

# Trees

Data Structures and Algorithms Tutorial

---

IIITH - S26

Binary Trees, Traversals, Applications

# Agenda

1. Overview
2. Binary Trees
3. Tree Traversals
4. Practice Problems
5. Summary

# Overview

---

# What is a Tree?

## Definition

- Hierarchical data structure with nodes connected by edges
- One node designated as **root**
- Each node (except root) has exactly one parent
- No cycles!

## Key Terms

- **Root**: Topmost node (no parent)
- **Child**: Node directly connected to another node when moving away from root
- **Parent**: Node directly connected to another node when moving toward root
- **Leaf**: Node with no children
- **Depth**: Distance from root
- **Height**: Maximum distance to any leaf descendant

# Why Trees?

## Real-world Examples

- File system hierarchy
- HTML DOM structure
- Organization charts
- Family trees
- Decision trees
- Parse trees in compilers

## Computational Advantages

- Efficient representation of hierarchies
- Natural recursive structure
- Fast lookup with balanced trees
- Flexible size (dynamic growth)

# Binary Trees

---

# Binary Tree: Definition

## Main Idea

- Each node has at most **two** children
- Children designated as **left** and **right**

## Properties

- Maximum nodes at level  $i$ :  $2^i$
- Maximum nodes in tree of height  $h$ :  $2^{h+1} - 1$
- Minimum height for  $n$  nodes:  $\lceil \log_2(n + 1) \rceil - 1$

# Binary Tree: Node Structure

Memory: Each node stores data + 2 pointers

```
typedef struct Node {  
    int data;  
    struct Node *left;  
    struct Node *right;  
} Node;
```

## Creating a Node

```
Node* createNode(int value) {  
    Node* newNode = (Node*)malloc(sizeof(Node));  
    newNode->data = value;  
    newNode->left = NULL;  
    newNode->right = NULL;  
    return newNode;  
}
```

# Tree Properties

## Height vs Depth

- **Depth**: Distance from root (root has depth 0)
- **Height**: Longest path to a leaf
- Height of tree = max height of any node

## Full Binary Tree

- Every node has 0 or 2 children

# Tree Traversals

---

# Tree Traversals: Overview

## What is Traversal?

- Visiting every node exactly once
- Processing node data during visit

## Types of Traversals

- Depth-First Search (DFS)
  - Preorder (Root, Left, Right)
  - Inorder (Left, Root, Right)
  - Postorder (Left, Right, Root)
- Breadth-First Search (BFS)
  - Level Order (level by level)

All traversals are  $O(n)$  time,  $O(h)$  space

# Preorder Traversal

Order: Root -> Left -> Right

Algorithm

1. Visit (process) root
2. Traverse left subtree
3. Traverse right subtree

Use Cases

- Creating copy of tree
- Prefix expression evaluation
- When you want to process root before children

Example: For tree 1-(2,3) with 2 having children 4,5 Output: 1 2 4 5 3

# Inorder Traversal

Order: Left -> Root -> Right

Algorithm

1. Traverse left subtree
2. Visit root
3. Traverse right subtree

Use Cases

- Infix expression evaluation
- When you need sorted order from special trees

Example: For tree 1-(2,3) with 2 having children 4,5 Output: 4 2 5 1 3

# Postorder Traversal

Order: Left -> Right -> Root

Algorithm

1. Traverse left subtree
2. Traverse right subtree
3. Visit root

Use Cases

- Deleting tree (must delete children first)
- Postfix expression evaluation
- Computing subtree properties (size, height)

Example: For tree 1-(2,3) with 2 having children 4,5 Output: 4 5 2 3 1

# Level Order Traversal (BFS)

## Main Idea

- Visit nodes level by level, left to right
- Uses **queue** (FIFO) for traversal

## Algorithm

1. Enqueue root
2. While queue not empty:
  - Dequeue node, process it
  - Enqueue left child (if exists)
  - Enqueue right child (if exists)

Example: For tree 1-(2,3) with 2 having children 4,5 Output: 1 2 3 4 5

## Use Cases

- Finding shortest path
- Level-based problems
- Minimum depth of tree

# Practice Problems

---

## Problem 1: Maximum Depth

Problem Statement Find the maximum depth (height) of a binary tree.

## Intuition

- For ANY node, it's depth would be 1 plus whoever is greater from left or right?
- If a node is NULL, its depth is 0 (base case).
- So it leads to this recursive solution where you just run the function over and over again and add 1 for the current node and take max for left and right.

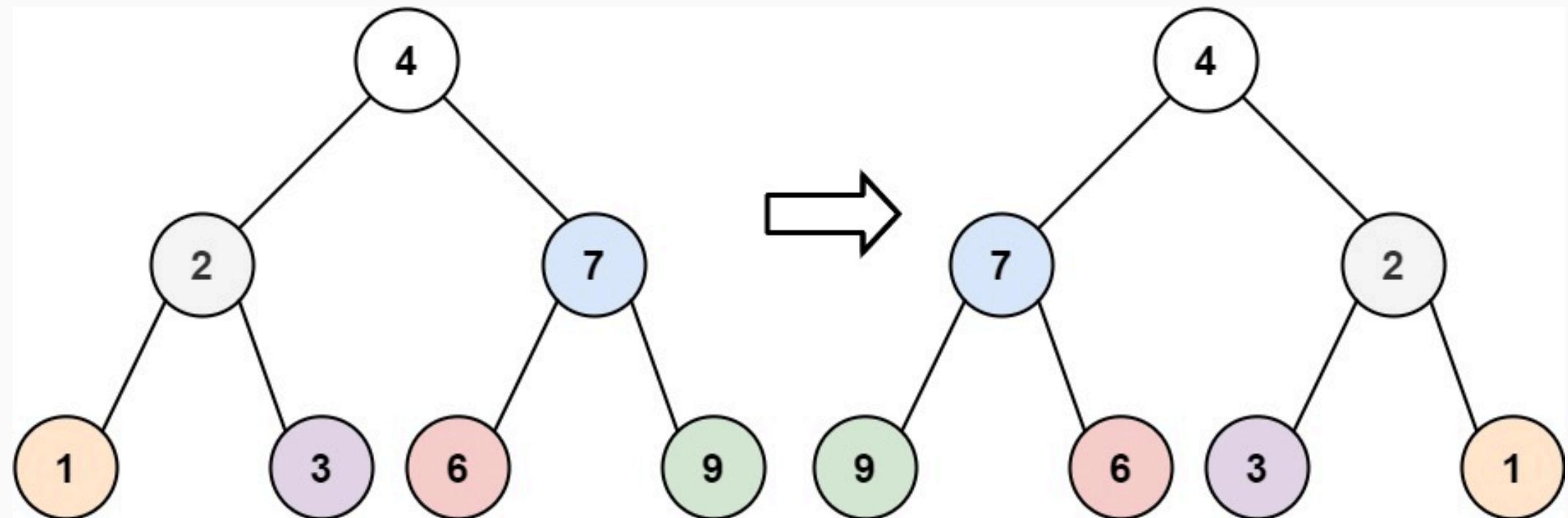
## Solution

```
int maxDepth(Node* root) {  
    if (root == NULL) return 0;  
    return 1 + max(maxDepth(root->left), maxDepth(root->right));  
}
```

Time:  $O(n)$  | Space:  $O(h)$

## Problem 2: Invert Binary Tree

Problem Statement Invert a binary tree (mirror image).



## Intuition

- To no one's surprise, it's recursion again :D
- Inverting a tree means mirroring it, and mirror will be local. So you only need to swap left and right of the current node.
- Once current is done, you can do it for their children as well.

## Solution

```
Node* invertTree(Node* root) {  
    if (root == NULL) return NULL;  
    Node* temp = root->left;  
    root->left = invertTree(root->right);  
    root->right = invertTree(temp);  
    return root;  
}
```

Time:  $O(n)$  | Space:  $O(h)$

## Problem 3: Same Tree

Problem Statement Check if two binary trees are identical.

### Intuition

- Again, it's mostly “local”.
- Both NULL: same
- One NULL, one not: different
- Different values: different
- Recursively check left and right subtrees

## Solution

```
int isSameTree(Node* p, Node* q) {  
    if (p == NULL && q == NULL) return 1;  
    if (p == NULL || q == NULL) return 0;  
    return (p->data == q->data) &&  
        isSameTree(p->left, q->left) &&  
        isSameTree(p->right, q->right);  
}
```

Time:  $O(n)$  | Space:  $O(h)$

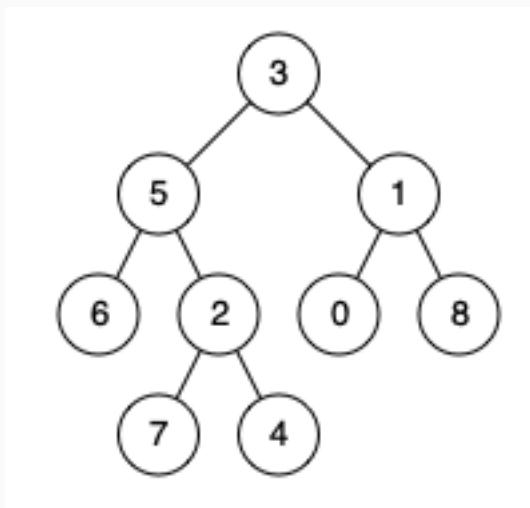
## Problem 4: Lowest Common Ancestor

Problem Statement Find LCA of two nodes in a binary tree.

Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.



## Intuition

- If the current node is p or q, we return it -> as it could be the answer.
- Recursively search the left and right subtrees to see where p and q are located.
- If one subtree returns p and the other returns q, then the current node is the first point where their paths meet, so it is the LCA.
- If one side is Null and other is not, it means both p and q lie in THAT exact subtree, so we pass that upwards.

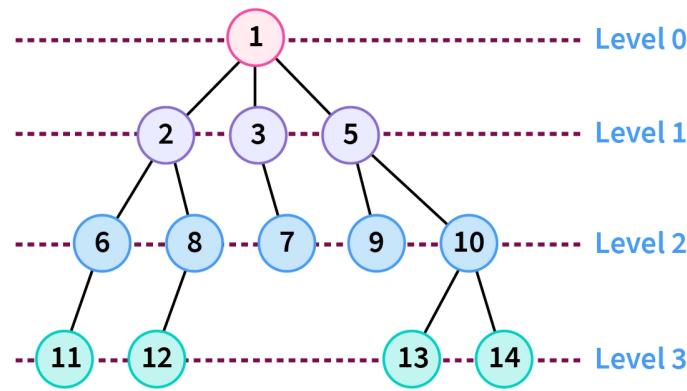
## Solution

```
Node* LCA(Node* root, Node* p, Node* q) {  
    if (root == NULL || root == p || root == q)  
        return root;  
    Node* left = LCA(root->left, p, q);  
    Node* right = LCA(root->right, p, q);  
    if (left && right) return root;  
    return left ? left : right;  
}
```

Time:  $O(n)$  | Space:  $O(h)$

# Problem 5: Level Order Traversal

Problem Statement Return level order traversal as list of lists (each level separate).



## Intuition

- A queue enforces the FIFO order naturally: nodes are processed in the same sequence they are inserted.
- When a node is processed, its children are added to the back of the queue
- making sure that they are visited only after all nodes of the current level.

## Solution

- Check Code!

Time:  $O(n)$  | Space:  $O(w)$  where  $w = \max$  width

# Problem 6: Diameter of Binary Tree

Problem Statement Find the length of the longest path between any two nodes.

Intuition

- Diameter at node = left height + right height
- Track global maximum while computing heights
- Path may or may not pass through root!!

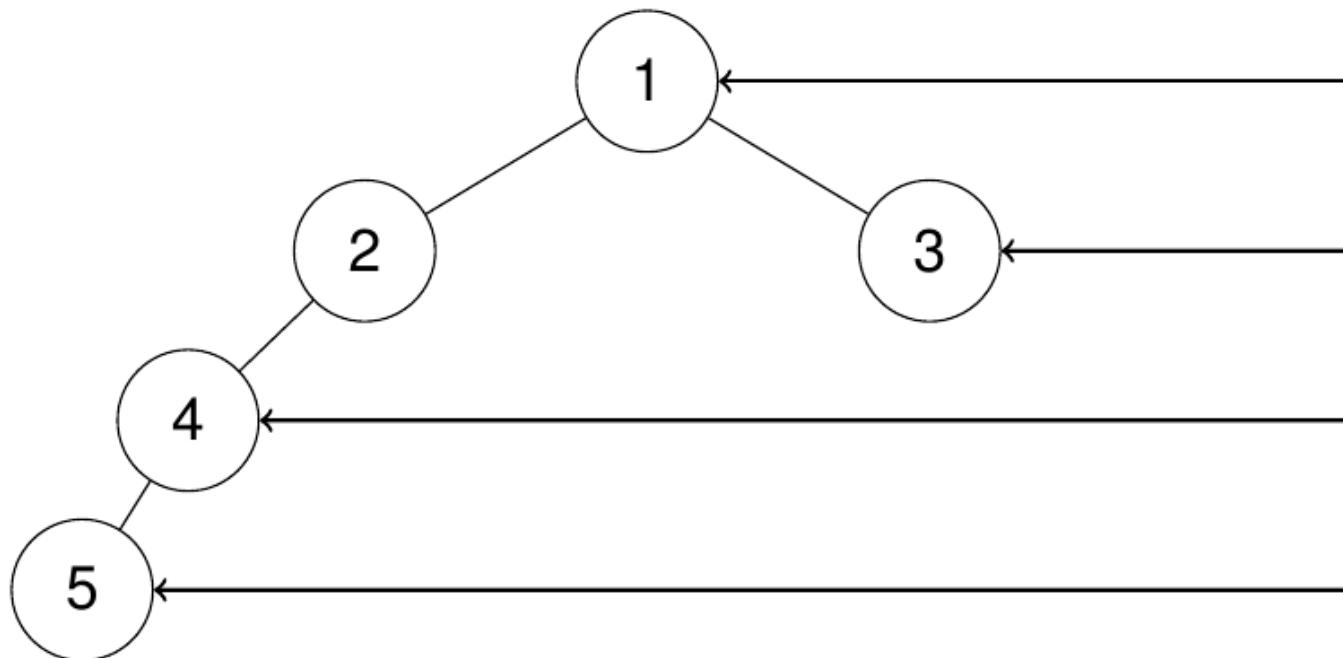
## Solution

```
int diameter = 0;  
  
int height(Node* root) {  
    if (root == NULL) return 0;  
    int left = height(root->left);  
    int right = height(root->right);  
    diameter = max(diameter, left + right);  
    return 1 + max(left, right);  
}
```

Time:  $O(n)$  | Space:  $O(h)$

## Problem 7: Right Side View

Problem Statement Return nodes visible from right side of tree.



## Intuition

- Level order traversal, take last node of each level
- Or: DFS with “current level” tracking
- In the DFS, this is NOT Inorder, but actually Postorder, for left view -> Inorder.

## Solution

```
void rightSideView(Node* root, int level, int* maxLevel) {  
    if (root == NULL) return;  
    if (level > *maxLevel) {  
        printf("%d ", root->data);  
        *maxLevel = level;  
    }  
    rightSideView(root->right, level + 1, maxLevel);  
    rightSideView(root->left, level + 1, maxLevel);  
}
```

Time:  $O(n)$  | Space:  $O(h)$

# Summary

---

# Summary: Tree Operations

Operation	Time	Space
Traversals	$O(n)$	$O(h)$
Height/Depth	$O(n)$	$O(h)$
Diameter	$O(n)$	$O(h)$
LCA	$O(n)$	$O(h)$
Level Order	$O(n)$	$O(w)$

Where  $h$  = height,  $w$  = max width of tree

# Takeaway Problems!

## Balanced Binary Tree

Check if binary tree is height-balanced (for every node, left and right subtrees differ by at most 1).

## Path Sum

Check if root-to-leaf path exists with given sum.

## Symmetric Tree

Check if tree is a mirror of itself.

# When to Use Trees?

Use Trees when:

- Need hierarchical data representation
- Implementing file systems
- Expression evaluation
- Decision making processes
- Organizational structures
- Network routing (spanning trees)

Remember

- Recursion is ALL you need for the most part.
- Traversal choice depends on problem requirements
- Always handle NULL/base case first
- Most tree problems can be solved with DFS or BFS

Questions?

Thank you!