Trees

Hierarchical Data Structures

Minseok Jeon
DGIST

November 2, 2025

Outline

Introduction to Trees

What is a Tree?

Definition: A hierarchical data structure consisting of nodes connected by edges.

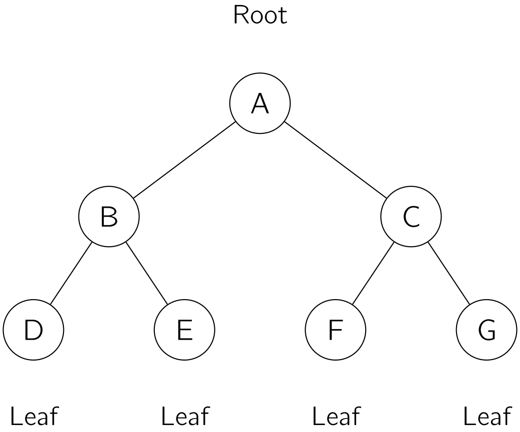
Key Properties:

- Exactly one path between any two nodes
- One designated **root** node at the top
- No cycles (acyclic graph)
- Each node has zero or more **children**

Tree Terminology:

- **Root:** The topmost node
- **Parent:** A node that has children
- **Child:** A node connected to a parent
- **Leaf:** A node with no children
- **Height:** Longest path from root to a leaf
- **Depth:** Distance from root to a node

Basic Tree Structure



- **Root:** A

Why Use Trees?

Advantages:

- **Natural hierarchy:** File systems, organizational charts
- **Efficient search:** O(log n) in balanced trees
- **Fast insertion/deletion:** Compared to sorted arrays
- **Flexible structure:** Adapts to different data patterns

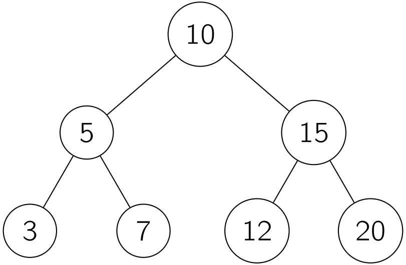
Common Applications:

- Database indexing (B-trees)
- File systems (directory trees)
- Expression parsing (syntax trees)
- Decision making (decision trees)
- Network routing
- Compression algorithms (Huffman trees)

Binary Trees

Binary Trees

Definition: A tree where each node has at most two children (left and right).

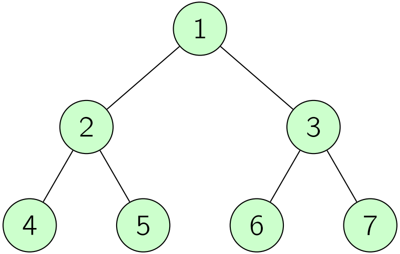


Properties:

- Each node: at most 2 children
- Left child $<$ Parent (in BST)
- Right child $>$ Parent (in BST)

Types of Binary Trees: Full Binary Tree

Full Binary Tree: Every node has either 0 or 2 children (no nodes with 1 child).

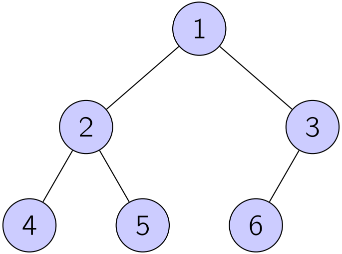


Characteristics:

- All internal nodes have exactly 2 children
- All leaves at same or adjacent levels
- Useful for expression trees

Types of Binary Trees: Complete Binary Tree

Complete Binary Tree: All levels filled except possibly the last, which is filled from left to right.

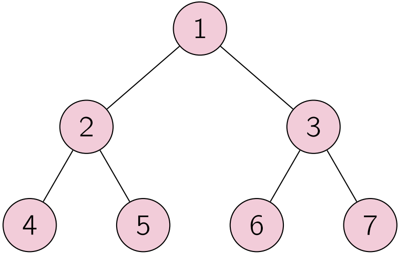


Characteristics:

- Used in heap data structures
- Can be efficiently stored in arrays
- Left-to-right filling at each level

Types of Binary Trees: Perfect Binary Tree

Perfect Binary Tree: All internal nodes have 2 children and all leaves are at the same level.



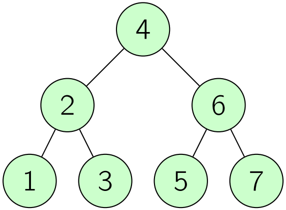
Properties:

- Total nodes = $2^{h+1} - 1$ (h = height)
- All leaves at level h
- Maximally space-efficient

Types of Binary Trees: Balanced vs Skewed

Balanced Binary Tree

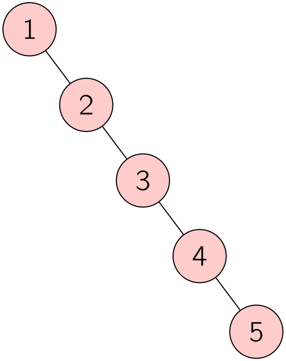
Height difference ≤ 1 for all nodes



Height: O(log n)

Skewed Binary Tree

All nodes only have one child



Height: O(n)

Binary Tree Implementation

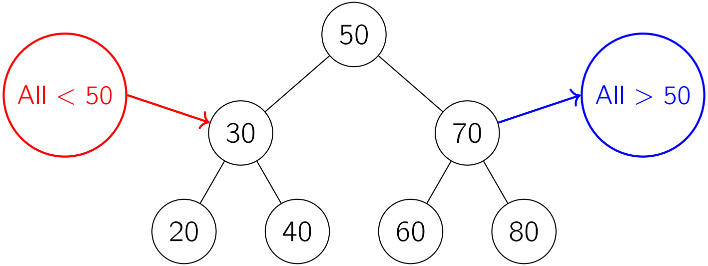
```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
```

Binary Search Trees (BST)

Binary Search Tree (BST)

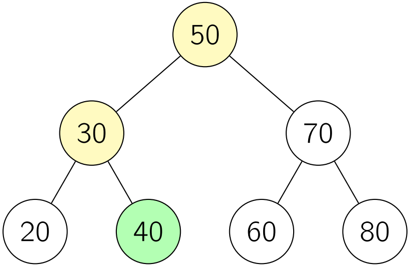
Definition: A binary tree where for every node:

- All values in left subtree < node value
- All values in right subtree > node value
- Both subtrees are also BSTs



BST Search Operation

Algorithm: Start at root and compare target with current node.



Search for 40:

1. Start at 50: $40 < 50 \to$ go left

BST Search Implementation

```python
def search(self, value):
    """Search for a value in the BST"""
    return self._search_recursive(self.root, value)

def _search_recursive(self, node, value):
    # Base case: empty tree or value found
    if node is None or node.value == value:
        return node

    # Value is smaller: search left subtree
    if value < node.value:
        return self._search_recursive(node.left, value)

    # Value is larger: search right subtree
    return self._search_recursive(node.right, value)

# Iterative version
```

BST Insert Operation

```python
def insert(self, value):
    """Insert a value into the BST"""
    if not self.root:
        self.root = TreeNode(value)
    else:
        self._insert_recursive(self.root, value)

def _insert_recursive(self, node, value):
    # Insert in left subtree
    if value < node.value:
        if node.left is None:
            node.left = TreeNode(value)
        else:
            self._insert_recursive(node.left, value)
    # Insert in right subtree
    else:
        if node.right is None:
```

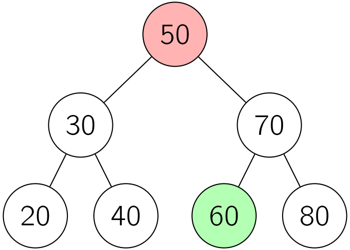
BST Delete Operation: Three Cases

Case 1: Node is a leaf → Simply remove it

Case 2: Node has one child → Replace with child

Case 3: Node has two children → Replace with:

- **Inorder successor:** Smallest value in right subtree
- **Inorder predecessor:** Largest value in left subtree



BST Delete Implementation

```python
def delete(self, value):
    self.root = self._delete_recursive(self.root, value)

def _delete_recursive(self, node, value):
    if node is None:
        return None

    if value < node.value:
        node.left = self._delete_recursive(node.left, value)
    elif value > node.value:
        node.right = self._delete_recursive(node.right, value)
    else:
        # Case 1: Leaf or one child
        if node.left is None:
            return node.right
        if node.right is None:
            return node.left

        # Case 2: Two children - find inorder successor
```

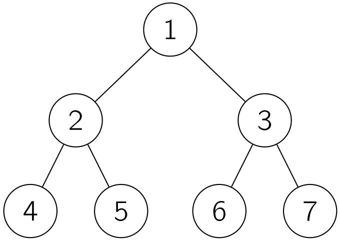
Tree Traversals

Tree Traversal Overview

Traversal: Visiting all nodes in a specific order.

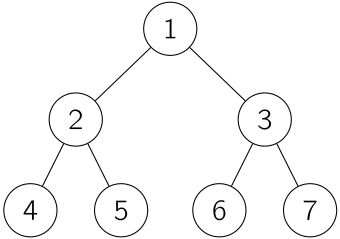
Four Main Types:

1. **Inorder (Left-Root-Right):** Produces sorted order in BST
2. **Preorder (Root-Left-Right):** Used for copying trees
3. **Postorder (Left-Right-Root):** Used for deleting trees
4. **Level-order (Breadth-first):** Level by level



Inorder Traversal (Left-Root-Right)

Order: Left subtree → Root → Right subtree



Result: 4, 2, 5, 1, 6, 3, 7

Use Cases:

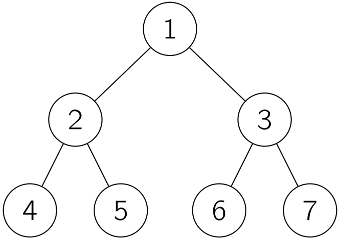
- Getting sorted order from BST
- Expression tree evaluation

Inorder Traversal Implementation

```python
def inorder(self):
    """Return list of values in inorder"""
    result = []
    self._inorder_recursive(self.root, result)
    return result

def _inorder_recursive(self, node, result):
    if node:
        # Left subtree
        self._inorder_recursive(node.left, result)
        # Root
        result.append(node.value)
        # Right subtree
        self._inorder_recursive(node.right, result)

# Iterative version using stack
def inorder_iterative(self):
```

Preorder Traversal (Root-Left-Right)

Order: Root → Left subtree → Right subtree



Result: 1, 2, 4, 5, 3, 6, 7

Use Cases:

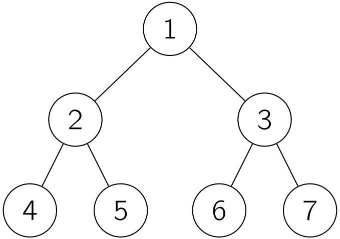
- Creating a copy of the tree
- Prefix expression of an expression tree
- Serializing a tree

Preorder Traversal Implementation

```
def preorder(self):
    """Return list of values in preorder"""
    result = []
    self._preorder_recursive(self.root, result)
    return result

def _preorder_recursive(self, node, result):
    if node:
        # Root
        result.append(node.value)
        # Left subtree
        self._preorder_recursive(node.left, result)
        # Right subtree
        self._preorder_recursive(node.right, result)

# Iterative version
def preorder_iterative(self):
```

Postorder Traversal (Left-Right-Root)

Order: Left subtree → Right subtree → Root



Result: 4, 5, 2, 6, 7, 3, 1

Use Cases:

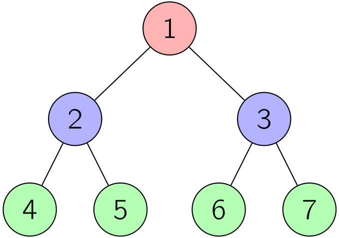
- Deleting a tree (delete children before parent)
- Postfix expression of an expression tree
- Computing directory sizes in file system

Postorder Traversal Implementation

```python
def postorder(self):
    """Return list of values in postorder"""
    result = []
    self._postorder_recursive(self.root, result)
    return result

def _postorder_recursive(self, node, result):
    if node:
        # Left subtree
        self._postorder_recursive(node.left, result)
        # Right subtree
        self._postorder_recursive(node.right, result)
        # Root
        result.append(node.value)

# Iterative version using two stacks
def postorder_iterative(self):
```

Level-Order Traversal (Breadth-First)

Order: Visit nodes level by level, left to right



Level 0: 1
Level 1: 2, 3
Level 2: 4, 5, 6, 7

Result: 1, 2, 3, 4, 5, 6, 7

Use Cases:

- Finding shortest path in unweighted tree

Level-Order Traversal Implementation

```python
from collections import deque

def level_order(self):
    """Return list of values in level-order"""
    if not self.root:
        return []

    result = []
    queue = deque([self.root])

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

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

Heaps

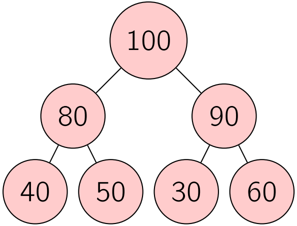
Heap Data Structure

Definition: A complete binary tree satisfying the heap property.

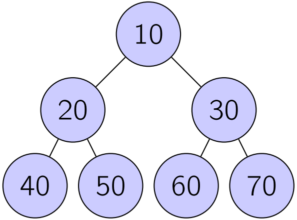
Two Types:

- **Max Heap:** Parent $\geq$ children (root is maximum)
- **Min Heap:** Parent $\leq$ children (root is minimum)

Max Heap



Min Heap

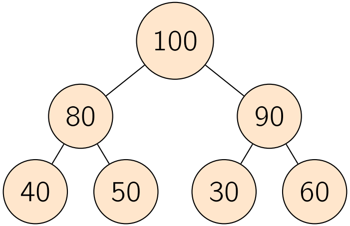


Heap Array Representation

Key Insight: Complete binary tree can be stored efficiently in an array.

Index Relationships:

- **Parent of i:** $(i-1)/2$
- **Left child of i:** $2i+1$
- **Right child of i:** $2i+2$



Array: [100, 80, 90, 40, 50, 30, 60]
Indices: 0 1 2 3 4 5 6

Heap Insert Operation

Algorithm:

1. Add new element at the end (maintain complete tree property)
2. **Heapify Up:** Compare with parent and swap if needed
3. Repeat until heap property restored

```python
def insert(self, value):
    """Insert value into max heap"""
    self.heap.append(value)
    self._heapify_up(len(self.heap) - 1)

def _heapify_up(self, index):
    parent = (index - 1) // 2

    # Max heap: if current > parent, swap
    if index > 0 and self.heap[index] > self.heap[parent]:
        self.heap[index], self.heap[parent] = \
            self.heap[parent], self.heap[index]
```

Heap Extract Max/Min Operation

Algorithm:

1. Remove and return root (max/min element)
2. Replace root with last element
3. **Heapify Down:** Compare with children and swap with larger/smaller
4. Repeat until heap property restored

```
def extract_max(self):
    if not self.heap:
        return None

    max_val = self.heap[0]
    self.heap[0] = self.heap[-1]
    self.heap.pop()
    self._heapify_down(0)
    return max_val

def _heapify_down(self, index):
    largest = index
```

Heap Applications

Priority Queue:

- Task scheduling (OS process scheduling)
- Event-driven simulation
- Dijkstra's shortest path algorithm

Heap Sort:

- Build max heap from array
- Repeatedly extract max
- O(n log n) time, O(1) space

Top-K Problems:

- Find K largest/smallest elements
- Maintain min/max heap of size K
- Streaming data scenarios

Median Finding:

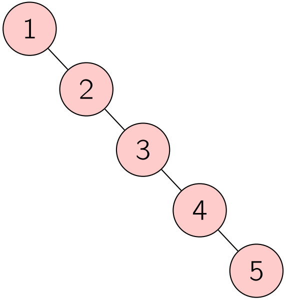
- Two heaps: max heap for lower half, min heap for upper half

Balanced Trees

Why Balanced Trees?

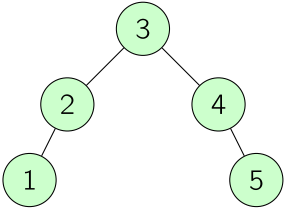
Problem with BST: Can degenerate to O(n) operations in worst case.

Worst Case BST



Height = n
Operations: O(n)

Balanced BST



Height = log n
Operations: O(log n)

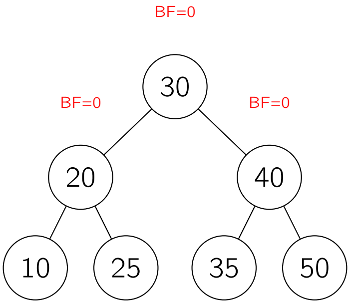
Solution: Self-balancing trees (AVL, Red-Black)

AVL Trees

Definition: BST where height difference of left and right subtrees ≤ 1 for all nodes.

Balance Factor: height(left) - height(right)

- Must be -1, 0, or +1 for all nodes



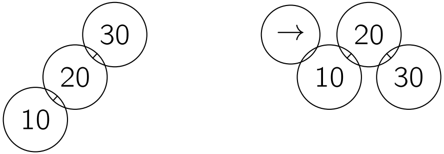
Properties:

- Height is always O(log n)

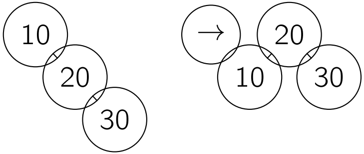
AVL Tree Rotations

Four Rotation Cases:

1. Left-Left (LL)
Single right rotation



2. Right-Right (RR)
Single left rotation



3. Left-Right (LR)
Left rotate, then right rotate

4. Right-Left (RL)
Right rotate, then left rotate

AVL Tree Rotation Implementation

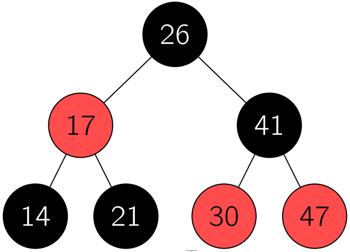
```
def _rotate_right(self, y):
    """Right rotation"""
    x = y.left
    T2 = x.right

    x.right = y
    y.left = T2

    y.height = 1 + max(self._get_height(y.left),
                       self._get_height(y.right))
    x.height = 1 + max(self._get_height(x.left),
                       self._get_height(x.right))
    return x

def _rotate_left(self, x):
    """Left rotation"""
    y = x.right
    T2 = y.left
```

Red-Black Trees

Definition: BST with additional color property for balancing.

Properties:

1. Every node is either red or black
2. Root is always black
3. All leaves (NIL) are black
4. Red nodes have black children (no two red nodes in a row)
5. All paths from root to leaves have same number of black nodes



Red-Black Trees vs AVL Trees

Feature	AVL Tree	Red-Black Tree
Balance	Strictly balanced	Approximately balanced
Height	$\leq 1.44 \log n$	$\leq 2 \log n$
Search	Faster	Slightly slower
Insert/Delete	More rotations	Fewer rotations
Use case	Read-heavy	Insert/delete-heavy
Examples	Databases	Java TreeMap, C++ map

Key Takeaway:

- AVL: Better for lookup-intensive applications
- Red-Black: Better for applications with frequent insertions/deletions

Real-World Applications

Database Indexing

B-Trees and B+ Trees: Variants of balanced trees used in databases.

Why Trees for Databases?

- Fast search: O(log n)
- Efficient range queries
- Good disk I/O performance (nodes = disk blocks)
- Support for ordered traversal

B+ Tree Features:

- All data in leaf nodes
- Internal nodes only store keys
- Leaves linked for sequential access
- Used in: MySQL, PostgreSQL, SQLite

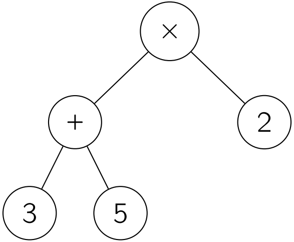
Example:

- Index on employee ID
- Fast lookups: SELECT * WHERE id = 12345

Expression Parsing

Syntax Trees: Represent mathematical or code expressions.

Example: $(3 + 5) \times 2$



Evaluation:

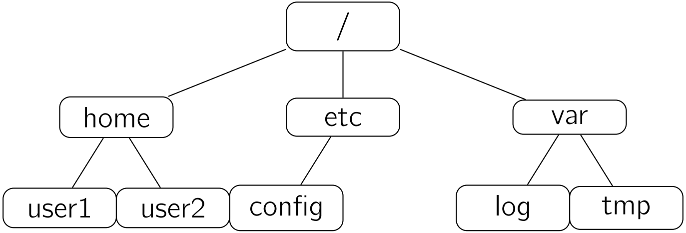
- Postorder traversal: 3, 5, +, 2, $\times$
- Result: $(3 + 5) = 8$, then $8 \times 2 = 16$

Applications:

- Compilers and interpreters

File Systems

Directory Structure: Trees represent hierarchical file organization.

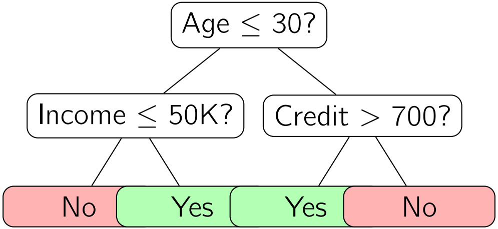


Operations:

- Navigate directories: O(depth)
- List contents: Visit children
- Calculate directory size: Postorder traversal
- Search for files: DFS or BFS

Decision Trees (Machine Learning)

Decision Trees: Tree-based models for classification and regression.



Properties:

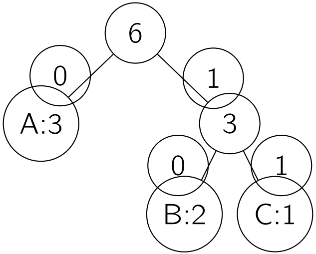
- Internal nodes: Decision rules
- Leaves: Predictions (class labels or values)
- Path from root to leaf: Decision path

Algorithms: ID3, C4.5, CART, Random Forests, Gradient Boosting

Huffman Coding (Compression)

Huffman Tree: Optimal prefix-free binary code for data compression.

Example: Compress "AAABBC"



Encoding:

- A: 0, B: 10, C: 11
- "AAABBC" $\rightarrow$ 0 0 0 10 10 11 (10 bits vs 18 bits)

Used in: ZIP, JPEG, MP3

Complexity Analysis

Tree Operations Complexity

Operation	BST (avg)	BST (worst)	Balanced
Search	O(log n)	O(n)	O(log n)
Insert	O(log n)	O(n)	O(log n)
Delete	O(log n)	O(n)	O(log n)
Find Min/Max	O(log n)	O(n)	O(log n)
Traversal	O(n)	O(n)	O(n)

Key Points:

- Unbalanced BST: Worst case O(n) when tree becomes skewed
- Balanced trees (AVL, Red-Black): Guaranteed O(log n)
- Traversals always O(n) - must visit every node

Heap Operations Complexity

Operation	Time Complexity
Insert	O(log n)
Extract Max/Min	O(log n)
Get Max/Min (peek)	O(1)
Build Heap	O(n)
Heapify	O(log n)
Heap Sort	O(n log n)

Space Complexity: O(n)

Note:

- Get max/min is O(1) - root element
- Build heap from array is O(n), not O(n log n)

Space Complexity

Tree Storage:

Structure	Space
Binary Tree (pointer-based)	O(n)
Complete Binary Tree (array)	O(n)
AVL Tree (with height)	O(n)
Red-Black Tree (with color)	O(n)
Heap (array-based)	O(n)

Additional Space:

- Recursive traversals: O(h) stack space
- Iterative traversals with stack/queue: O(h) or O(w)
  - h = height, w = maximum width
- Level-order traversal: O(w) for queue

Summary

Key Concepts Recap

Binary Trees:

- Hierarchical structure with at most 2 children per node
- Types: Full, Complete, Perfect, Balanced, Skewed

Binary Search Trees:

- Ordered binary tree: left $<$ root $<$ right
- Operations: Search, Insert, Delete - O(h)
- Can degenerate to O(n) without balancing

Traversals:

- Inorder (sorted), Preorder (copy), Postorder (delete), Level-order

Heaps:

- Complete binary tree with heap property
- Priority queue, Heap sort, Top-K problems

Key Concepts Recap (continued)

Balanced Trees:

- AVL: Strict balance, faster search
- Red-Black: Approximate balance, faster insert/delete
- Both guarantee O(log n) operations

Applications:

- Database indexing (B-trees, B+ trees)
- Expression parsing (syntax trees)
- File systems (directory trees)
- Machine learning (decision trees)
- Compression (Huffman coding)

Complexity:

- Balanced trees: O(log n) for search, insert, delete
- Heaps: O(log n) for insert/extract, O(1) for peek
- Traversals: Always O(n)

When to Use Which Tree?

Use Case	Tree Type
Fast search in sorted data	BST, AVL, Red-Black
Priority queue	Min/Max Heap
Frequent inserts/deletes	Red-Black Tree
Lookup-heavy workload	AVL Tree
Database indexing	B-Tree, B+ Tree
Expression evaluation	Syntax Tree
File system	N-ary Tree
Decision making	Decision Tree
Compression	Huffman Tree
Range queries	Segment Tree, Interval Tree

Practice Problems

Basic:

- Implement inorder, preorder, postorder traversals
- Find height of a binary tree
- Check if a binary tree is a valid BST
- Find lowest common ancestor (LCA)

Intermediate:

- Serialize and deserialize a binary tree
- Convert sorted array to balanced BST
- Implement AVL tree with rotations
- Find kth smallest element in BST

Advanced:

- Implement Red-Black tree
- Morris traversal (constant space)
- Segment tree for range queries
- Trie for prefix matching

Next Steps

Continue Learning:

- Advanced trees: Trie, Segment Tree, Fenwick Tree
- Graph algorithms (trees are special graphs)
- Dynamic programming with trees
- Practice on LeetCode, HackerRank

Resources:

- CLRS: Introduction to Algorithms
- GeeksforGeeks tree tutorials
- Visualgo.net for visualizations
- Project: Implement your own balanced tree library

Real-World Projects:

- Build a simple database with B-tree indexing
- Create an expression evaluator
- Implement file compression with Huffman coding
- Design a decision tree classifier

Thank You!

Questions?

Trees: The Foundation of Hierarchical Thinking