Fundamentals

Why we need tree?

We've already learned a few *linear data structures* — arrays, linked lists, stacks, and queues. What these structures have in common is that the data is arranged in a linear direction. Each element has exactly one predecessor and one successor.

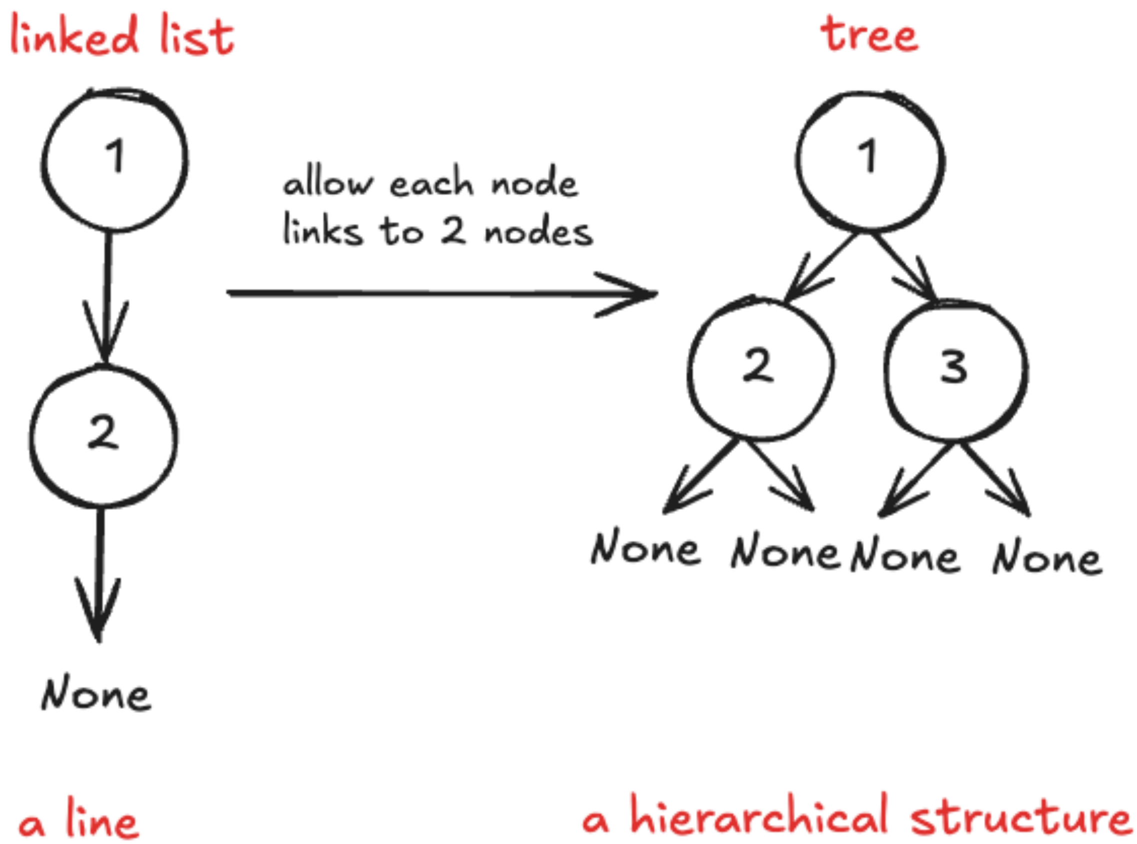
But in the real world, data relationships are rarely that simple. Many systems have a *branching* or *layered* structure. For example:

- A **file system**: root folder → subfolder → file
- A **website navigation**: homepage → category → subcategory → page

In these examples, one node can have *multiple children*, not just a single "next" element. That kind of structure is called a **hierarchical structure**.

So, why do we need trees?

Because trees let us *efficiently represent and manipulate* this kind of hierarchical data. Whenever relationships involve "containment," "dependency," or "inheritance," a tree is often the most natural way to model them.



Definition and Properties

A **tree** is a **non-linear data structure** that represents hierarchical relationships.

It's made up of a collection of **nodes** connected by **edges**, starting from a single **root node** and expanding downward into **children**.

Let's highlight a few key properties:

- **One unique root:** There is exactly one root node, and all nodes are reachable from it.
- **Parent–child structure:** Every node (except the root) has exactly one parent. Edges represent these parent–child connections.
- **No cycles:** Trees contain no loops — you can't start from a node and come back to it by following edges. That's the main difference between a *tree* and a *graph*.

You can also think of a tree as a *recursive structure*.

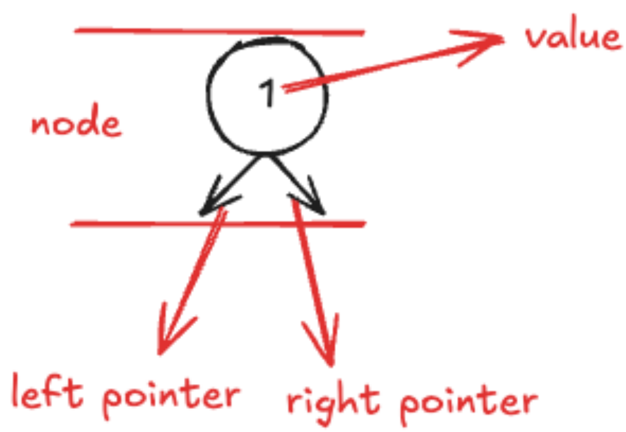
A tree is either empty, or it consists of a root node and several subtrees. Each child itself is the root of another smaller tree — and that's exactly why recursion fits trees so naturally.

Terminologies

Let's go through the most important terms you'll use again and again when talking about trees.

Node

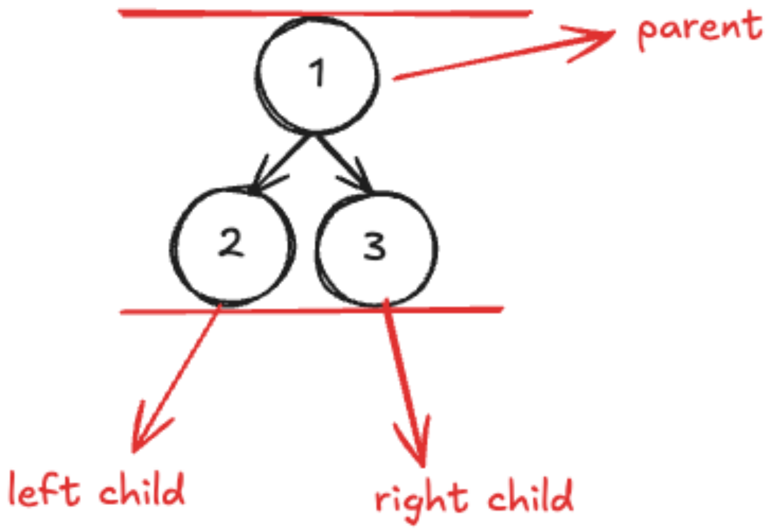
Each element in a tree is called a "node." Nodes represent entities and may contain data.



Parent / Child

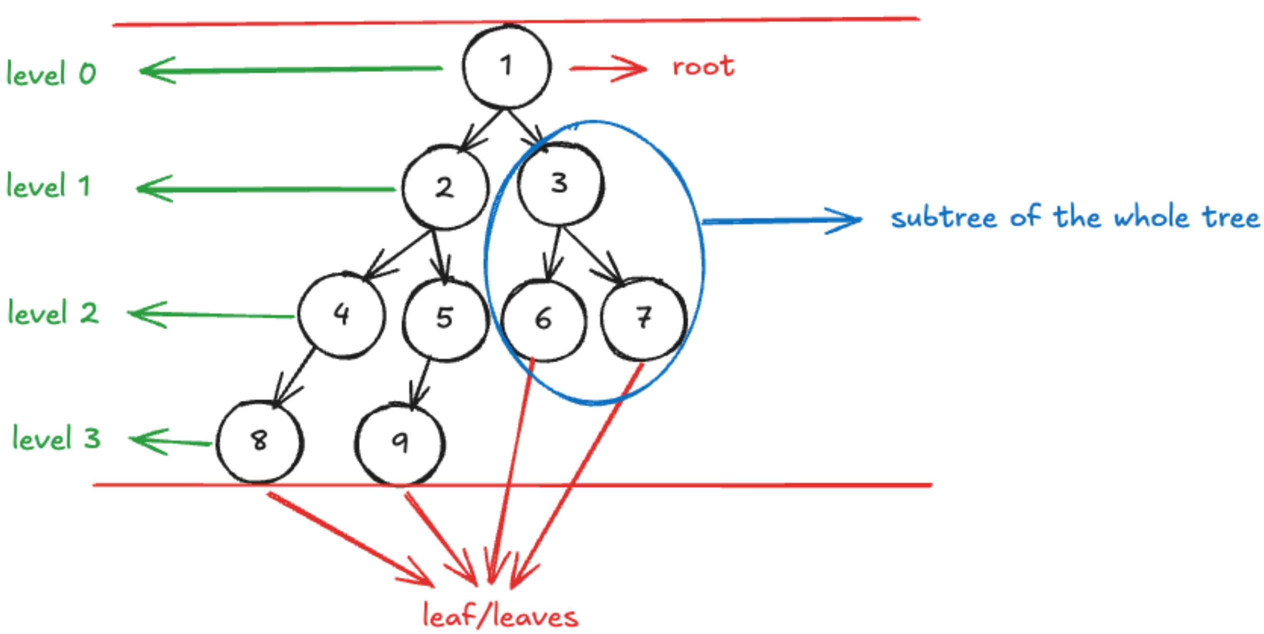
Nodes are connected by edges.

If a node connects to another node *below* it, we call the first one the **parent**, and the ones below it its **children**.



Root / Leaf / Subtree / Level

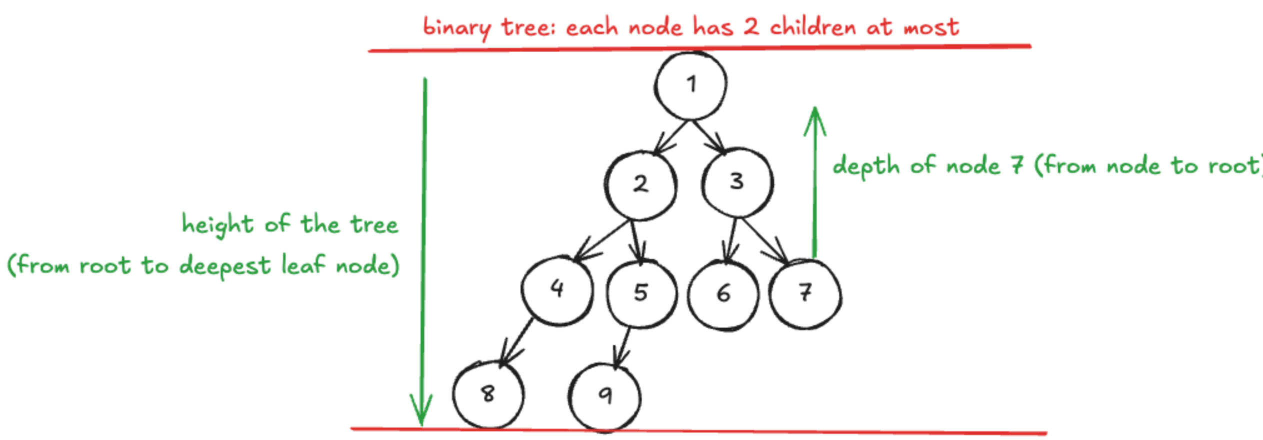
- The **root** is the topmost node — it's where the tree begins.
- A **leaf** (or terminal node) is a node that has no children.
- A **subtree** consists of a node and all its descendants.
- The **level** of a node is how far it is from the root — the root is level 0, its children are level 1, and so on.



Depth / Height / Binary Tree

Depth and Height are two concepts that are often confused.

- Depth — *"How far am I from the root?"*
  - The depth of a node is the number of edges from the root to that node. The root's depth is 0.
  - The depth of a tree is the maximum depth among all nodes.
- Height — *"How far am I from the leaves?"*
  - The height of a node is the number of edges on the longest path from that node down to a leaf. A leaf's height is 0.
  - The height of a tree is the height of its root.
- A **binary tree** is a tree in which each node has at most two children — a left child and a right child.

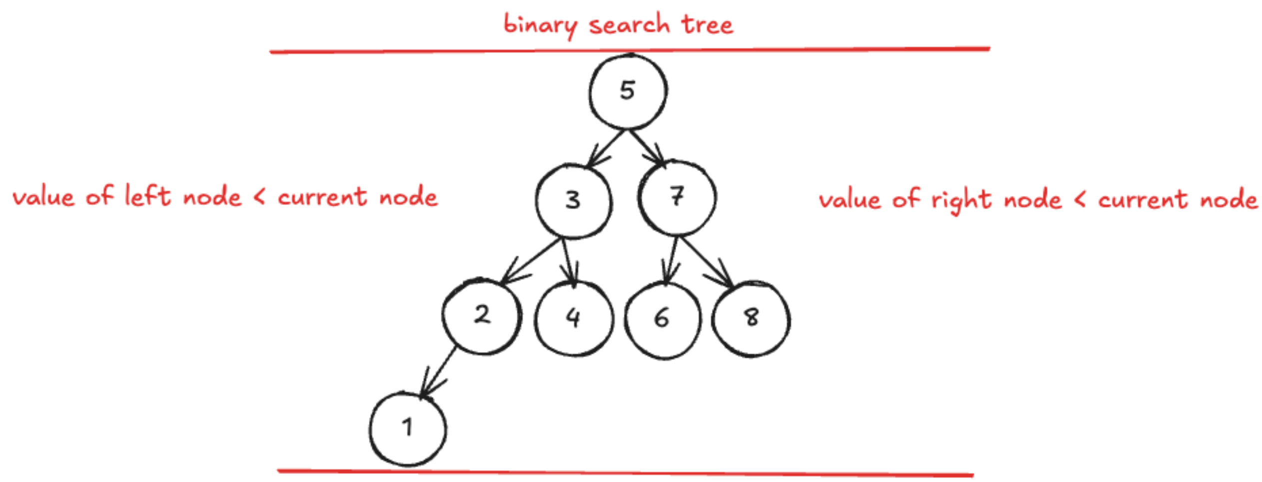


Binary Search Tree

A **binary search tree** is a special kind of binary tree that satisfies the **ordering property**:

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

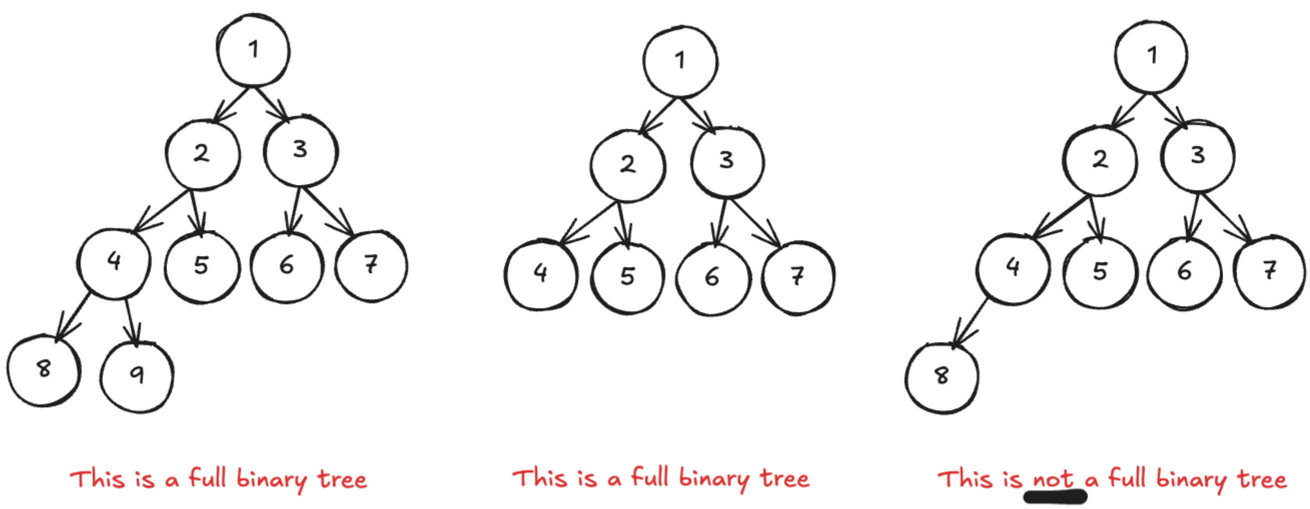
This property allows efficient searching, insertion, and deletion.



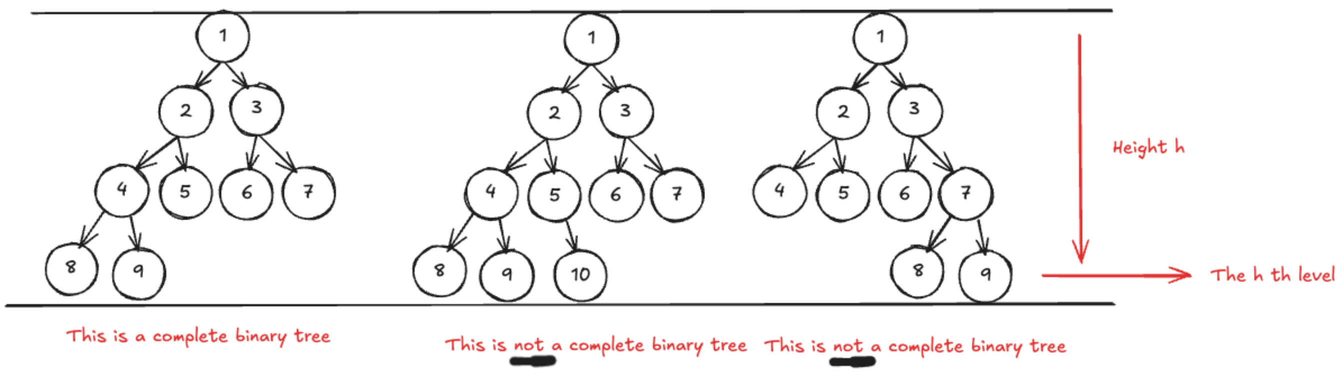
Full, Complete, and Balanced Binary Trees

Let’s go through three special types of binary trees you’ll encounter often.

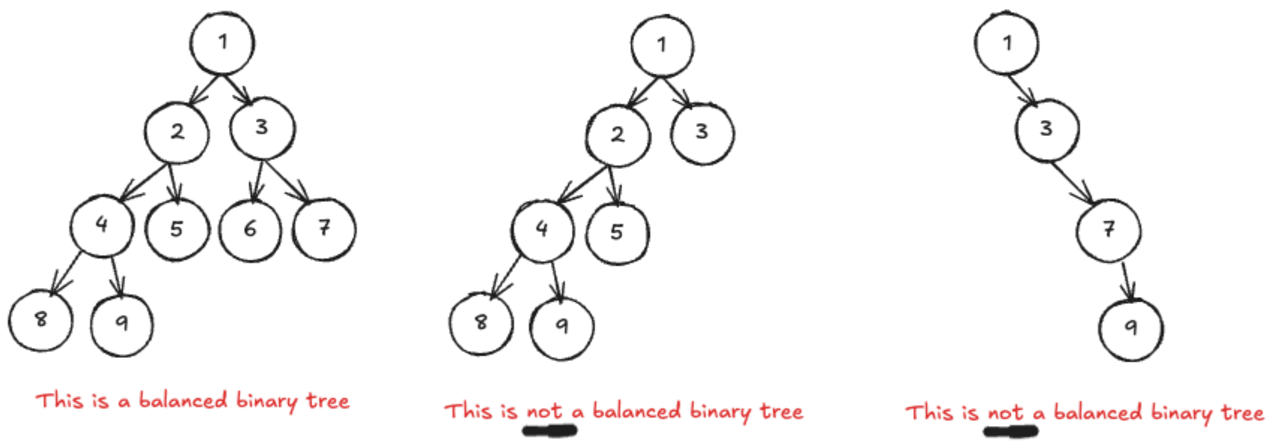
- **Full Binary Tree**: Every node has either *two children* or *none*. In other words, no node has only one child.



- **Complete Binary Tree**: A tree is *complete* if all levels are completely filled except possibly the last one, and the last level’s nodes are filled from left to right without gaps.



- **Balanced Binary Tree**: For every node, the height difference between its left and right subtrees is at most 1. Common examples include **AVL Trees** (strictly balanced), **Red–Black Trees** (loosely balanced), and **B / B+ Trees** used in database indexing.



Ways to Represent a Tree

There are two common ways to represent trees in code.

Parent Array Representation

We can use an array `parent[]` to record each node's parent index.

- The **index** represents the node itself.

- The **value** represents its parent.

```
        0
      / | \
     1  2  3
       / \
      4   5
```

The parent array looks like this:

```
index:   0  1  2  3  4  5
parent: -1  0  0  0  2  2
```

- Node 0's parent is `-1` — meaning it's the root.
- Nodes 1, 2, 3 have parent 0.
- Nodes 4, 5 have parent 2.

This method is compact and simple — perfect for *static trees* that don't change often. It's fast to find a node's parent (`O(1)`), but to find children, you have to scan the entire array (`O(n)`).

Linked Structure (TreeNode Class)

A more flexible way is to use a **linked structure**. Each node stores its own references to its children.

Each node typically contains:

- a value `val`
- pointers to child nodes (e.g., `left`, `right`, or a list of `children`)

```python
# Binary Tree Node
class TreeNode:
    def __init__(self, value=0):
        self.val = value
        self.left = None
        self.right = None

# N-ary Tree Node
class TreeNode:
    def __init__(self, value=0):
        self.val = value
        self.children = []
```

This representation is flexible for insertions and deletions, and it works naturally with **recursive** algorithms like

This representation is flexible for insertions and deletions, and it works naturally with **recursive** algorithms like DFS or BFS.

However, it does require extra space for pointers, and it doesn't support random access like arrays do.

Traversal

Before we go any further, we need to talk about **tree traversal —** in other words, how we visit every node in a tree in a particular order.

Unlike arrays or linked lists, trees are *not linear*. There's no single "next" element. So, we must define the order we visit nodes.

In general, there are two main types of traversal:

- **Depth-First Search (DFS) –** we go as deep as possible before backtracking.
- **Breadth-First Search (BFS) –** we visit level by level, from top to bottom.

Depth-First Search (DFS)

The core idea is: **go all the way down one path before turning back.**

In a binary tree, we have three common ways to order the visit:

Type	Visiting Order	Code
Preorder	Root → left → right	`visit(root); dfs(left); dfs(right)`
Inorder	left → Root → right	`dfs(left); visit(root); dfs(right)`
Postorder	left → right → Root	`dfs(left); dfs(right); visit(root)`

```
        1
       / \
      2   3
     / \
    4   5
```

- Preorder → `[1, 2, 4, 5, 3]`
- Inorder → `[4, 2, 5, 1, 3]`
- Postorder → `[4, 5, 2, 3, 1]`

- Postorder → `[4, 5, 2, 3, 1]`

Every recursive call is processing a **subtree**. So the logic is always "do something with the current node, then do the same thing with its children."

```
# Preorder
def preorder(root):
    if not root:
      return

    print(root.val)
    preorder(root.left)
    preorder(root.right)

# Inorder
def preorder(root):
    if not root:
      return

    preorder(root.left)
    print(root.val)
    preorder(root.right)

# Postorder
def preorder(root):
    if not root:
      return

    preorder(root.left)
    preorder(root.right)
    print(root.val)
```

We can also implement DFS **iteratively** using a **stack**:

```
def preorder_iterative(root):
    if not root:
        return

    stack = [root]
    while stack:
        node = stack.pop()
        print(node.val, end=' ')
        # Push right first so left is processed first
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
```

And for an **N-ary tree** (where each node can have multiple children):

And for an **N-ary tree** (where each node can have multiple children):

```
def dfs(root):
    if not root:
        return
    print(root.val, end=' ')
    for child in root.children:
        dfs(child)
```

Breadth-First Search (BFS)

BFS, or **level-order traversal**, visits nodes level by level, from top to bottom, left to right or right to left.

```
        1
       / \
      2   3
     / \
    4   5
```

Output: `[1, 2, 3, 4, 5]`

We typically use a **queue** for BFS:

```
from collections import deque

def level_order(root):
    if not root:
        return

    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

Trie

Now that we've explored binary trees and N-ary trees, let's look at another very special type of tree — the **Trie**

Now that we've explored binary trees and N-ary trees, let's look at another very special type of tree — the **Trie** (also called a *prefix tree* or *dictionary tree*).

A Trie is still a tree, but it's built specifically for **storing and searching strings efficiently**, especially when we care about **prefixes**.

Think about a dictionary. Suppose we store these words:

```
["cat", "car", "dog"]
```

If we use a normal list, checking whether any word starts with `"ca"` would take O(n·m) time — n words, each of length m.

But with a Trie, we share common prefixes and reduce the time to **O(m)**, where m is the prefix length.

Here's what the Trie looks like after inserting those three words:

```
(root)
 ├── c
 │    └── a
 │         ├── t (end)
 │         └── r (end)
 └── d
      └── o
           └── g (end)
```

Structure

Each node represents a prefix, and it stores:

- a map of children (character → next node)
- a flag `isEnd` that marks the end of a word

```
class TrieNode:
    def __init__(self):
        self.children = {}   # e.g. {'a': TrieNode(), 'b': TrieNode(), ...}
        self.isEnd = False   # marks the end of a valid word
```

The Trie itself just has a single root node:

```
class Trie:
    def __init__(self):
        self.root = TrieNode()
```

Operations

A Trie supports three basic operations:

- **Insert** a word by moving down the tree, creating nodes if necessary. When you reach the end, mark `isEnd = True`.

```
# Insert
def insert(self, word):
    node = self.root
    for ch in word:
        if ch not in node.children:
            node.children[ch] = TrieNode()
        node = node.children[ch]
    node.isEnd = True
```

- **Search** for a word by following the characters one by one. If any character path is missing, the word doesn't exist.

```
# Search
def search(self, word):
    node = self.root
    for ch in word:
        if ch not in node.children:
            return False
        node = node.children[ch]
    return node.isEnd
```

- **Check whether a given prefix exists** in the Trie — we don't need to reach the end of a word.

```
# Prefix Match
def startsWith(self, prefix):
    node = self.root
    for ch in prefix:
        if ch not in node.children:
            return False
        node = node.children[ch]
    return True
```

Operation	Time Complexity	Description

Operation	Time Complexity	Description
Insert	O(L)	L = length of the word
Search	O(L)	character-by-character lookup
Prefix match	O(L)	no need to reach word end

Key Techniques

Recursive Thinking

When solving tree problems, recursion is almost always at the core.

Return

So, before you even start coding, ask yourself one question: “What does my recursive function return?”

That single question determines how you’ll design your recursion.

Return Type	Example	Purpose
Value (int / sum)	maximum depth, path sum	aggregate information
Node (TreeNode)	lowest common ancestor (LCA)	return a target node
Boolean (bool)	isBalanced, isSameTree	decision problems
Tuple (multi-value)	height + balance flag	composite return

Divide & Conquer Template

This is the fundamental pattern behind almost every *bottom-up* tree problem — like maximum depth, tree diameter, path sum, balance check, or lowest common ancestor.

Think of it like “divide” = go down, and “conquer” = come back up and combine results.

Think of it like "divide" — go down, and "conquer" — come back up and combine results.

```
def helper(root):
    # 1. base case
    if not root:
        return ...

    # 2. divide
    left = helper(root.left)
    right = helper(root.right)

    # 3. conquer / combine
    res = process(root, left, right)

    return res
```

Top-down vs Bottom-up

This is the part that many people feel confused about — both are recursive, but the **direction of information flow** is different.

Approach	Key Idea	When to Use
Top-down (from root downward)	pass down current state (path, depth, etc.)	when each node depends on information from ancestors
Bottom-up (from leaves upward)	return subtree's information and combine it	when parent's result depends on children's data

For example:

- Top-down："Given the current depth, compute each node's depth."
- Bottom-up："Return subtree depth to parent, and combine to find the overall max."

So, top-down propagates **context**, bottom-up aggregates **results**.

Supporting Data Structures

Trees pair naturally with other structures — understanding when to use which data structure makes a huge difference.

Stack / Queue / Deque

Stack / Queue / Deque

Structure	Role	Example
Stack	iterative DFS	preorder, inorder, postorder traversal
Queue	BFS traversal	level order, tree width
Deque	flexible double-end operations	zigzag level order

HashMap

We often use a hash map to keep **relationships** or **fast lookup** information.

- Map values to nodes → quick `O(1)` access
- Record parent pointers → useful for upward traversal (like LCA)
- Track visited nodes → avoid revisiting in cyclic or reconstructed graphs

Common use cases:

- Building a tree from preorder + inorder (use an inorder index map)
- Lowest Common Ancestor with a `parent` dictionary
- Path-sum problems (store prefix sums)

Global Variables

Sometimes, one recursive function can't return all the information you need. That's when we use **a global variable** to store a running maximum or result.

Global variables let recursion *return one thing* but still *track another* globally.

```
global_max = float('-inf')

def dfs(root):
    global global_max
    if not root:
        return 0
    left = max(dfs(root.left), 0)
    right = max(dfs(root.right), 0)
    global_max = max(global_max, left + right + root.val)
    return max(left, right) + root.val
```

> This pattern is common in problems like:

> - Maximum path sum
> - Longest diameter
> - Deepest node in a tree

Problem Patterns

At the key point of all tree problems are three key questions:

Step	Ask Yourself
Traversal	How do I visit every node? (DFS or BFS)
Return Value	What information do I need from children?
Processing Time	When should I handle the logic? (Pre / In / Post order)

Traversal

These are the most fundamental. You define an order (preorder, inorder, postorder, or level order) and apply a simple operation at each node.

- 144. Binary Tree Preorder Traversal
- 94. Binary Tree Inorder Traversal
- 145. Binary Tree Postorder Traversal
- 102. Binary Tree Level Order Traversal
- 103. Binary Tree Zigzag Level Order Traversal
- 107. Binary Tree Level Order Traversal II
- 199. Binary Tree Right Side View
- 545. Boundary of Binary Tree

Construction & Reconstruction

Here, the goal is to **rebuild** the tree based on given traversal sequences. You usually divide traversal arrays into left and right subtrees and build recursively.

- 105. Construct Binary Tree from Preorder and Inorder Traversal
- 106. Construct Binary Tree from Inorder and Postorder Traversal
- 889. Construct Binary Tree from Preorder and Postorder Traversal
- 1028. Recover a Tree From Preorder Traversal
- 297. Serialize and Deserialize Binary Tree

Path Problems

These focus on **paths and cumulative values**. We often carry current sums or track paths during DFS, and use a global variable to track the best result.

Pattern summary: “Record the path → update global result → backtrack.”

- 112. Path Sum
- 113. Path Sum II
- 437. Path Sum III
- 666. Path Sum IV
- 124. Binary Tree Maximum Path Sum
- 129. Sum Root to Leaf Numbers
- 1022. Sum of Root To Leaf Binary Numbers
- 3590. Kth Smallest Path XOR Sum

Tree Structure

Here, we’re comparing or matching tree structures. Most are *double-recursion problems* — you compare the current pair of nodes, then compare their children.

- 100. Same Tree
- 572. Subtree of Another Tree
- 101. Symmetric Tree

Depth / Height

These rely on bottom-up recursion. Each call returns its subtree’s height, and you combine that information to compute things like tree height, balance, diameter, or width.

- 104. Maximum Depth of Binary Tree
- 111. Minimum Depth of Binary Tree
- 110. Balanced Binary Tree
- 543. Diameter of Binary Tree
- 1245. Tree Diameter
- 662. Maximum Width of Binary Tree

BST

In BST problems, you'll leverage the property **left < root < right** for ordered operations. They often combine recursion with binary search–style logic.

- 450. Delete Node in a BST
- 285. Inorder Successor in BST
- 510. Inorder Successor in BST II
- 98. Validate Binary Search Tree
- 230. Kth Smallest Element in a BST
- 776. Split BST
- 333. Largest BST Subtree
- 449. Serialize and Deserialize BST
- 449. Serialize and Deserialize BST

N-ary Tree

Now we move beyond binary trees — each node can have multiple children. The traversal logic remains the same; the only difference is iterating over a list instead of two branches.

- 589. N-ary Tree Preorder Traversal
- 590. N-ary Tree Postorder Traversal
- 429. N-ary Tree Level Order Traversal
- 559. Maximum Depth of N-ary Tree
- 1522. Diameter of N-Ary Tree
- 428. Serialize and Deserialize N-ary Tree

Trie

Finally, we come back to the Trie — the prefix tree. These problems revolve around prefix search, replacement, or word construction, often mixing DFS with Trie operations.

- 212. Word Search II
- 648. Replace Words
- 208. Implement Trie (Prefix Tree)
- 211. Design Add and Search Words Data Structure