deque([root])

        while queue:
            node = queue.popleft()
            result.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        return result

    # Variant 2: Level order with level separation
    def level_order_by_levels():
        result = []
        queue = deque([root])

        while queue:
            level_size = len(queue)
            current_level = []

            for _ in range(level_size):
                node = queue.popleft()
                current_level.append(node.val)

                if node.left:
                    queue.append(node.left)
                if node.right:
                    queue.append(node.right)

            result.append(current_level)

        return result

    # Variant 3: Right to left level order
    def level_order_right_to_left():
        result = []
        queue = deque([root])

        while queue:
            level_size = len(queue)
```

```python
        current_level = []

        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            # Add right child first for right-to-left
            if node.right:
                queue.append(node.right)
            if node.left:
                queue.append(node.left)

        result.append(current_level)

    return result

# Variant 4: Zigzag level order
def zigzag_level_order():
    result = []
    queue = deque([root])
    left_to_right = True

    while queue:
        level_size = len(queue)
        current_level = []

        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        if not left_to_right:
            current_level.reverse()

        result.append(current_level)
        left_to_right = not left_to_right

    return result

return {
    'simple': simple_level_order(),
    'by_levels': level_order_by_levels(),
    'right_to_left': level_order_right_to_left(),
    'zigzag': zigzag_level_order()
}
```

Advanced Level Order Applications

Tree Width and Statistics

```
def tree_statistics_bfs(root):
    """Calculate comprehensive tree statistics using BFS"""
    from collections import deque

    if not root:
        return {
            'width': 0,
            'height': -1,
            'nodes_per_level': [],
            'max_level_width': 0,
            'total_nodes': 0
        }

    queue = deque([root])
    height = -1
    nodes_per_level = []
    max_width = 0
    total_nodes = 0

    while queue:
        level_size = len(queue)
        height += 1
        nodes_per_level.append(level_size)
        max_width = max(max_width, level_size)
        total_nodes += level_size

        for _ in range(level_size):
            node = queue.popleft()

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

    return {
        'width': max_width,
        'height': height,
        'nodes_per_level': nodes_per_level,
        'max_level_width': max_width,
        'total_nodes': total_nodes,
        'average_width': total_nodes / (height + 1) if height &gt;= 0 else 0
    }
```

Chapter 5: Binary Search Trees (Complete Theory & Implementation)

[204]

5.1 BST Mathematical Properties

BST Invariant

For every node v in BST:

- All nodes in left subtree have values < v.val
- All nodes in right subtree have values > v.val
- Both subtrees are also BSTs (recursive property)

Mathematical Analysis

Search Cost Analysis:

- Best case: O(log n) - balanced tree
- Average case: O(log n) - random insertion order
- Worst case: O(n) - skewed tree (sorted input)

Expected Height of Random BST:
E[height] = O(log n)

Proof: The expected height of a randomly built BST is approximately 2.99 log n.

Complete BST Implementation with Theory

```
class BinarySearchTreeComplete:
    """
    Complete BST implementation with all operations and theoretical analysis
    """

    def __init__(self):
        self.root = None
        self.size = 0
        self.modification_count = 0  # For iterator invalidation

    def insert(self, val):
        """
        Insert value maintaining BST property

        Time Complexity:
        - Best/Average: O(log n)
        - Worst: O(n) for skewed tree

        Space Complexity: O(log n) for recursion stack
        """
        self.root = self._insert_recursive(self.root, val)
        self.modification_count += 1
```

```
    def _insert_recursive(self, node, val):
        # Base case: create new node
        if not node:
            self.size += 1
            return TreeNode(val)

        # Recursive case: maintain BST property
        if val &lt; node.val:
            node.left = self._insert_recursive(node.left, val)
        elif val &gt; node.val:
            node.right = self._insert_recursive(node.right, val)
        # Equal values: do nothing (no duplicates)

        return node

    def insert_iterative(self, val):
        """
        Iterative insertion - more space efficient
        Space Complexity: O(1)
        """
        if not self.root:
            self.root = TreeNode(val)
            self.size += 1
            return

        current = self.root

        while True:
            if val &lt; current.val:
                if not current.left:
                    current.left = TreeNode(val)
                    self.size += 1
                    break
                current = current.left
            elif val &gt; current.val:
                if not current.right:
                    current.right = TreeNode(val)
                    self.size += 1
                    break
                current = current.right
            else:
                # Duplicate value
                break

    def search(self, val):
        """
        Search for value in BST

        Returns: TreeNode if found, None otherwise
        Time Complexity: O(log n) average, O(n) worst
        """
        return self._search_recursive(self.root, val)

    def _search_recursive(self, node, val):
        if not node or node.val == val:
```

```
            return node

        if val &lt; node.val:
            return self._search_recursive(node.left, val)
        else:
            return self._search_recursive(node.right, val)

    def search_iterative(self, val):
        """Iterative search - no recursion overhead"""
        current = self.root

        while current:
            if val == current.val:
                return current
            elif val &lt; current.val:
                current = current.left
            else:
                current = current.right

        return None

    def delete(self, val):
        """
        Delete node with given value

        Three cases:
        1. No children: Simply remove
        2. One child: Replace with child
        3. Two children: Replace with inorder successor
        """
        self.root, deleted = self._delete_recursive(self.root, val)
        if deleted:
            self.modification_count += 1
        return deleted

    def _delete_recursive(self, node, val):
        if not node:
            return node, False

        if val &lt; node.val:
            node.left, deleted = self._delete_recursive(node.left, val)
            return node, deleted
        elif val &gt; node.val:
            node.right, deleted = self._delete_recursive(node.right, val)
            return node, deleted
        else:
            # Node to delete found
            self.size -= 1

            # Case 1: No children (leaf)
            if not node.left and not node.right:
                return None, True

            # Case 2: One child
            if not node.left:
                return node.right, True
```

```
            if not node.right:
                return node.left, True

            # Case 3: Two children
            # Find inorder successor (minimum in right subtree)
            successor = self._find_min(node.right)
            node.val = successor.val
            node.right, _ = self._delete_recursive(node.right, successor.val)
            self.size += 1  # Adjust since we decremented above
            return node, True

    def _find_min(self, node):
        """Find node with minimum value in subtree"""
        while node.left:
            node = node.left
        return node

    def _find_max(self, node):
        """Find node with maximum value in subtree"""
        while node.right:
            node = node.right
        return node

    def find_min_value(self):
        """Public interface to find minimum value"""
        if not self.root:
            return None
        return self._find_min(self.root).val

    def find_max_value(self):
        """Public interface to find maximum value"""
        if not self.root:
            return None
        return self._find_max(self.root).val

    def floor(self, val):
        """
        Find largest value &lt;= val (floor)

        Algorithm:
        1. If current node value == val, return it
        2. If current node value &gt; val, go left
        3. If current node value &lt; val, it's potential floor, go right
        """
        return self._floor_recursive(self.root, val)

    def _floor_recursive(self, node, val):
        if not node:
            return None

        if node.val == val:
            return node.val

        if node.val &gt; val:
            return self._floor_recursive(node.left, val)
```

```
        # node.val &lt; val, potential floor
        floor_right = self._floor_recursive(node.right, val)
        return floor_right if floor_right is not None else node.val

    def ceiling(self, val):
        """
        Find smallest value &gt;= val (ceiling)

        Algorithm:
        1. If current node value == val, return it
        2. If current node value &lt; val, go right
        3. If current node value &gt; val, it's potential ceiling, go left
        """
        return self._ceiling_recursive(self.root, val)

    def _ceiling_recursive(self, node, val):
        if not node:
            return None

        if node.val == val:
            return node.val

        if node.val &lt; val:
            return self._ceiling_recursive(node.right, val)

        # node.val &gt; val, potential ceiling
        ceiling_left = self._ceiling_recursive(node.left, val)
        return ceiling_left if ceiling_left is not None else node.val

    def kth_smallest(self, k):
        """
        Find kth smallest element (1-indexed)

        Uses inorder traversal property of BST
        Time: O(k) in best case, O(n) worst case
        """
        def inorder_kth(node):
            nonlocal k, result

            if not node or k &lt;= 0:
                return

            inorder_kth(node.left)

            k -= 1
            if k == 0:
                result = node.val
                return

            inorder_kth(node.right)

        result = None
        inorder_kth(self.root)
        return result

    def kth_largest(self, k):
```

```
        """Find kth largest element using reverse inorder"""
        def reverse_inorder(node):
            nonlocal k, result

            if not node or k &lt;= 0:
                return

            reverse_inorder(node.right)

            k -= 1
            if k == 0:
                result = node.val
                return

            reverse_inorder(node.left)

        result = None
        reverse_inorder(self.root)
        return result

    def range_query(self, low, high):
        """
        Find all values in range [low, high]

        Optimized: only visit nodes that could contain values in range
        """
        result = []

        def range_search(node):
            if not node:
                return

            # If current value is in range, add it
            if low &lt;= node.val &lt;= high:
                result.append(node.val)

            # Recursively search left if there could be values in range
            if node.val &gt; low:
                range_search(node.left)

            # Recursively search right if there could be values in range
            if node.val &lt; high:
                range_search(node.right)

        range_search(self.root)
        return sorted(result)

    def is_valid_bst(self, min_val=float('-inf'), max_val=float('inf')):
        """
        Validate BST property

        Each node must satisfy: min_val &lt; node.val &lt; max_val
        """
        def validate(node, min_val, max_val):
            if not node:
                return True
```

```
            if node.val &lt;= min_val or node.val &gt;= max_val:
                return False

            return (validate(node.left, min_val, node.val) and
                    validate(node.right, node.val, max_val))

        return validate(self.root, min_val, max_val)

    def inorder_traversal(self):
        """Get sorted sequence (inorder traversal of BST)"""
        result = []

        def inorder(node):
            if node:
                inorder(node.left)
                result.append(node.val)
                inorder(node.right)

        inorder(self.root)
        return result

    def get_height(self):
        """Calculate height of BST"""
        def height(node):
            if not node:
                return -1
            return 1 + max(height(node.left), height(node.right))

        return height(self.root)

    def get_statistics(self):
        """Get comprehensive BST statistics"""
        if not self.root:
            return {
                'size': 0,
                'height': -1,
                'min': None,
                'max': None,
                'is_balanced': True,
                'is_valid': True
            }

        return {
            'size': self.size,
            'height': self.get_height(),
            'min': self.find_min_value(),
            'max': self.find_max_value(),
            'is_balanced': self._is_balanced(),
            'is_valid': self.is_valid_bst(),
            'inorder': self.inorder_traversal()
        }

    def _is_balanced(self):
        """Check if BST is height-balanced"""
        def check_balance(node):
```

```
            if not node:
                return True, -1

            left_balanced, left_height = check_balance(node.left)
            if not left_balanced:
                return False, 0

            right_balanced, right_height = check_balance(node.right)
            if not right_balanced:
                return False, 0

            height = 1 + max(left_height, right_height)
            balanced = abs(left_height - right_height) &lt;= 1

            return balanced, height

        is_balanced, _ = check_balance(self.root)
        return is_balanced
```

BST Construction from Traversals

[208]

Build BST from Preorder

```
def build_bst_from_preorder(preorder):
    """
    Build BST from preorder traversal

    Key insight: Use min/max bounds to determine valid positions
    Time: O(n), Space: O(n)
    """
    if not preorder:
        return None

    def build(min_val, max_val):
        nonlocal idx

        if idx &gt;= len(preorder):
            return None

        val = preorder[idx]
        if val &lt; min_val or val &gt; max_val:
            return None

        idx += 1
        root = TreeNode(val)
        root.left = build(min_val, val)
        root.right = build(val, max_val)
        return root

    idx = 0
    return build(float('-inf'), float('inf'))
```

```python
def build_tree_from_preorder_inorder(preorder, inorder):
    """
    Build tree from preorder and inorder traversals

    Algorithm:
    1. First element of preorder is root
    2. Find root position in inorder
    3. Left part of inorder = left subtree
    4. Right part of inorder = right subtree
    5. Recursively build subtrees
    """
    if not preorder or not inorder:
        return None

    # Create root from first preorder element
    root_val = preorder[0]
    root = TreeNode(root_val)

    # Find root position in inorder
    root_idx = inorder.index(root_val)

    # Build left subtree
    left_inorder = inorder[:root_idx]
    left_preorder = preorder[1:1 + len(left_inorder)]
    root.left = build_tree_from_preorder_inorder(left_preorder, left_inorder)

    # Build right subtree
    right_inorder = inorder[root_idx + 1:]
    right_preorder = preorder[1 + len(left_inorder):]
    root.right = build_tree_from_preorder_inorder(right_preorder, right_inorder)

    return root
```

Chapter 6: Balanced Trees (AVL & Red-Black Trees)

[203] [207] [209]

6.1 AVL Trees (Complete Implementation)

AVL Tree Theory

Definition: A self-balancing BST where heights of left and right subtrees differ by at most 1.

Balance Factor: BF(node) = height(left) - height(right)

- BF ∈ {-1, 0, 1} for all nodes
- If |BF| > 1, tree is unbalanced and needs rotation

Height Guarantee: For AVL tree with n nodes:

- Height ≤ 1.44 $\log_2(n + 2)$ - 0.328
- This guarantees O(log n) operations

Rotation Theory and Implementation

Single Rotations

Left Rotation (RR Case):

```
Before:     After:
   A          B
    \        / \
     B      A   C
      \
       C
```

Right Rotation (LL Case):

```
Before:     After:
     A        B
    /        / \
   B        C   A
  /
 C
```

Double Rotations

Left-Right Rotation (LR Case):

```
Before:    Intermediate:    After:
   A           A              C
  /           /              / \
 B    --&gt;    C      --&gt;     B   A
  \         /
   C       B
```

Right-Left Rotation (RL Case):

```
Before:    Intermediate:    After:
 A             A              C
  \             \            / \
   B     --&gt;     C    --&gt;   A   B
  /               \
 C                 B
```

Complete AVL Tree Implementation

```
class AVLNode:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1  # Height of subtree rooted at this node
```

```
class AVLTree:
    """
    Complete AVL Tree implementation with detailed explanations
    """

    def __init__(self):
        self.root = None
        self.size = 0

    def get_height(self, node):
        """Get height of node (0 for None)"""
        return node.height if node else 0

    def get_balance_factor(self, node):
        """Calculate balance factor: height(left) - height(right)"""
        if not node:
            return 0
        return self.get_height(node.left) - self.get_height(node.right)

    def update_height(self, node):
        """Update height based on children heights"""
        if node:
            node.height = 1 + max(self.get_height(node.left),
                                  self.get_height(node.right))

    def right_rotate(self, y):
        """
        Right rotation (LL case)

        Before:      After:
            y           x
           / \         / \
          x   T3      T1  y
         / \             / \
        T1  T2          T2  T3

        Time: O(1), maintains BST property
        """
        x = y.left
        T2 = x.right

        # Perform rotation
        x.right = y
        y.left = T2

        # Update heights (order matters!)
        self.update_height(y)
        self.update_height(x)

        return x  # New root of subtree

    def left_rotate(self, x):
        """
        Left rotation (RR case)
```

```
        Before:      After:
          x             y
         / \           / \
        T1  y         x   T3
           / \       / \
          T2  T3    T1  T2
        """
        y = x.right
        T2 = y.left

        # Perform rotation
        y.left = x
        x.right = T2

        # Update heights
        self.update_height(x)
        self.update_height(y)

        return y  # New root of subtree

    def insert(self, val):
        """Insert value maintaining AVL property"""
        self.root = self._insert_recursive(self.root, val)

    def _insert_recursive(self, node, val):
        """
        Recursive insertion with rebalancing

        Algorithm:
        1. Perform normal BST insertion
        2. Update height
        3. Get balance factor
        4. Perform rotations if needed
        """

        # Step 1: Perform normal BST insertion
        if not node:
            self.size += 1
            return AVLNode(val)

        if val &lt; node.val:
            node.left = self._insert_recursive(node.left, val)
        elif val &gt; node.val:
            node.right = self._insert_recursive(node.right, val)
        else:
            return node  # Duplicate values not allowed

        # Step 2: Update height of current node
        self.update_height(node)

        # Step 3: Get balance factor
        balance = self.get_balance_factor(node)

        # Step 4: If unbalanced, there are 4 cases

        # Left Left Case
```

```
        if balance &gt; 1 and val &lt; node.left.val:
            return self.right_rotate(node)

        # Right Right Case
        if balance &lt; -1 and val &gt; node.right.val:
            return self.left_rotate(node)

        # Left Right Case
        if balance &gt; 1 and val &gt; node.left.val:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)

        # Right Left Case
        if balance &lt; -1 and val &lt; node.right.val:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        # Return unchanged node if balanced
        return node

    def delete(self, val):
        """Delete value maintaining AVL property"""
        self.root = self._delete_recursive(self.root, val)

    def _delete_recursive(self, node, val):
        """Recursive deletion with rebalancing"""

        # Step 1: Perform normal BST deletion
        if not node:
            return node

        if val &lt; node.val:
            node.left = self._delete_recursive(node.left, val)
        elif val &gt; node.val:
            node.right = self._delete_recursive(node.right, val)
        else:
            # Node to be deleted found
            self.size -= 1

            if not node.left or not node.right:
                temp = node.left if node.left else node.right

                if not temp:  # No child case
                    temp = node
                    node = None
                else:  # One child case
                    node = temp
            else:
                # Two children case
                temp = self._find_min_node(node.right)
                node.val = temp.val
                node.right = self._delete_recursive(node.right, temp.val)
                self.size += 1  # Adjust count

        if not node:
            return node
```

```
        # Step 2: Update height
        self.update_height(node)

        # Step 3: Get balance factor
        balance = self.get_balance_factor(node)

        # Step 4: Rebalance if needed

        # Left Left Case
        if balance &gt; 1 and self.get_balance_factor(node.left) &gt;= 0:
            return self.right_rotate(node)

        # Left Right Case
        if balance &gt; 1 and self.get_balance_factor(node.left) &lt; 0:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)

        # Right Right Case
        if balance &lt; -1 and self.get_balance_factor(node.right) &lt;= 0:
            return self.left_rotate(node)

        # Right Left Case
        if balance &lt; -1 and self.get_balance_factor(node.right) &gt; 0:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def _find_min_node(self, node):
        """Find minimum node in subtree"""
        while node.left:
            node = node.left
        return node

    def search(self, val):
        """Search for value (same as BST)"""
        current = self.root

        while current:
            if val == current.val:
                return current
            elif val &lt; current.val:
                current = current.left
            else:
                current = current.right

        return None

    def is_balanced(self, node=None):
        """Check if tree is balanced (for verification)"""
        if node is None:
            node = self.root

        if not node:
            return True
```

```
        balance = self.get_balance_factor(node)

        return (abs(balance) &lt;= 1 and
                self.is_balanced(node.left) and
                self.is_balanced(node.right))

    def get_tree_info(self):
        """Get comprehensive tree information"""
        def get_node_info(node, level=0):
            if not node:
                return []

            info = [{
                'val': node.val,
                'level': level,
                'height': node.height,
                'balance_factor': self.get_balance_factor(node),
                'left_child': node.left.val if node.left else None,
                'right_child': node.right.val if node.right else None
            }]

            info.extend(get_node_info(node.left, level + 1))
            info.extend(get_node_info(node.right, level + 1))

            return info

        return {
            'size': self.size,
            'height': self.get_height(self.root),
            'is_balanced': self.is_balanced(),
            'nodes': get_node_info(self.root)
        }
```

AVL Tree Complexity Analysis

Operation	Time Complexity	Space Complexity	Notes
Search	O(log n)	O(log n)	Guaranteed balanced
Insert	O(log n)	O(log n)	At most 2 rotations
Delete	O(log n)	O(log n)	At most O(log n) rotations
Height	O(1)	O(1)	Stored in each node

Rotation Analysis:

- Single rotation: O(1) time, changes 2 nodes
- Double rotation: O(1) time, changes 3 nodes
- Maximum rotations per insertion: 2
- Maximum rotations per deletion: O(log n)

Chapter 7: Heap Data Structures (Complete Theory & Applications)

[201] [202]

7.1 Heap Theory and Mathematical Properties

Heap Definition

A **heap** is a complete binary tree that satisfies the heap property:

- **Min Heap**: For every node, parent ≤ children
- **Max Heap**: For every node, parent ≥ children

Mathematical Properties

Shape Property (Complete Binary Tree):

- All levels filled except possibly last
- Last level filled left to right
- Height = $\lfloor \log_2(n) \rfloor$

Array Representation Formulas:

```
For node at index i (0-based):
- Parent: (i-1)/2
- Left child: 2i+1
- Right child: 2i+2

For node at index i (1-based):
- Parent: i/2
- Left child: 2i
- Right child: 2i+1
```

Heap Height Analysis:

- Minimum height: $\lfloor \log_2(n) \rfloor$ (complete tree)
- Maximum nodes at height h: $\lceil n/2^{(h+1)} \rceil$

Complete Heap Implementation

```
class MinHeapComplete:
    """
    Complete Min Heap implementation with detailed analysis
    """

    def __init__(self, capacity=None):
        self.heap = []
        self.size = 0
        self.capacity = capacity
        self.comparison_count = 0  # For analysis
```

```
    def parent_index(self, i):
        """Get parent index. Math: (i-1)//2"""
        return (i - 1) // 2 if i &gt; 0 else None

    def left_child_index(self, i):
        """Get left child index. Math: 2*i + 1"""
        left = 2 * i + 1
        return left if left &lt; self.size else None

    def right_child_index(self, i):
        """Get right child index. Math: 2*i + 2"""
        right = 2 * i + 2
        return right if right &lt; self.size else None

    def has_left_child(self, i):
        return self.left_child_index(i) is not None

    def has_right_child(self, i):
        return self.right_child_index(i) is not None

    def has_parent(self, i):
        return self.parent_index(i) is not None

    def left_child_value(self, i):
        return self.heap[self.left_child_index(i)]

    def right_child_value(self, i):
        return self.heap[self.right_child_index(i)]

    def parent_value(self, i):
        return self.heap[self.parent_index(i)]

    def insert(self, val):
        """
        Insert value maintaining heap property

        Algorithm:
        1. Add element at end (maintain complete tree)
        2. Bubble up to restore heap property

        Time: O(log n) - height of tree
        Space: O(1) - no additional space
        """
        if self.capacity and self.size &gt;= self.capacity:
            raise OverflowError("Heap is full")

        # Step 1: Add at end
        self.heap.append(val)
        self.size += 1

        # Step 2: Bubble up
        self._heapify_up(self.size - 1)

    def _heapify_up(self, index):
        """
```

```
        Restore heap property by moving element up

        Invariant: heap property satisfied except possibly at index
        """
        steps = []  # For debugging/visualization

        while self.has_parent(index):
            parent_idx = self.parent_index(index)
            self.comparison_count += 1

            if self.heap[index] &gt;= self.heap[parent_idx]:
                break  # Heap property satisfied

            # Swap with parent
            steps.append(f"Swapping {self.heap[index]} with parent {self.heap[parent_idx]}"
            self.heap[index], self.heap[parent_idx] = self.heap[parent_idx], self.heap[inde
            index = parent_idx

        return steps

    def extract_min(self):
        """
        Remove and return minimum element (root)

        Algorithm:
        1. Save root value
        2. Move last element to root
        3. Remove last element
        4. Bubble down from root

        Time: O(log n)
        Space: O(1)
        """
        if self.size == 0:
            raise IndexError("Heap is empty")

        # Step 1: Save min value
        min_val = self.heap[0]

        # Step 2: Move last to root
        self.heap[0] = self.heap[self.size - 1]

        # Step 3: Remove last
        self.size -= 1
        self.heap.pop()

        # Step 4: Bubble down if heap not empty
        if self.size &gt; 0:
            self._heapify_down(0)

        return min_val

    def _heapify_down(self, index):
        """
        Restore heap property by moving element down
```

```
        Algorithm:
        1. Compare with children
        2. Swap with smaller child if necessary
        3. Continue until heap property restored
        """
        steps = []

        while self.has_left_child(index):
            # Find smaller child
            smaller_child_idx = self.left_child_index(index)

            if (self.has_right_child(index) and
                self.right_child_value(index) &lt; self.left_child_value(index)):
                smaller_child_idx = self.right_child_index(index)

            self.comparison_count += 1

            # If heap property satisfied, stop
            if self.heap[index] &lt;= self.heap[smaller_child_idx]:
                break

            # Swap with smaller child
            steps.append(f"Swapping {self.heap[index]} with child {self.heap[smaller_child_
            self.heap[index], self.heap[smaller_child_idx] = \
                self.heap[smaller_child_idx], self.heap[index]
            index = smaller_child_idx

        return steps

    def peek(self):
        """Get minimum without removing. Time: O(1)"""
        if self.size == 0:
            raise IndexError("Heap is empty")
        return self.heap[0]

    def build_heap(self, arr):
        """
        Build heap from array using Floyd's algorithm

        Time: O(n) - better than n insertions O(n log n)

        Algorithm:
        1. Copy array
        2. Start from last non-leaf node
        3. Heapify down for each node
        """
        self.heap = arr[:]
        self.size = len(arr)
        self.comparison_count = 0

        # Start from last non-leaf: (n-2)//2 down to 0
        for i in range((self.size - 2) // 2, -1, -1):
            self._heapify_down(i)

    def increase_key(self, index, new_val):
        """
```

```
        Increase value at index (for min heap, may need to bubble down)
        """
        if index &gt;= self.size:
            raise IndexError("Index out of range")

        old_val = self.heap[index]
        self.heap[index] = new_val

        if new_val &lt; old_val:
            self._heapify_up(index)
        elif new_val &gt; old_val:
            self._heapify_down(index)

    def delete(self, index):
        """Delete element at specific index"""
        if index &gt;= self.size:
            raise IndexError("Index out of range")

        # Move last element to deleted position
        self.heap[index] = self.heap[self.size - 1]
        self.size -= 1
        self.heap.pop()

        if index &lt; self.size:
            parent_idx = self.parent_index(index)

            # Decide whether to bubble up or down
            if (parent_idx is not None and
                self.heap[index] &lt; self.heap[parent_idx]):
                self._heapify_up(index)
            else:
                self._heapify_down(index)

    def is_valid_heap(self):
        """Verify heap property"""
        for i in range(self.size):
            left_idx = self.left_child_index(i)
            right_idx = self.right_child_index(i)

            if left_idx and self.heap[i] &gt; self.heap[left_idx]:
                return False
            if right_idx and self.heap[i] &gt; self.heap[right_idx]:
                return False

        return True

    def get_statistics(self):
        """Get heap statistics"""
        import math

        return {
            'size': self.size,
            'height': int(math.log2(self.size)) if self.size &gt; 0 else -1,
            'min_value': self.peek() if self.size &gt; 0 else None,
            'is_valid': self.is_valid_heap(),
            'comparisons_made': self.comparison_count,
```

```
            'array_representation': self.heap[:]
        }

class MaxHeap(MinHeapComplete):
    """Max Heap implementation - inherits from MinHeap but reverses comparisons"""

    def _heapify_up(self, index):
        steps = []

        while self.has_parent(index):
            parent_idx = self.parent_index(index)
            self.comparison_count += 1

            # Reversed comparison for max heap
            if self.heap[index] &lt;= self.heap[parent_idx]:
                break

            steps.append(f"Swapping {self.heap[index]} with parent {self.heap[parent_idx]}"
            self.heap[index], self.heap[parent_idx] = self.heap[parent_idx], self.heap[inde
            index = parent_idx

        return steps

    def _heapify_down(self, index):
        steps = []

        while self.has_left_child(index):
            # Find larger child for max heap
            larger_child_idx = self.left_child_index(index)

            if (self.has_right_child(index) and
                self.right_child_value(index) &gt; self.left_child_value(index)):
                larger_child_idx = self.right_child_index(index)

            self.comparison_count += 1

            # Reversed comparison for max heap
            if self.heap[index] &gt;= self.heap[larger_child_idx]:
                break

            steps.append(f"Swapping {self.heap[index]} with child {self.heap[larger_child_i
            self.heap[index], self.heap[larger_child_idx] = \
                self.heap[larger_child_idx], self.heap[index]
            index = larger_child_idx

        return steps

    def extract_max(self):
        """Extract maximum element"""
        return self.extract_min()  # Same algorithm, different comparisons

    def peek_max(self):
        """Get maximum without removing"""
        return self.peek()
```

Heap Applications

Priority Queue Implementation

```
class PriorityQueue:
    """Priority Queue using heap"""

    def __init__(self, min_heap=True):
        self.heap = MinHeapComplete() if min_heap else MaxHeap()
        self.min_heap = min_heap

    def enqueue(self, item, priority):
        """Add item with priority"""
        # For min heap: lower number = higher priority
        # For max heap: higher number = higher priority
        if self.min_heap:
            self.heap.insert((priority, item))
        else:
            self.heap.insert((-priority, item))  # Negate for max heap

    def dequeue(self):
        """Remove highest priority item"""
        if self.heap.size == 0:
            raise IndexError("Priority queue is empty")

        priority, item = self.heap.extract_min()
        if not self.min_heap:
            priority = -priority

        return item, priority

    def peek(self):
        """Get highest priority item without removing"""
        if self.heap.size == 0:
            raise IndexError("Priority queue is empty")

        priority, item = self.heap.peek()
        if not self.min_heap:
            priority = -priority

        return item, priority

# Usage examples
def heap_sort(arr):
    """Sort array using heap sort algorithm"""
    # Build max heap
    heap = MaxHeap()
    heap.build_heap(arr[:])

    result = []
    while heap.size &gt; 0:
        result.append(heap.extract_max())

    return result
```

```
def find_k_largest(arr, k):
    """Find k largest elements using min heap"""
    if k &gt; len(arr):
        return arr

    # Use min heap of size k
    heap = MinHeapComplete()

    for num in arr:
        if heap.size &lt; k:
            heap.insert(num)
        elif num &gt; heap.peek():
            heap.extract_min()
            heap.insert(num)

    return [heap.extract_min() for _ in range(heap.size)][::-1]

def merge_k_sorted_arrays(arrays):
    """Merge k sorted arrays using min heap"""
    heap = MinHeapComplete()
    result = []

    # Initialize heap with first element from each array
    for i, arr in enumerate(arrays):
        if arr:
            heap.insert((arr[0], i, 0))  # (value, array_index, element_index)

    while heap.size &gt; 0:
        val, arr_idx, elem_idx = heap.extract_min()
        result.append(val)

        # Add next element from same array if exists
        if elem_idx + 1 &lt; len(arrays[arr_idx]):
            next_val = arrays[arr_idx][elem_idx + 1]
            heap.insert((next_val, arr_idx, elem_idx + 1))

    return result
```

Chapter 8: Trie & Prefix Trees (Complete Implementation)

[212] [213]

8.1 Trie Theory and Applications

Trie Definition

A **Trie** (prefix tree) is a tree-like data structure for storing strings where:

- Each node represents a character
- Each path from root to node represents a prefix
- Complete words are marked with special flag

Mathematical Properties

Space Complexity: O(ALPHABET_SIZE × N × M)

- N = number of strings
- M = average length of strings
- ALPHABET_SIZE = size of character set

Time Complexities:

- Insert: O(M) where M = string length
- Search: O(M)
- Delete: O(M)
- Prefix search: O(P) where P = prefix length

Complete Trie Implementation

```
class TrieNodeComplete:
    """
    Trie node with comprehensive features
    """
    def __init__(self):
        self.children = {}  # Character -&gt; TrieNode mapping
        self.is_end_of_word = False
        self.word_count = 0  # How many times this word appears
        self.prefix_count = 0  # How many words pass through this node
        self.word = None  # Store complete word for easy retrieval

    def __repr__(self):
        return f"TrieNode(end={self.is_end_of_word}, children={len(self.children)})"

class TrieComplete:
    """
    Complete Trie implementation with all operations and optimizations
    """

    def __init__(self):
        self.root = TrieNodeComplete()
        self.total_words = 0
        self.total_unique_words = 0

    def insert(self, word):
        """
        Insert word into trie

        Algorithm:
        1. Start from root
        2. For each character, move to child (create if needed)
        3. Mark end of word
        4. Update counters

        Time: O(M) where M = length of word
        Space: O(M) in worst case (all new characters)
```

```python
        """
        if not word:
            return

        node = self.root

        # Traverse/create path for each character
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNodeComplete()

            node = node.children[char]
            node.prefix_count += 1  # Increment prefix counter

        # Mark end of word
        if not node.is_end_of_word:
            self.total_unique_words += 1

        node.is_end_of_word = True
        node.word_count += 1
        node.word = word  # Store complete word
        self.total_words += 1

    def search(self, word):
        """
        Search for exact word in trie

        Returns: True if word exists, False otherwise
        Time: O(M)
        """
        node = self._find_node(word)
        return node is not None and node.is_end_of_word

    def starts_with(self, prefix):
        """
        Check if any word starts with given prefix

        Returns: True if prefix exists, False otherwise
        Time: O(P) where P = length of prefix
        """
        return self._find_node(prefix) is not None

    def _find_node(self, word):
        """
        Helper method to find node corresponding to word/prefix

        Returns: TrieNode if path exists, None otherwise
        """
        node = self.root

        for char in word:
            if char not in node.children:
                return None
            node = node.children[char]

        return node
```

```
def delete(self, word):
    """
    Delete word from trie

    Algorithm:
    1. Find the word
    2. Decrement counters
    3. Remove nodes if they're no longer needed

    Time: O(M)
    Returns: True if word was deleted, False if not found
    """
    def _delete_recursive(node, word, index):
        if index == len(word):
            # End of word reached
            if not node.is_end_of_word:
                return False  # Word doesn't exist

            # Mark as not end of word
            node.is_end_of_word = False
            node.word_count -= 1

            # Return True if this node can be deleted
            # (no other words pass through it)
            return len(node.children) == 0 and node.word_count == 0

        char = word[index]
        child_node = node.children.get(char)

        if not child_node:
            return False  # Word doesn't exist

        # Recursively delete
        should_delete_child = _delete_recursive(child_node, word, index + 1)

        if should_delete_child:
            del node.children[char]
            # Return True if current node can also be deleted
            return (not node.is_end_of_word and
                    len(node.children) == 0 and
                    node.word_count == 0)

        return False

    if self.search(word):
        _delete_recursive(self.root, word, 0)
        self.total_words -= 1
        self.total_unique_words -= 1
        return True

    return False

def get_all_words_with_prefix(self, prefix):
    """
    Get all words that start with given prefix
```

```
        Algorithm:
        1. Find node corresponding to prefix
        2. DFS from that node to collect all words

        Time: O(P + N) where P = prefix length, N = number of results
        """
        prefix_node = self._find_node(prefix)
        if not prefix_node:
            return []

        words = []
        self._collect_all_words(prefix_node, prefix, words)
        return words

    def _collect_all_words(self, node, current_word, words):
        """
        DFS helper to collect all words from a subtree
        """
        if node.is_end_of_word:
            words.append(current_word)

        for char, child_node in sorted(node.children.items()):
            self._collect_all_words(child_node, current_word + char, words)

    def auto_complete(self, prefix, max_suggestions=10):
        """
        Get autocomplete suggestions for given prefix

        Returns: List of suggested words (limited by max_suggestions)
        """
        suggestions = self.get_all_words_with_prefix(prefix)

        # Sort by frequency (word_count) if available
        suggestions.sort(key=lambda word: self._find_node(word).word_count, reverse=True)

        return suggestions[:max_suggestions]

    def count_words_with_prefix(self, prefix):
        """
        Count how many words start with given prefix

        Uses prefix_count for O(P) time complexity
        """
        node = self._find_node(prefix)
        return node.prefix_count if node else 0

    def longest_common_prefix(self):
        """
        Find longest common prefix of all words in trie

        Algorithm:
        1. Start from root
        2. While there's only one child and it's not end of word
        3. Continue building prefix
        """
```

```
        if self.total_unique_words == 0:
            return ""

        prefix = ""
        node = self.root

        while (len(node.children) == 1 and
               not node.is_end_of_word):
            char = next(iter(node.children))
            prefix += char
            node = node.children[char]

        return prefix

    def find_shortest_unique_prefix(self, word):
        """
        Find shortest unique prefix for given word

        Returns: Shortest prefix that uniquely identifies the word
        """
        if not self.search(word):
            return None

        node = self.root
        prefix = ""

        for char in word:
            prefix += char
            node = node.children[char]

            # If this node has only one word passing through it,
            # then current prefix is unique
            if node.prefix_count == 1:
                return prefix

        return word  # Entire word is needed

    def word_break(self, s):
        """
        Check if string s can be segmented using words in trie

        Dynamic Programming approach with trie optimization
        Time: O(N²) where N = length of string
        """
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True  # Empty string can always be segmented

        for i in range(1, n + 1):
            for j in range(i):
                if dp[j] and self.search(s[j:i]):
                    dp[i] = True
                    break

        return dp[n]
```

```python
    def get_statistics(self):
        """Get comprehensive trie statistics"""
        def calculate_stats(node, depth=0):
            stats = {
                'nodes': 1,
                'max_depth': depth,
                'total_depth': depth,
                'leaf_nodes': 1 if len(node.children) == 0 else 0,
                'end_word_nodes': 1 if node.is_end_of_word else 0
            }

            for child in node.children.values():
                child_stats = calculate_stats(child, depth + 1)
                stats['nodes'] += child_stats['nodes']
                stats['max_depth'] = max(stats['max_depth'], child_stats['max_depth'])
                stats['total_depth'] += child_stats['total_depth']
                stats['leaf_nodes'] += child_stats['leaf_nodes']
                stats['end_word_nodes'] += child_stats['end_word_nodes']

            return stats

        stats = calculate_stats(self.root)

        return {
            'total_words': self.total_words,
            'unique_words': self.total_unique_words,
            'total_nodes': stats['nodes'],
            'max_depth': stats['max_depth'],
            'average_depth': stats['total_depth'] / stats['nodes'] if stats['nodes'] &gt; 0
            'leaf_nodes': stats['leaf_nodes'],
            'end_word_nodes': stats['end_word_nodes'],
            'longest_common_prefix': self.longest_common_prefix()
        }

    def visualize(self, max_depth=3):
        """
        Create a visual representation of trie structure
        """
        def _visualize_recursive(node, prefix="", depth=0, is_last=True):
            if depth &gt; max_depth:
                return ["... (truncated)"]

            lines = []

            # Current node representation
            connector = "└── " if is_last else "├── "
            if depth == 0:
                lines.append("ROOT")
            else:
                char = prefix[-1] if prefix else ""
                end_marker = " (END)" if node.is_end_of_word else ""
                count_info = f" [{node.prefix_count}]" if node.prefix_count &gt; 0 else ""
                lines.append(connector + char + end_marker + count_info)

            # Children
            children = list(node.children.items())
```

```
            for i, (char, child) in enumerate(children):
                is_last_child = (i == len(children) - 1)
                extension = "    " if is_last else "│   "
                child_lines = _visualize_recursive(child, prefix + char, depth + 1, is_last

                for j, line in enumerate(child_lines):
                    if j == 0:
                        lines.append(extension + line)
                    else:
                        lines.append(extension + line)

            return lines

        return "\n".join(_visualize_recursive(self.root))
```

Trie Applications and Advanced Algorithms

Word Games and Dictionary Operations

```
class TrieWordGame(TrieComplete):
    """Trie optimized for word games like Scrabble, Boggle"""

    def find_anagrams(self, word):
        """Find all anagrams of given word in trie"""
        from collections import Counter

        target_count = Counter(word)
        anagrams = []

        def dfs(node, path, remaining_count):
            if node.is_end_of_word and not any(remaining_count.values()):
                anagrams.append(path)

            for char, child in node.children.items():
                if remaining_count.get(char, 0) &gt; 0:
                    remaining_count[char] -= 1
                    dfs(child, path + char, remaining_count)
                    remaining_count[char] += 1

        dfs(self.root, "", target_count.copy())
        return anagrams

    def boggle_solver(self, board, min_word_length=3):
        """Solve Boggle game using trie"""
        if not board or not board[0]:
            return []

        rows, cols = len(board), len(board[0])
        found_words = set()

        def dfs(row, col, node, path, visited):
            if (row &lt; 0 or row &gt;= rows or col &lt; 0 or col &gt;= cols or
                (row, col) in visited):
                return
```

```
            char = board[row][col].lower()
            if char not in node.children:
                return

            next_node = node.children[char]
            new_path = path + char
            visited.add((row, col))

            # Check if we found a word
            if (next_node.is_end_of_word and
                len(new_path) &gt;= min_word_length):
                found_words.add(new_path)

            # Explore all 8 directions
            for dr, dc in [(-1,-1), (-1,0), (-1,1), (0,-1), (0,1), (1,-1), (1,0), (1,1)]:
                dfs(row + dr, col + dc, next_node, new_path, visited)

            visited.remove((row, col))

        # Try starting from each cell
        for i in range(rows):
            for j in range(cols):
                dfs(i, j, self.root, "", set())

        return sorted(list(found_words))

    def spell_checker(self, word, max_distance=2):
        """Find words within edit distance using trie"""
        suggestions = []

        def dfs(node, target, current_word, current_row):
            if len(current_row) &gt; max_distance + 1:
                return

            if node.is_end_of_word and current_row[-1] &lt;= max_distance:
                suggestions.append((current_word, current_row[-1]))

            for char, child_node in node.children.items():
                new_word = current_word + char
                new_row = [current_row[0] + 1]

                for i in range(1, len(target) + 1):
                    if char == target[i-1]:
                        new_row.append(current_row[i-1])
                    else:
                        new_row.append(min(
                            current_row[i] + 1,      # deletion
                            new_row[i-1] + 1,        # insertion
                            current_row[i-1] + 1     # substitution
                        ))

                if min(new_row) &lt;= max_distance:
                    dfs(child_node, target, new_word, new_row)

        first_row = list(range(len(word) + 1))
```

```
        dfs(self.root, word, "", first_row)

        # Sort by edit distance, then alphabetically
        suggestions.sort(key=lambda x: (x[1], x[0]))
        return [word for word, _ in suggestions]
```

Chapter 9: Advanced Trees (Segment & Fenwick Trees)

[215] [216]

9.1 Segment Tree (Complete Implementation)

Segment Tree Theory

Purpose: Efficiently answer range queries and perform range updates on arrays.

Structure:

- Complete binary tree built on top of array
- Each node stores information about a segment [l, r]
- Leaves represent single elements
- Internal nodes represent union of children segments

Key Properties:

- Height: O(log n)
- Total nodes: ≤ 4n (practical bound)
- Space complexity: O(4n)

[216]

Complete Segment Tree Implementation

```
class SegmentTreeComplete:
    """
    Complete Segment Tree implementation supporting multiple operations
    """

    def __init__(self, arr, operation='sum'):
        """
        Initialize segment tree with array and operation type

        Args:
            arr: Input array
            operation: 'sum', 'min', 'max', 'gcd', 'xor'
        """
        self.n = len(arr)
        self.arr = arr[:]
        self.tree = [0] * (4 * self.n)  # 4n space is sufficient
        self.lazy = [0] * (4 * self.n)  # For lazy propagation
```

```python
        self.operation = operation

        # Define operation functions
        self.ops = {
            'sum': (lambda x, y: x + y, 0),
            'min': (lambda x, y: min(x, y), float('inf')),
            'max': (lambda x, y: max(x, y), float('-inf')),
            'gcd': (self._gcd, 0),
            'xor': (lambda x, y: x ^ y, 0)
        }

        self.combine, self.identity = self.ops[operation]

        # Build the tree
        self.build(1, 0, self.n - 1)

    def _gcd(self, a, b):
        """Helper function for GCD operation"""
        while b:
            a, b = b, a % b
        return a

    def build(self, node, start, end):
        """
        Build segment tree recursively

        Args:
            node: Current node index in tree
            start, end: Range [start, end] this node represents

        Time: O(n) - visit each array element exactly once
        """
        if start == end:
            # Leaf node
            self.tree[node] = self.arr[start]
        else:
            # Internal node
            mid = (start + end) // 2

            # Build left and right subtrees
            self.build(2 * node, start, mid)
            self.build(2 * node + 1, mid + 1, end)

            # Combine results from children
            self.tree[node] = self.combine(
                self.tree[2 * node],
                self.tree[2 * node + 1]
            )

    def update_point(self, node, start, end, idx, val):
        """
        Update single point in array

        Time: O(log n) - traverse path from root to leaf
        """
        if start == end:
```

```
            # Leaf node - update value
            self.arr[idx] = val
            self.tree[node] = val
        else:
            mid = (start + end) // 2

            if idx &lt;= mid:
                # Update in left subtree
                self.update_point(2 * node, start, mid, idx, val)
            else:
                # Update in right subtree
                self.update_point(2 * node + 1, mid + 1, end, idx, val)

            # Recompute current node value
            self.tree[node] = self.combine(
                self.tree[2 * node],
                self.tree[2 * node + 1]
            )

    def query_range(self, node, start, end, l, r):
        """
        Query range [l, r]

        Returns: Result of operation on range [l, r]
        Time: O(log n) - at most 4 * log n nodes visited
        """
        if r &lt; start or end &lt; l:
            # No overlap
            return self.identity

        if l &lt;= start and end &lt;= r:
            # Complete overlap
            return self.tree[node]

        # Partial overlap - check both children
        mid = (start + end) // 2
        left_result = self.query_range(2 * node, start, mid, l, r)
        right_result = self.query_range(2 * node + 1, mid + 1, end, l, r)

        return self.combine(left_result, right_result)

    def update_range_lazy(self, node, start, end, l, r, val):
        """
        Range update with lazy propagation

        Updates range [l, r] by adding val to each element
        Time: O(log n) amortized
        """
        # Apply pending lazy update if exists
        if self.lazy[node] != 0:
            if self.operation == 'sum':
                self.tree[node] += (end - start + 1) * self.lazy[node]
            else:
                # For other operations, lazy propagation needs modification
                self.tree[node] += self.lazy[node]
```

```
            # Propagate to children if not leaf
            if start != end:
                self.lazy[2 * node] += self.lazy[node]
                self.lazy[2 * node + 1] += self.lazy[node]

            self.lazy[node] = 0

        # No overlap
        if r &lt; start or end &lt; l:
            return

        # Complete overlap
        if l &lt;= start and end &lt;= r:
            if self.operation == 'sum':
                self.tree[node] += (end - start + 1) * val
            else:
                self.tree[node] += val

            # Mark children as lazy if not leaf
            if start != end:
                self.lazy[2 * node] += val
                self.lazy[2 * node + 1] += val

            return

        # Partial overlap
        mid = (start + end) // 2
        self.update_range_lazy(2 * node, start, mid, l, r, val)
        self.update_range_lazy(2 * node + 1, mid + 1, end, l, r, val)

        # Recompute current node (after handling lazy updates on children)
        self._push_down(2 * node, start, mid)
        self._push_down(2 * node + 1, mid + 1, end)

        self.tree[node] = self.combine(
            self.tree[2 * node],
            self.tree[2 * node + 1]
        )

    def _push_down(self, node, start, end):
        """Helper to push down lazy updates"""
        if self.lazy[node] != 0:
            if self.operation == 'sum':
                self.tree[node] += (end - start + 1) * self.lazy[node]
            else:
                self.tree[node] += self.lazy[node]

            if start != end:
                self.lazy[2 * node] += self.lazy[node]
                self.lazy[2 * node + 1] += self.lazy[node]

            self.lazy[node] = 0

    def query_range_lazy(self, node, start, end, l, r):
        """Query with lazy propagation support"""
        self._push_down(node, start, end)
```

```
        if r &lt; start or end &lt; l:
            return self.identity

        if l &lt;= start and end &lt;= r:
            return self.tree[node]

        mid = (start + end) // 2
        left_result = self.query_range_lazy(2 * node, start, mid, l, r)
        right_result = self.query_range_lazy(2 * node + 1, mid + 1, end, l, r)

        return self.combine(left_result, right_result)

    # Public interface methods
    def update(self, idx, val):
        """Public method to update single point"""
        self.update_point(1, 0, self.n - 1, idx, val)

    def query(self, l, r):
        """Public method to query range"""
        return self.query_range(1, 0, self.n - 1, l, r)

    def range_update(self, l, r, val):
        """Public method to update range (with lazy propagation)"""
        self.update_range_lazy(1, 0, self.n - 1, l, r, val)

    def range_query(self, l, r):
        """Public method to query range (with lazy propagation)"""
        return self.query_range_lazy(1, 0, self.n - 1, l, r)

    def get_tree_structure(self):
        """Visualize tree structure for debugging"""
        result = []

        def dfs(node, start, end, depth=0):
            if node &gt;= len(self.tree):
                return

            indent = "  " * depth
            if start == end:
                result.append(f"{indent}Node {node}: [{start}] = {self.tree[node]}")
            else:
                result.append(f"{indent}Node {node}: [{start},{end}] = {self.tree[node]}")

                if 2 * node &lt; len(self.tree):
                    mid = (start + end) // 2
                    dfs(2 * node, start, mid, depth + 1)
                    dfs(2 * node + 1, mid + 1, end, depth + 1)

        dfs(1, 0, self.n - 1)
        return "\n".join(result)

# Specialized segment trees
class RangeMinimumQuery(SegmentTreeComplete):
    """Segment Tree optimized for Range Minimum Query"""
```

```
    def __init__(self, arr):
        super().__init__(arr, 'min')

    def find_min_in_range(self, l, r):
        """Find minimum element in range [l, r]"""
        return self.query(l, r)

    def count_elements_less_than(self, l, r, threshold):
        """Count elements in range [l, r] that are less than threshold"""
        def count_recursive(node, start, end, l, r, threshold):
            if r &lt; start or end &lt; l:
                return 0

            if self.tree[node] &gt;= threshold:
                return 0  # All elements in this range &gt;= threshold

            if start == end:
                return 1 if self.tree[node] &lt; threshold else 0

            mid = (start + end) // 2
            return (count_recursive(2 * node, start, mid, l, r, threshold) +
                    count_recursive(2 * node + 1, mid + 1, end, l, r, threshold))

        return count_recursive(1, 0, self.n - 1, l, r, threshold)

class RangeSumQuery(SegmentTreeComplete):
    """Segment Tree optimized for Range Sum Query with updates"""

    def __init__(self, arr):
        super().__init__(arr, 'sum')

    def get_sum(self, l, r):
        """Get sum of elements in range [l, r]"""
        return self.query(l, r)

    def add_range(self, l, r, val):
        """Add val to all elements in range [l, r]"""
        self.range_update(l, r, val)

    def get_average(self, l, r):
        """Get average of elements in range [l, r]"""
        total_sum = self.query(l, r)
        count = r - l + 1
        return total_sum / count if count &gt; 0 else 0
```

9.2 Fenwick Tree (Binary Indexed Tree)

[215]

Fenwick Tree Theory

Purpose: Efficiently calculate prefix sums and handle point updates.

Key Insight: Use binary representation to determine responsibility of each index.

Bit Magic:

- `i & (-i)`: Isolates lowest set bit
- `i += i & (-i)`: Moves to next responsible index
- `i -= i & (-i)`: Moves to parent in query

Advantages over Segment Tree:

- Less memory: O(n) vs O(4n)
- Simpler implementation
- Better constants

Disadvantages:

- Only works for invertible operations (sum, XOR)
- No range updates (without tricks)

Complete Fenwick Tree Implementation

```
class FenwickTreeComplete:
    """
    Complete Binary Indexed Tree implementation with detailed analysis
    """

    def __init__(self, size_or_array):
        """
        Initialize Fenwick Tree

        Args:
            size_or_array: Either size of array or initial array
        """
        if isinstance(size_or_array, int):
            self.n = size_or_array
            self.tree = [0] * (self.n + 1)  # 1-indexed
            self.original = [0] * (self.n + 1)
        else:
            arr = size_or_array
            self.n = len(arr)
            self.tree = [0] * (self.n + 1)
            self.original = [0] + arr[:]  # Make 1-indexed

            # Build tree
            for i in range(1, self.n + 1):
                self.update(i - 1, arr[i - 1])  # Convert to 0-indexed for public API

    def update(self, idx, val):
        """
        Add val to element at index idx (0-indexed)
```

```
        Algorithm:
        1. Convert to 1-indexed
        2. While within bounds:
           a. Add val to current position
           b. Move to next responsible position using bit magic

        Time: O(log n)
        Space: O(1)
        """
        idx += 1  # Convert to 1-indexed
        delta = val - (self.original[idx] if idx &lt;= len(self.original) - 1 else 0)

        if idx &lt; len(self.original):
            self.original[idx] += delta

        while idx &lt;= self.n:
            self.tree[idx] += delta
            idx += idx &amp; (-idx)  # Add lowest set bit (LSB)

    def prefix_sum(self, idx):
        """
        Get sum of elements from index 0 to idx (inclusive, 0-indexed)

        Algorithm:
        1. Convert to 1-indexed
        2. While index &gt; 0:
           a. Add current tree value to result
           b. Move to parent using bit magic

        Time: O(log n)
        Space: O(1)
        """
        idx += 1  # Convert to 1-indexed
        result = 0

        while idx &gt; 0:
            result += self.tree[idx]
            idx -= idx &amp; (-idx)  # Remove lowest set bit (LSB)

        return result

    def range_sum(self, left, right):
        """
        Get sum of elements in range [left, right] (0-indexed)

        Uses prefix sum property: sum[l,r] = prefix[r] - prefix[l-1]

        Time: O(log n)
        """
        if left == 0:
            return self.prefix_sum(right)
        return self.prefix_sum(right) - self.prefix_sum(left - 1)

    def set_value(self, idx, val):
        """
        Set element at index to specific value (0-indexed)
```

```
        Implementation: update(idx, val - current_value)
        """
        current = self.range_sum(idx, idx)
        self.update(idx, val - current)

    def find_kth_element(self, k):
        """
        Find index of kth smallest element (1-indexed k)

        Uses binary search on Fenwick tree
        Time: O(log²n) or O(log n) with optimizations
        """
        if k &lt;= 0:
            return -1

        # Binary search approach
        pos = 0
        bit_mask = 1

        # Find highest power of 2 &lt;= n
        while bit_mask &lt;= self.n:
            bit_mask &lt;&lt;= 1
        bit_mask &gt;&gt;= 1

        # Binary search using bit manipulation
        while bit_mask &gt; 0:
            next_pos = pos + bit_mask

            if next_pos &lt;= self.n and self.tree[next_pos] &lt; k:
                k -= self.tree[next_pos]
                pos = next_pos

            bit_mask &gt;&gt;= 1

        return pos  # 0-indexed result

    def range_update_point_query(self, left, right, val):
        """
        Add val to all elements in range [left, right]

        Uses difference array technique with Fenwick tree
        Requires separate Fenwick tree for range updates
        """
        # This would require a separate difference array BIT
        # Implementation depends on specific requirements
        pass

    def get_statistics(self):
        """Get comprehensive statistics about the Fenwick tree"""
        total_sum = self.prefix_sum(self.n - 1) if self.n &gt; 0 else 0

        return {
            'size': self.n,
            'total_sum': total_sum,
            'tree_array': self.tree[1:],  # Exclude index 0
```

```
            'max_height': int(self.n.bit_length()) if self.n &gt; 0 else 0,
            'space_usage': len(self.tree) * 4  # Assuming 4 bytes per integer
        }

    def visualize_bit_operations(self, idx):
        """
        Visualize bit operations for understanding

        Shows the path taken during update and query operations
        """
        idx += 1  # Convert to 1-indexed
        original_idx = idx

        print(f"Bit operations for index {original_idx - 1} (0-indexed):")
        print(f"Binary representation: {bin(idx)}")

        # Update path
        print("\nUpdate path:")
        update_idx = idx
        step = 1

        while update_idx &lt;= self.n:
            lsb = update_idx &amp; (-update_idx)
            print(f"  Step {step}: {update_idx} (binary: {bin(update_idx)}) LSB: {lsb}")
            update_idx += lsb
            step += 1

        # Query path
        print("\nQuery path:")
        query_idx = idx
        step = 1

        while query_idx &gt; 0:
            lsb = query_idx &amp; (-query_idx)
            print(f"  Step {step}: {query_idx} (binary: {bin(query_idx)}) LSB: {lsb}")
            query_idx -= lsb
            step += 1

# 2D Fenwick Tree
class FenwickTree2D:
    """
    2D Fenwick Tree for 2D range sum queries
    """

    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
        self.tree = [[0] * (cols + 1) for _ in range(rows + 1)]

    def update(self, row, col, val):
        """Update point (row, col) with value val"""
        row += 1  # Convert to 1-indexed
        col += 1
        orig_col = col

        while row &lt;= self.rows:
```

```
            col = orig_col
            while col &lt;= self.cols:
                self.tree[row][col] += val
                col += col &amp; (-col)
            row += row &amp; (-row)

    def query(self, row, col):
        """Get sum of rectangle from (0,0) to (row,col)"""
        row += 1
        col += 1
        result = 0
        orig_col = col

        while row &gt; 0:
            col = orig_col
            while col &gt; 0:
                result += self.tree[row][col]
                col -= col &amp; (-col)
            row -= row &amp; (-row)

        return result

    def range_query(self, row1, col1, row2, col2):
        """Get sum of rectangle from (row1,col1) to (row2,col2)"""
        result = self.query(row2, col2)

        if row1 &gt; 0:
            result -= self.query(row1 - 1, col2)
        if col1 &gt; 0:
            result -= self.query(row2, col1 - 1)
        if row1 &gt; 0 and col1 &gt; 0:
            result += self.query(row1 - 1, col1 - 1)

        return result

# Applications and advanced techniques
def inversion_count_using_fenwick(arr):
    """
    Count inversions in array using Fenwick tree

    Inversion: pair (i,j) where i &lt; j and arr[i] &gt; arr[j]
    Time: O(n log n)
    """
    # Coordinate compression
    sorted_unique = sorted(set(arr))
    rank_map = {val: i for i, val in enumerate(sorted_unique)}

    fenwick = FenwickTreeComplete(len(sorted_unique))
    inversions = 0

    for i in range(len(arr) - 1, -1, -1):
        rank = rank_map[arr[i]]

        # Count elements smaller than arr[i] that come after i
        inversions += fenwick.prefix_sum(rank - 1) if rank &gt; 0 else 0
```

```
        # Add current element
        fenwick.update(rank, 1)

    return inversions

def range_frequency_queries(arr, queries):
    """
    Answer range frequency queries using multiple Fenwick trees

    Query: count occurrences of value x in range [l, r]
    """
    from collections import defaultdict

    # Create Fenwick tree for each unique value
    fenwick_trees = defaultdict(lambda: FenwickTreeComplete(len(arr)))

    # Build trees
    for i, val in enumerate(arr):
        fenwick_trees[val].update(i, 1)

    results = []
    for l, r, x in queries:
        if x in fenwick_trees:
            count = fenwick_trees[x].range_sum(l, r)
            results.append(count)
        else:
            results.append(0)

    return results
```

Chapter 10: Tree Algorithms & Advanced Techniques

[211] [214]

10.1 Lowest Common Ancestor (LCA) Algorithms

LCA Theory

Definition: The Lowest Common Ancestor of nodes u and v is the deepest node that is an ancestor of both u and v.

Applications:

- Distance between nodes: `dist(u,v) = depth(u) + depth(v) - 2*depth(lca(u,v))`
- Path queries on trees
- Range minimum queries (LCA ↔ RMQ reduction)

Complete LCA Implementations

```
class LCAProcessor:
    """
    Complete LCA implementation with multiple algorithms
    """

    def __init__(self, root, n=None):
        self.root = root
        self.n = n or self._count_nodes(root)

        # For binary lifting
        self.LOG = 20  # ceil(log2(max_n)) + 1
        self.parent = [[-1] * self.LOG for _ in range(self.n)]
        self.depth = [0] * self.n

        # For Euler tour
        self.euler_tour = []
        self.first_occurrence = {}
        self.tour_depth = []

        # Preprocess based on chosen method
        self.preprocess_binary_lifting()
        self.preprocess_euler_tour()

    def _count_nodes(self, root):
        """Count total nodes in tree"""
        if not root:
            return 0
        return 1 + self._count_nodes(root.left) + self._count_nodes(root.right)

    def _get_node_id(self, node):
        """Get unique ID for node (assumes node.val is unique ID)"""
        return node.val if node else -1

    def preprocess_binary_lifting(self):
        """
        Preprocess tree for binary lifting LCA

        Time: O(n log n)
        Space: O(n log n)
        """
        self._dfs_binary_lifting(self.root, -1, 0)

        # Fill binary lifting table
        for j in range(1, self.LOG):
            for i in range(self.n):
                if self.parent[i][j-1] != -1:
                    self.parent[i][j] = self.parent[self.parent[i][j-1]][j-1]

    def _dfs_binary_lifting(self, node, par, d):
        """DFS to fill parent and depth arrays"""
        if not node:
            return

        node_id = self._get_node_id(node)
```

```
        if 0 &lt;= node_id &lt; self.n:
            self.parent[node_id][0] = par
            self.depth[node_id] = d

        self._dfs_binary_lifting(node.left, node_id, d + 1)
        self._dfs_binary_lifting(node.right, node_id, d + 1)

    def lca_binary_lifting(self, u, v):
        """
        Find LCA using binary lifting

        Time: O(log n) per query
        Algorithm:
        1. Bring both nodes to same level
        2. Binary search for LCA
        """
        if u &lt; 0 or u &gt;= self.n or v &lt; 0 or v &gt;= self.n:
            return -1

        # Make sure u is deeper
        if self.depth[u] &lt; self.depth[v]:
            u, v = v, u

        # Bring u to same level as v
        diff = self.depth[u] - self.depth[v]
        for i in range(self.LOG):
            if (diff &gt;&gt; i) &amp; 1:
                u = self.parent[u][i]
                if u == -1:
                    return -1

        if u == v:
            return u

        # Binary search for LCA
        for i in range(self.LOG - 1, -1, -1):
            if (self.parent[u][i] != -1 and
                self.parent[v][i] != -1 and
                self.parent[u][i] != self.parent[v][i]):
                u = self.parent[u][i]
                v = self.parent[v][i]

        return self.parent[u][0]

    def kth_ancestor(self, node, k):
        """
        Find kth ancestor of node using binary lifting

        Time: O(log k)
        """
        if node &lt; 0 or node &gt;= self.n:
            return -1

        for i in range(self.LOG):
            if (k &gt;&gt; i) &amp; 1:
                node = self.parent[node][i]
```

```
            if node == -1:
                break

        return node

    def distance_between_nodes(self, u, v):
        """Calculate distance between two nodes"""
        lca_node = self.lca_binary_lifting(u, v)
        if lca_node == -1:
            return -1

        return self.depth[u] + self.depth[v] - 2 * self.depth[lca_node]

    def preprocess_euler_tour(self):
        """
        Preprocess for Euler tour + RMQ approach to LCA

        Time: O(n) preprocessing + O(n log n) for RMQ
        """
        self._euler_dfs(self.root, 0)

        # Build sparse table for RMQ
        self.sparse_table = SparseTable(self.tour_depth)

    def _euler_dfs(self, node, depth):
        """DFS for Euler tour"""
        if not node:
            return

        node_id = self._get_node_id(node)

        # Add to tour
        self.euler_tour.append(node_id)
        self.tour_depth.append(depth)

        # Record first occurrence
        if node_id not in self.first_occurrence:
            self.first_occurrence[node_id] = len(self.euler_tour) - 1

        # Visit children
        if node.left:
            self._euler_dfs(node.left, depth + 1)
            # Return to current node
            self.euler_tour.append(node_id)
            self.tour_depth.append(depth)

        if node.right:
            self._euler_dfs(node.right, depth + 1)
            # Return to current node
            self.euler_tour.append(node_id)
            self.tour_depth.append(depth)

    def lca_euler_tour(self, u, v):
        """
        Find LCA using Euler tour + RMQ
```

```
        Time: O(1) per query after O(n log n) preprocessing
        """
        if u not in self.first_occurrence or v not in self.first_occurrence:
            return -1

        left = min(self.first_occurrence[u], self.first_occurrence[v])
        right = max(self.first_occurrence[u], self.first_occurrence[v])

        # Find minimum depth in range [left, right]
        min_depth_idx = self.sparse_table.range_minimum_query(left, right)

        return self.euler_tour[min_depth_idx]

    def lca_naive(self, root, p, q):
        """
        Naive LCA algorithm for comparison

        Time: O(n) per query
        """
        if not root:
            return None

        if root.val == p or root.val == q:
            return root.val

        left_lca = self.lca_naive(root.left, p, q)
        right_lca = self.lca_naive(root.right, p, q)

        if left_lca is not None and right_lca is not None:
            return root.val

        return left_lca if left_lca is not None else right_lca

class SparseTable:
    """Sparse Table for Range Minimum Query (used in Euler tour LCA)"""

    def __init__(self, arr):
        self.arr = arr
        self.n = len(arr)
        self.LOG = 20

        # Build sparse table
        self.st = [[0] * self.LOG for _ in range(self.n)]

        # Initialize for intervals of length 1
        for i in range(self.n):
            self.st[i][0] = i

        # Build for all other intervals
        j = 1
        while (1 &lt;&lt; j) &lt;= self.n:
            i = 0
            while (i + (1 &lt;&lt; j) - 1) &lt; self.n:
                left = self.st[i][j-1]
                right = self.st[i + (1 &lt;&lt; (j-1))][j-1]
```

```
                if arr[left] &lt;= arr[right]:
                    self.st[i][j] = left
                else:
                    self.st[i][j] = right

                i += 1
            j += 1

    def range_minimum_query(self, l, r):
        """Find index of minimum element in range [l, r]"""
        length = r - l + 1
        j = 0
        while (1 &lt;&lt; (j + 1)) &lt;= length:
            j += 1

        left = self.st[l][j]
        right = self.st[r - (1 &lt;&lt; j) + 1][j]

        if self.arr[left] &lt;= self.arr[right]:
            return left
        else:
            return right
```

10.2 Tree Diameter and Path Algorithms

Tree Diameter Theory

Definition: The diameter of a tree is the longest path between any two nodes.

Key Insight: The diameter path doesn't necessarily pass through the root.

Complete Diameter Implementation

```
class TreePathAlgorithms:
    """
    Complete implementation of tree path algorithms
    """

    def __init__(self):
        self.max_diameter = 0
        self.diameter_path = []

    def find_diameter(self, root):
        """
        Find diameter of tree

        Algorithm:
        1. For each node, calculate max path through that node
        2. Max path = left_height + right_height
        3. Track global maximum

        Time: O(n), Space: O(h)
        """
        self.max_diameter = 0
```

```
        self.diameter_path = []

        def dfs(node):
            if not node:
                return 0, []

            # Get heights and paths from children
            left_height, left_path = dfs(node.left)
            right_height, right_path = dfs(node.right)

            # Current diameter through this node
            current_diameter = left_height + right_height

            # Update global diameter if necessary
            if current_diameter &gt; self.max_diameter:
                self.max_diameter = current_diameter
                # Construct diameter path
                self.diameter_path = (left_path[::-1] +
                                     [node.val] +
                                     right_path)

            # Return height and path of taller subtree
            if left_height &gt; right_height:
                return left_height + 1, left_path + [node.val]
            else:
                return right_height + 1, right_path + [node.val]

        dfs(root)
        return self.max_diameter, self.diameter_path

    def find_diameter_two_dfs(self, adj_list, n):
        """
        Find diameter using two DFS calls (for general trees)

        Algorithm:
        1. DFS from any node to find farthest node
        2. DFS from that node to find farthest from it
        3. Distance in step 2 is the diameter

        Time: O(n), Space: O(n)
        """
        def dfs(start, adj):
            visited = [False] * n
            distances = [0] * n
            queue = [start]
            visited[start] = True
            farthest = start
            max_dist = 0

            while queue:
                node = queue.pop(0)

                for neighbor in adj[node]:
                    if not visited[neighbor]:
                        visited[neighbor] = True
                        distances[neighbor] = distances[node] + 1
```

```
                        queue.append(neighbor)

                        if distances[neighbor] &gt; max_dist:
                            max_dist = distances[neighbor]
                            farthest = neighbor

            return farthest, max_dist

        # First DFS from node 0
        farthest_from_0, _ = dfs(0, adj_list)

        # Second DFS from farthest node found
        farthest_from_farthest, diameter = dfs(farthest_from_0, adj_list)

        return diameter, (farthest_from_0, farthest_from_farthest)

    def maximum_path_sum(self, root):
        """
        Find maximum path sum in tree

        Path can start and end at any nodes
        Time: O(n), Space: O(h)
        """
        max_sum = [float('-inf')]

        def dfs(node):
            if not node:
                return 0

            # Get maximum path sums from subtrees (ignore negative)
            left_sum = max(0, dfs(node.left))
            right_sum = max(0, dfs(node.right))

            # Update global maximum (path through current node)
            max_sum[0] = max(max_sum[0], node.val + left_sum + right_sum)

            # Return maximum path sum starting from current node
            return node.val + max(left_sum, right_sum)

        dfs(root)
        return max_sum[0]

    def find_all_paths_with_sum(self, root, target_sum):
        """
        Find all root-to-leaf paths with given sum

        Time: O(n * h) in worst case
        """
        all_paths = []

        def dfs(node, current_path, current_sum):
            if not node:
                return

            # Add current node to path
            current_path.append(node.val)
```

```
            current_sum += node.val

            # Check if leaf node with target sum
            if not node.left and not node.right:
                if current_sum == target_sum:
                    all_paths.append(current_path[:])
            else:
                # Continue search in subtrees
                dfs(node.left, current_path, current_sum)
                dfs(node.right, current_path, current_sum)

            # Backtrack
            current_path.pop()

        dfs(root, [], 0)
        return all_paths

    def count_paths_with_sum(self, root, target_sum):
        """
        Count all paths (not necessarily root-to-leaf) with given sum

        Uses prefix sum technique with backtracking
        Time: O(n), Space: O(n)
        """
        def dfs(node, current_sum, prefix_sums):
            if not node:
                return 0

            current_sum += node.val

            # Count paths ending at current node
            count = prefix_sums.get(current_sum - target_sum, 0)

            # Add current sum to prefix sums
            prefix_sums[current_sum] = prefix_sums.get(current_sum, 0) + 1

            # Recurse on children
            count += dfs(node.left, current_sum, prefix_sums)
            count += dfs(node.right, current_sum, prefix_sums)

            # Remove current sum (backtrack)
            prefix_sums[current_sum] -= 1
            if prefix_sums[current_sum] == 0:
                del prefix_sums[current_sum]

            return count

        return dfs(root, 0, {0: 1})  # Initialize with empty path

    def find_path_between_nodes(self, root, start, end):
        """
        Find path between two nodes in tree

        Algorithm:
        1. Find LCA of start and end
        2. Get path from start to LCA
```

```
        3. Get path from LCA to end
        4. Combine paths
        """
        def find_path_to_node(node, target, path):
            if not node:
                return False

            path.append(node.val)

            if node.val == target:
                return True

            if (find_path_to_node(node.left, target, path) or
                find_path_to_node(node.right, target, path)):
                return True

            path.pop()
            return False

        # Find paths to both nodes from root
        path_to_start = []
        path_to_end = []

        if not find_path_to_node(root, start, path_to_start):
            return []
        if not find_path_to_node(root, end, path_to_end):
            return []

        # Find LCA (last common node in paths)
        lca_idx = 0
        min_len = min(len(path_to_start), len(path_to_end))

        while (lca_idx &lt; min_len and
               path_to_start[lca_idx] == path_to_end[lca_idx]):
            lca_idx += 1

        lca_idx -= 1  # Last common index

        # Construct path: start -&gt; LCA -&gt; end
        result_path = path_to_start[lca_idx:][::-1][1:]  # Reverse and remove LCA
        result_path.extend(path_to_end[lca_idx:])        # Add LCA to end

        return result_path
```

10.3 Tree DP and Advanced Techniques

Tree Dynamic Programming

Key Insight: Use tree structure to avoid overlapping subproblems.

Complete Tree DP Implementation

```
class TreeDP:
    """
    Complete Tree Dynamic Programming implementations
    """

    def maximum_independent_set(self, root):
        """
        Find maximum weight independent set in tree

        Independent set: no two adjacent nodes

        DP state:
        - include[node] = max weight including current node
        - exclude[node] = max weight excluding current node

        Time: O(n), Space: O(n)
        """
        def dp(node):
            if not node:
                return 0, 0  # (include, exclude)

            left_include, left_exclude = dp(node.left)
            right_include, right_exclude = dp(node.right)

            # Include current node: cannot include children
            include = node.val + left_exclude + right_exclude

            # Exclude current node: can choose optimally for children
            exclude = (max(left_include, left_exclude) +
                      max(right_include, right_exclude))

            return include, exclude

        include, exclude = dp(root)
        return max(include, exclude)

    def tree_coloring_ways(self, root, colors):
        """
        Count ways to color tree nodes such that adjacent nodes have different colors

        DP state: dp[node][color] = ways to color subtree with node colored 'color'

        Time: O(n * colors), Space: O(n * colors)
        """
        memo = {}

        def dp(node, parent_color):
            if not node:
                return 1

            if (node, parent_color) in memo:
                return memo[(node, parent_color)]

            total_ways = 0
```

```python
            # Try each color for current node (except parent's color)
            for color in range(colors):
                if color != parent_color:
                    left_ways = dp(node.left, color)
                    right_ways = dp(node.right, color)
                    total_ways += left_ways * right_ways

            memo[(node, parent_color)] = total_ways
            return total_ways

        return dp(root, -1)  # Root has no parent constraint

    def subtree_sizes(self, root):
        """
        Calculate size of each subtree

        Time: O(n), Space: O(n)
        """
        sizes = {}

        def dfs(node):
            if not node:
                return 0

            left_size = dfs(node.left)
            right_size = dfs(node.right)

            current_size = 1 + left_size + right_size
            sizes[node.val] = current_size

            return current_size

        dfs(root)
        return sizes

    def tree_rerooting_dp(self, adj_list, n):
        """
        Tree rerooting technique for calculating answers with each node as root

        Example: Calculate sum of distances from each node to all other nodes

        Two-pass algorithm:
        1. First DFS: calculate answer assuming node 0 is root
        2. Second DFS: reroot and recalculate answers

        Time: O(n), Space: O(n)
        """
        # First DFS: calculate subtree sizes and initial answer
        subtree_size = [0] * n
        dp_down = [0] * n  # Answer for subtree

        def dfs1(node, parent):
            subtree_size[node] = 1
            dp_down[node] = 0
```

```
            for neighbor in adj_list[node]:
                if neighbor != parent:
                    dfs1(neighbor, node)
                    subtree_size[node] += subtree_size[neighbor]
                    dp_down[node] += dp_down[neighbor] + subtree_size[neighbor]

        # Second DFS: reroot and calculate answers
        dp_up = [0] * n    # Answer considering parent contribution
        answer = [0] * n   # Final answer for each node as root

        def dfs2(node, parent):
            answer[node] = dp_down[node] + dp_up[node]

            for neighbor in adj_list[node]:
                if neighbor != parent:
                    # Calculate dp_up for neighbor
                    # Remove neighbor's contribution from current node
                    remaining_down = dp_down[node] - dp_down[neighbor] - subtree_size[neigh
                    remaining_size = n - subtree_size[neighbor]

                    dp_up[neighbor] = dp_up[node] + remaining_down + remaining_size

                    dfs2(neighbor, node)

        dfs1(0, -1)
        dp_up[0] = 0
        dfs2(0, -1)

        return answer

    def tree_matching(self, root):
        """
        Find maximum matching in tree

        Matching: set of edges with no common vertices

        DP states:
        - matched[node] = max matching if current node is matched with parent
        - not_matched[node] = max matching if current node is not matched with parent
        """
        def dp(node, parent):
            if not node:
                return 0, 0

            not_matched = 0  # Current node not matched with parent

            for child in [node.left, node.right]:
                if child and child != parent:
                    child_matched, child_not_matched = dp(child, node)
                    not_matched += max(child_matched, child_not_matched)

            # If current node is matched with parent
            matched = 1  # Edge between current and parent
            for child in [node.left, node.right]:
                if child and child != parent:
                    _, child_not_matched = dp(child, node)
```

```
                matched += child_not_matched

            return matched, not_matched

        if not root:
            return 0

        # Root has no parent, so it's effectively "not matched" with parent
        _, result = dp(root, None)
        return result

# Centroid Decomposition
class CentroidDecomposition:
    """
    Centroid Decomposition for tree path queries
    """

    def __init__(self, adj_list, n):
        self.adj = adj_list
        self.n = n
        self.removed = [False] * n
        self.subtree_size = [0] * n
        self.centroid_parent = [-1] * n

        self.decompose(0, -1)

    def get_subtree_size(self, node, parent):
        """Calculate subtree size excluding removed nodes"""
        self.subtree_size[node] = 1

        for neighbor in self.adj[node]:
            if neighbor != parent and not self.removed[neighbor]:
                self.subtree_size[node] += self.get_subtree_size(neighbor, node)

        return self.subtree_size[node]

    def find_centroid(self, node, parent, tree_size):
        """Find centroid of current tree"""
        for neighbor in self.adj[node]:
            if (neighbor != parent and
                not self.removed[neighbor] and
                self.subtree_size[neighbor] &gt; tree_size // 2):
                return self.find_centroid(neighbor, node, tree_size)

        return node

    def decompose(self, node, parent):
        """Recursively decompose tree using centroids"""
        tree_size = self.get_subtree_size(node, -1)
        centroid = self.find_centroid(node, -1, tree_size)

        self.removed[centroid] = True
        self.centroid_parent[centroid] = parent

        # Process centroid (problem-specific logic would go here)
        self.process_centroid(centroid)
```

```
        # Recursively decompose remaining subtrees
        for neighbor in self.adj[centroid]:
            if not self.removed[neighbor]:
                self.decompose(neighbor, centroid)

    def process_centroid(self, centroid):
        """Process the current centroid (implement based on problem)"""
        # This is where problem-specific logic would be implemented
        # For example: counting paths, updating data structures, etc.
        pass

    def count_paths_through_centroid(self, centroid, target_sum):
        """
        Count paths passing through centroid with given sum
        Example application of centroid decomposition
        """
        from collections import defaultdict

        def get_paths_from_subtree(start, parent, current_sum, paths):
            paths.append(current_sum)

            for neighbor in self.adj[start]:
                if neighbor != parent and not self.removed[neighbor]:
                    get_paths_from_subtree(neighbor, start, current_sum + neighbor, paths)

        count = 0
        all_paths = []

        # Get paths from each subtree rooted at centroid's children
        for child in self.adj[centroid]:
            if not self.removed[child]:
                subtree_paths = []
                get_paths_from_subtree(child, centroid, child, subtree_paths)

                # Count paths using existing paths
                for path_sum in subtree_paths:
                    complement = target_sum - centroid - path_sum
                    for existing_path in all_paths:
                        if existing_path == complement:
                            count += 1

                all_paths.extend(subtree_paths)

        # Count paths that end at centroid
        for path_sum in all_paths:
            if path_sum + centroid == target_sum:
                count += 1

        # Count path that is just centroid
        if centroid == target_sum:
            count += 1

        return count
```

This comprehensive handbook continues with interview patterns, real-world applications, and complete code implementations. Would you like me to continue with the remaining chapters covering interview patterns, complexity analysis, and practical applications?