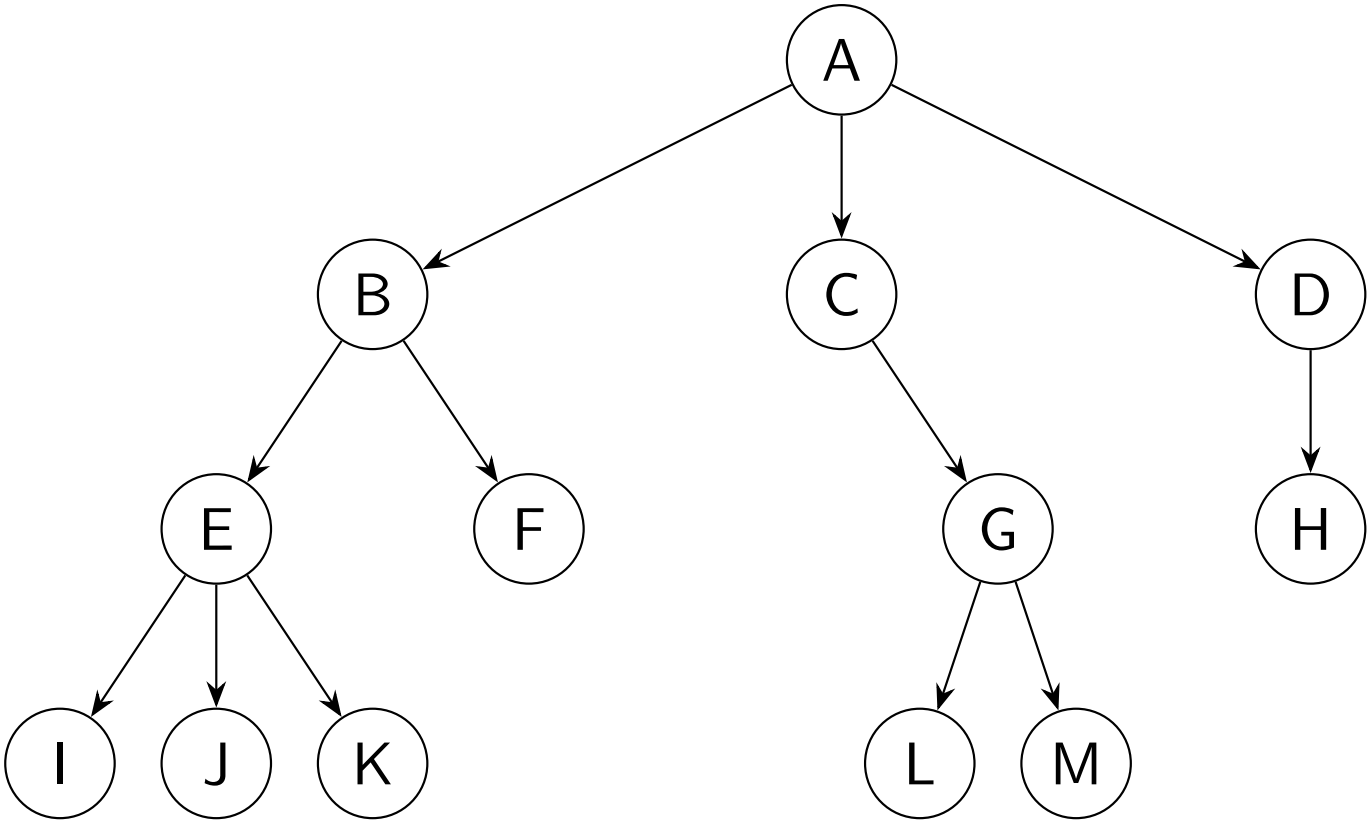
Data Structures and Algorithms

Tree in DSA

Department of IT
Course Instructor:- Dr Arathi Sankar P

October 22, 2025

Trees: A Fundamental Data Structure



Key Concepts

- **Non-linear Data Structure:** Unlike arrays or linked lists, trees represent hierarchical relationships.
- **Root Node:** The topmost node in the tree (e.g., node A in the diagram).
- **Child/Parent:** A node directly connected to another node one level below it is a child; the node above is its parent.
- **Leaf Node:** A node with no children (e.g., I, J, K, L, M, H).
- **Path:** A sequence of consecutive edges from a source node to a destination node.
- **Ancestor:** Any node on the path from the root to the current node. (e.g., the ancestors of node E are B and A).
- **Descendant:** Any node that is in the subtree rooted at the current node. (e.g., the descendants of node C are G, L, and M).

Key Concepts

- **Sibling:** Nodes that share the same Parent. (e.g., nodes B, C, and D are siblings).
- **Degree:** The total number of subtrees attached to that node. Alternatively, it is the number of its children. (e.g., The degree of node A is 3, and the degree of node E is 3). Degree of Tree=max(Degree of all Nodes in the Tree).
- **Edges:** Links connecting two nodes.
- **Subtree:** A tree consisting of a node and all its descendants.
- **Depth:** The length of the path from the root to the node. The root node has a depth of 0.
- **Height:** The length of the path from the node to the deepest leaf in its subtree. The height of the tree is the height of its root node.

n nodes = n-1 edges

Applications of Trees in DSA

- **Hierarchical Data Representation**
  - File systems (folders & subfolders)
  - Organizational structures (CEO $\rightarrow$ Managers $\rightarrow$ Employees)
- **Expression Parsing & Evaluation**
  - Expression trees for arithmetic expressions
  - Used in compilers
- **Searching & Sorting**
  - Binary Search Tree (BST) for efficient search, insertion, deletion
  - Balanced BSTs (AVL, Red-Black) for optimized performance
- **Networking & Routing**
  - Trie trees for IP routing & autocomplete
  - Decision trees for network packet classification
- **Database Indexing**
  - B-Trees / B+ Trees used for indexing
  - Fast retrieval of records
- **Game Development & AI**
  - Game trees for decision-making (Chess, Tic-Tac-Toe)
  - Minimax algorithm

What is a Binary Tree?

- **Definition:** Each node in a binary tree can have **at most 2 children**.
- These children are typically referred to as the **left child** and the **right child**.

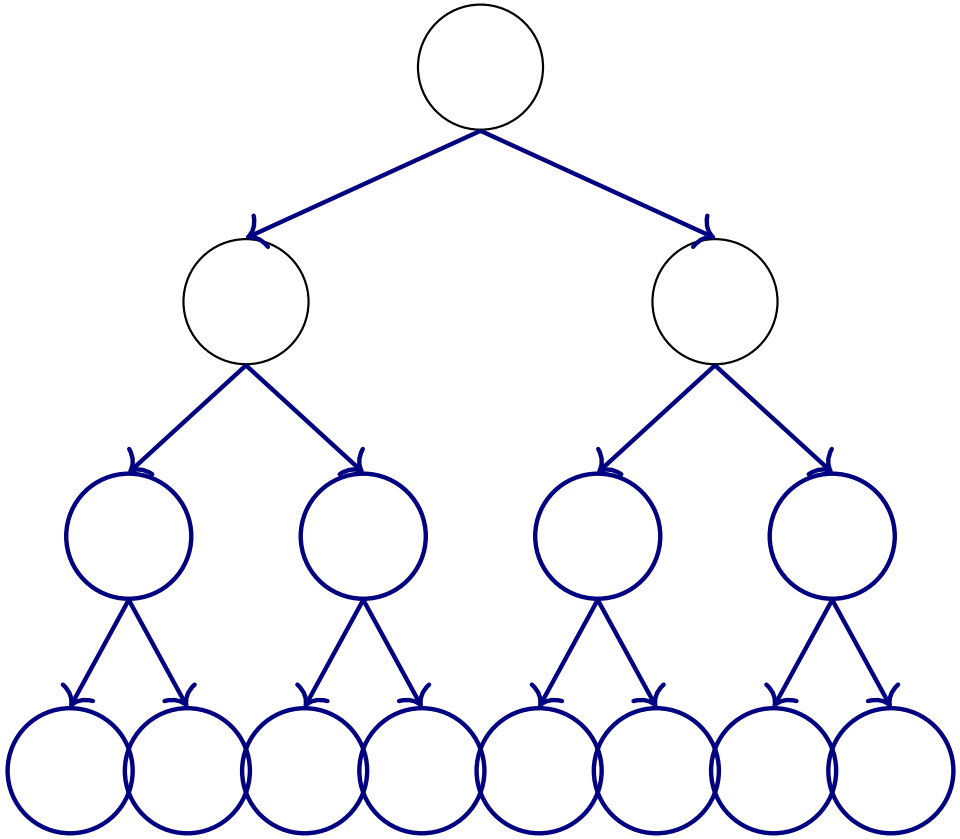
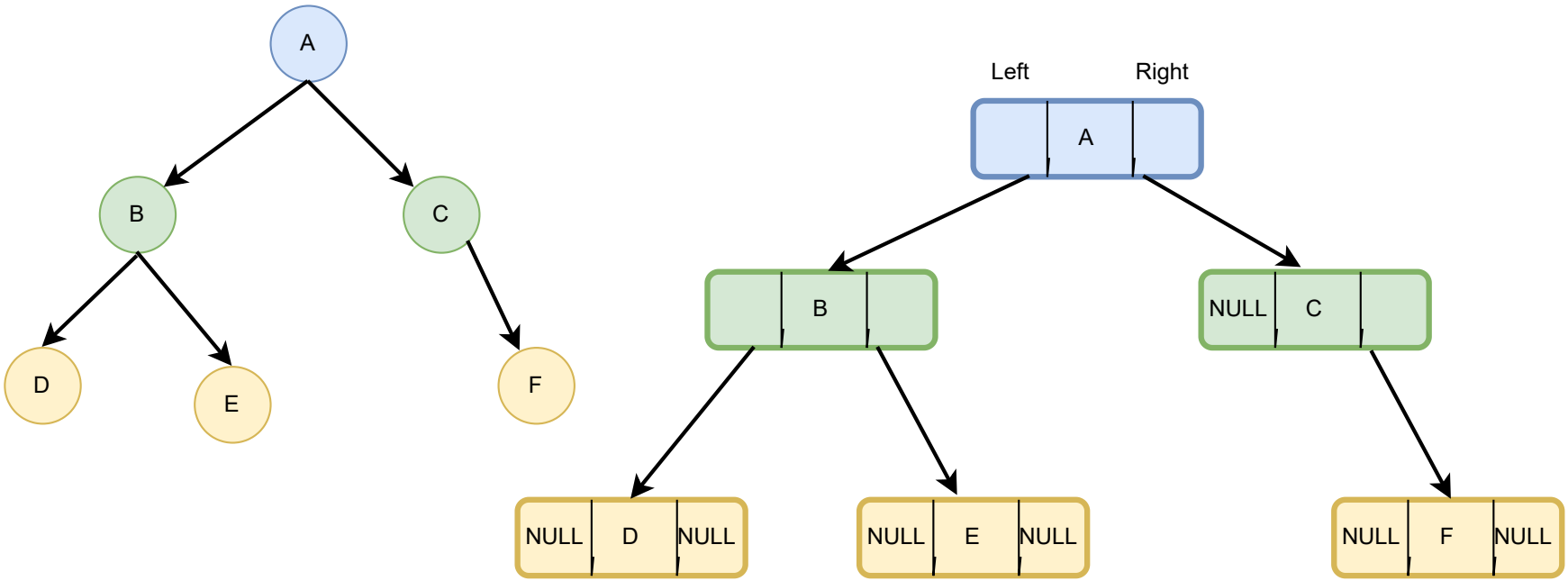


Figure: A sample Binary Tree structure.

Binary Tree - Logical and Memory Representation



Maximum Nodes at a Level

- Levels in a tree are typically indexed starting from $i = 0$ (for the root).
- The **maximum number of nodes possible** at any level i is:

$$\text{Max Nodes}(i) = 2^i$$

Example (from diagram)

- Level $i = 0$: $2^0 = 1$ node (The root)
- Level $i = 1$: $2^1 = 2$ nodes
- Level $i = 2$: $2^2 = 4$ nodes
- Level $i = 3$: $2^3 = 8$ nodes
- Level $i = 4$: $2^4 = 16$ nodes

Properties of Height and Total Nodes

- Let h be the **height** of the binary tree (often measured as the maximum level index, starting from 0).
- The **maximum number of nodes** (N) in a binary tree of height h is:

$$N = 2^0 + 2^1 + 2^2 + \cdots + 2^h$$

$$N_{\text{max}} = 2^{h+1} - 1$$

- This formula comes from the sum of a Geometric Progression.

Relationship between N and h

If N is the total number of nodes in a **complete** binary tree:

- $N + 1 = 2^{h+1}$
- $\log_2(N + 1) = h + 1$
- **Minimum Height:** $h_{\text{min}} = \lceil \log_2(N + 1) \rceil - 1$

Maximum and Minimum Height

- Let N be the total number of nodes in the tree.
- **Maximum Height** (h_{max}):
  - Occurs in a **skewed tree** (where each node has only one child).
  - $h_{\text{max}} = N - 1$
- **Minimum Height** (h_{min}):
  - Occurs in a **complete binary tree** (where all levels are full).
  - $h_{\text{min}} = \lceil \log_2(N + 1) \rceil - 1$

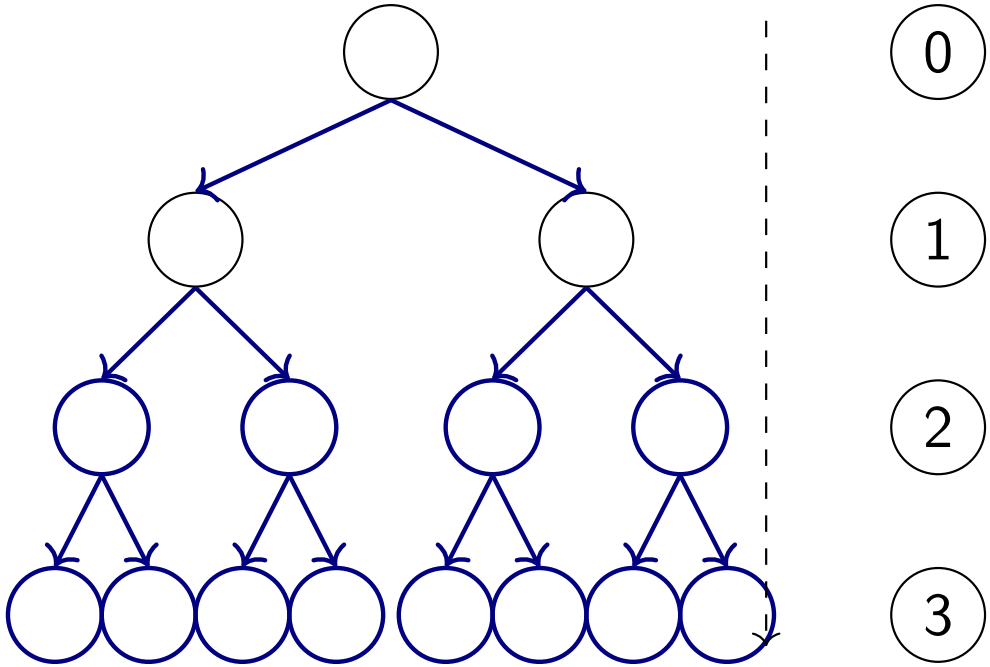
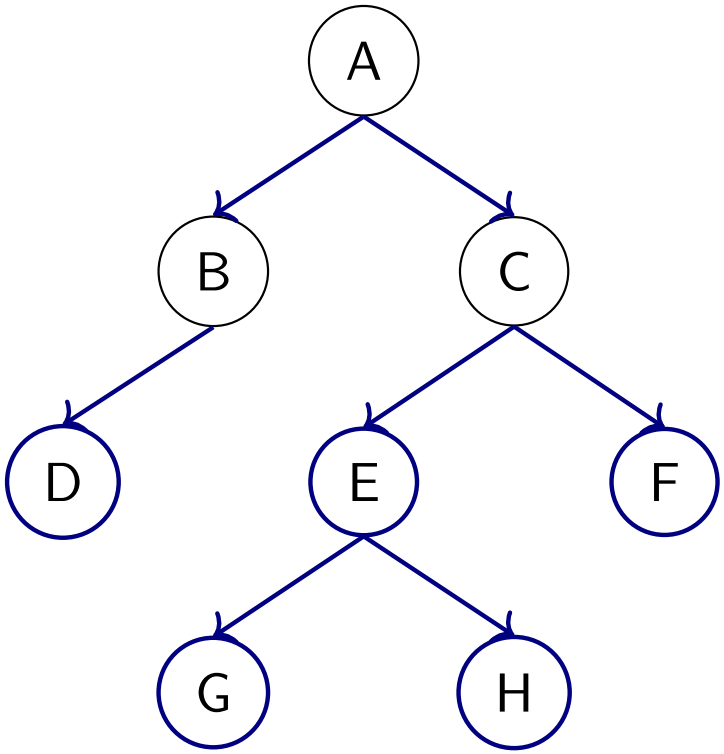


Figure: Example of a Perfect Binary Tree (Height = 3)

Binary Trees

A **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the **left child** and the **right child**.

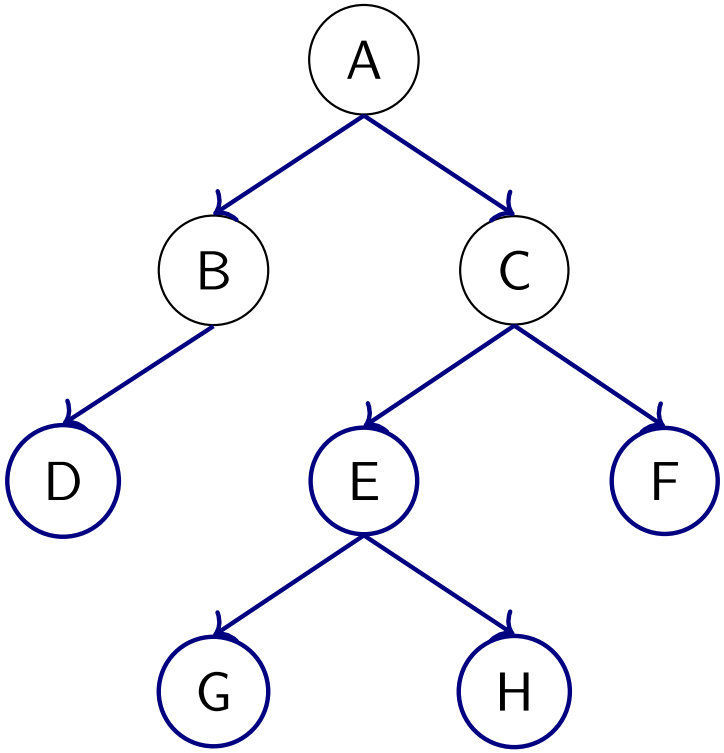
Binary Tree - Example



Each node has either **0, 1 or 2 Children**

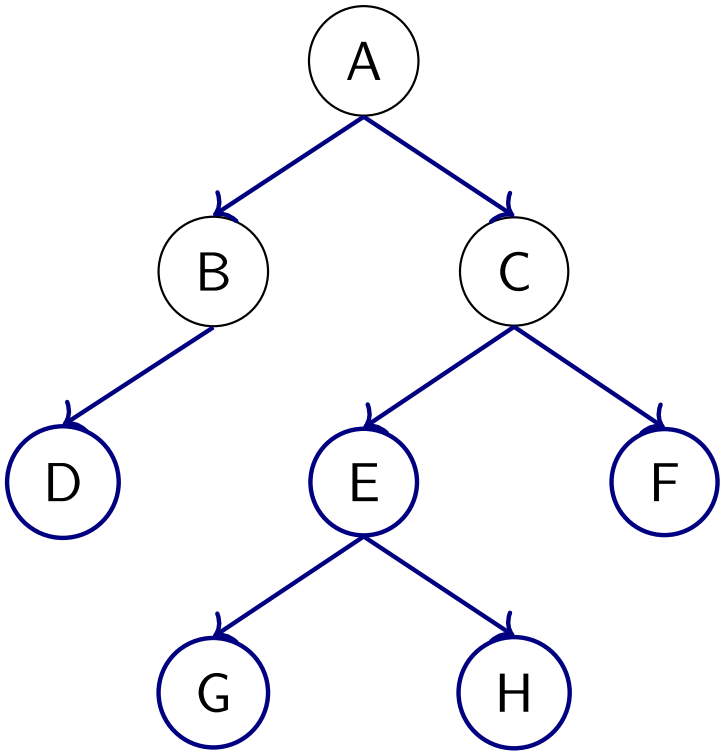
Size of a Binary Tree

Size = 8



The total number of nodes in a binary tree is known as the **size of a binary tree**.

External Nodes & Internal Nodes



External node is a node with no children (a leaf node)

Internal node is a node with at least one child (a non-leaf node)

Types of Binary Trees

- A node can have **at most 2 children**.

Main Classifications:

1. **Full/Proper/Strict Binary Tree**
2. **Complete Binary Tree**
3. **Perfect Binary Tree**
4. **Degenerate Binary Tree**

Definition (Perfect Binary Tree)

- All **internal nodes** have **2 children**.
- All **leaf nodes** are at the **same level**.

What is a Binary Tree?

- A fundamental data structure where each node has at most two children (left and right).
- We classify trees based on the structural constraints applied.

Notation Used

- h: **Height** of the tree (a single-node tree has $h = 0$).
- N: **Total Number of Nodes** in the tree.

1. Full Binary Tree (Proper/Strict)

Definition

A tree is Full if every node has either zero or two children. No node has only one child.

Minimum Nodes (N_{min}) for height h:

- Occurs when the tree is 'skewed' but still full (e.g., a spine of nodes where each has two children, but all branches are short).
- $$N_{min} = 2h + 1$$

Maximum Nodes (N_{max}) for height h:

- Achieved when the tree is also Perfect.
- $$N_{max} = 2^{h+1} - 1$$

2. Complete Binary Tree

Definition

- Every level, except possibly the last, is completely filled.
- All nodes at the last level are as far left as possible.

Application: This structure enables efficient array representation (e.g., Binary Heaps) due to compact, left-packed storage.

- **Maximum Nodes (N_{max}):** Achieved when the tree is Perfect.

$$N_{max} = 2^{h+1} - 1$$

- **Minimum Nodes (N_{min}):** Occurs when all levels up to $h-1$ are full, and there is only one node at level h.

$$N_{min} = 2^{h}$$

3. Perfect Binary Tree

Definition

- All internal nodes have exactly two children.
- All leaf nodes are at the same level (depth).

Key Relationship: A Perfect Binary Tree is always both Full and Complete. It is the most symmetric and balanced structure.

Total Nodes (N):

$$N = 2^{h+1} - 1$$

Height (h):

$$h = \log_2(N + 1) - 1$$

4. Degenerate Binary Tree (Skewed)

Definition

A tree where every internal node has only one child.

Consequence: It effectively devolves into a simple linked list.

- This structure leads to the worst-case $O(N)$ time complexity for search operations.

Node Constraints: The relationship is linear.

- Total Nodes (N): $N = h + 1$
- Height (h): $h = N - 1$

Summary of Node Limits (Based on Height h)

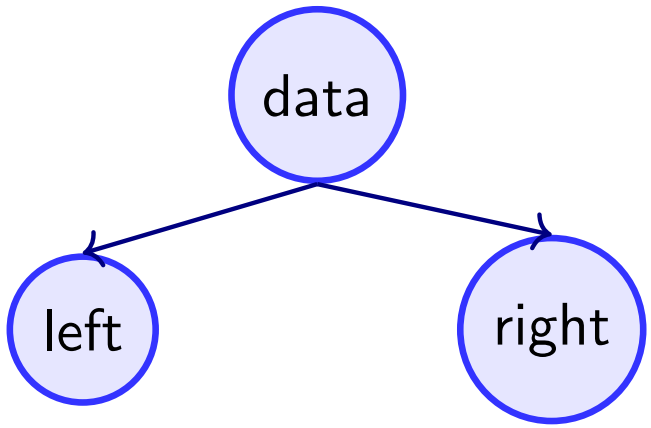
Table: Binary Tree Node Counts by Height

Tree Type	Minimum Nodes (N_{min})	Maximum Nodes (N_{max})
Degenerate (Skewed)	$h+1$	$h+1$
Full (Strict)	$2h+1$	$2^{h+1}-1$
Complete	2^h	$2^{h+1}-1$
Perfect	$2^{h+1}-1$	$2^{h+1}-1$

The Core Binary Node Structure

1Binary tree implementation.

- A hierarchical data structure composed of nodes.
- Each node has at most two children: a left child and a right child.
- The top-most node is called the root.



2. C Structure Definition (from the code)

Listing: The Node Structure

```
struct node {
    int data;
    struct node *left;
    struct node *right;
};
```

Recursive Tree Construction ('Create()')

C Code Snippet

Listing: Recursive Create Function

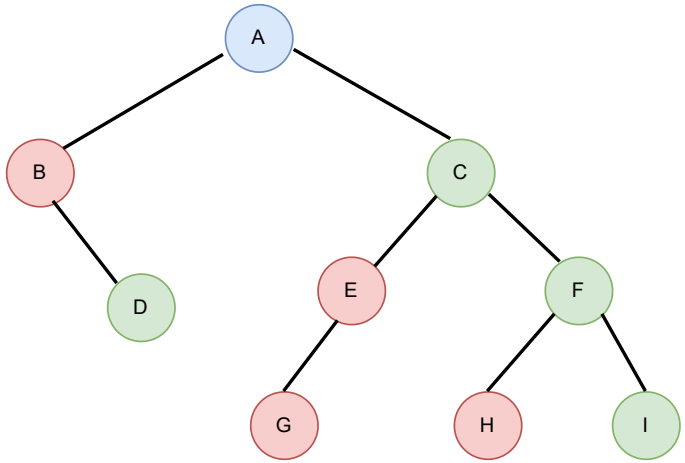
```
struct Node *Create() {
    int x;
    printf( Enter data (-1 for null)
        :  );
    scanf( %d , &x);

    if (x == -1) {return NULL};

    newnode = (struct Node*)malloc(
        sizeof(newnode));
    newnode->data = x;
    printf(  left child of %d:\n , x
        );
    newnode->left = Create();
    printf( Enter right child of %d
        :\n , x);
    newnode->right = Create();
    return newnode;
```

How Recursion Works

1. Read data for the current node.
2. If data is -1, return NULL (base case).
3. Allocate memory for the new node.
4. Recursively call 'Create()' for the left child.
5. Recursively call 'Create()' for the right child.
6. The process unwinds, linking all children back up to the root.

Binary Tree Traversal Orders

Binary Tree Structure:



Traversal Results

Inorder: B D A G E C H F I

Preorder: A B D C E G F H I

Postorder: D B G E H I F C A

Traversal Rules:

- **Inorder:** Left → Root → Right
- **Preorder:** Root → Left → Right
- **Postorder:** Left → Right → Root

Essential Operation - Inorder Traversal

Traversal is the process of visiting every node in the tree exactly once. The order of visiting the Left subtree (L), the Root (R), and the Right subtree (R) defines the traversal type.

Three Main Traversal Types

- Preorder: R $\rightarrow$ L $\rightarrow$ R (Used for copying trees)
- Inorder: L $\rightarrow$ R $\rightarrow$ R (Prints sorted data for BSTs)
- Postorder: L $\rightarrow$ R $\rightarrow$ R (Used for deleting trees)

Inorder C Implementation

Listing: Inorder Traversal Code

```
void Inorder(struct node *root) {
    if (root == NULL) {
        return;
    }
    // 1. Traverse Left Subtree
    Inorder(root->left);

    // 2. Visit Root
    printf( %d  , root->data);

    // 3. Traverse Right Subtree
    Inorder(root->right);
}
```

Preorder Traversal

Root → Left → Right

Listing: Preorder Traversal (Recursive)

```
void Preorder (struct node *root) {
    // 1. Check for the base case (empty subtree)
    if (root == NULL) {
        return;
    }

    printf( %d , root->data);
    Preorder(root->left);

    Preorder(root->right);
}
```

```
void main () {
    struct node *root;
    printf( Pre-order is:  );
    Preorder(root); // Initiate traversal
}
```

Postorder Traversal

Left → Right → Root

Listing: Postorder Traversal (Recursive)

```
void Postorder (struct node *root) {
    // 1. Check for the base case (empty subtree)
    if (root == NULL) {
        return;
    }

    Postorder(root->left); // 1. Recurse Left

    Postorder(root->right); // 2. Recurse Right

    printf( /%d , root->data); // 3. Visit Root (Print)
}
```

```
void main () {
    struct node *root;
    printf( Post-order is:  );
    Postorder(root); // Initiate traversal
}
```

Understanding the Inputs

- We are given two traversal sequences for a unique binary tree:
  - **Post-order Traversal:** `9, 1, 2, 12, 7, 5, 3, 11, 4, 8`
  - **In-order Traversal:** `9, 5, 1, 7, 2, 12, 8, 4, 3, 11`
- **Key Properties:**
  - **Post-order (Left, Right, Root):** The Root is always the *last* element.
  - **In-order (Left, Root, Right):** The Root divides the sequence into its Left and Right subtrees.

Construction Procedure

Step-by-step Guide

1. **Identify the Root (from Post-order):**
   - The **last** element in the `Post-order` sequence is always the **Root** of the current subtree.
   - **Example:** The main Root is **8**.
2. **Locate Root in In-order Partition:**
   - Find the identified Root in the `In-order` sequence.
   - This position splits the `In-order` into its **Left Subtree** nodes (elements to the left) and **Right Subtree** nodes (elements to the right).
   - **Example:** `In-order:` `(9, 5, 1, 7, 2, 12)`, `8`, `(4, 3, 11)`
     - Left Subtree nodes: $9, 5, 1, 7, 2, 12$
     - Right Subtree nodes: $4, 3, 11$

Construction Procedure (Cont.)

Sub-sequence Generation & Recursion

- **Partition Post-order Sub-sequences:**
  - Exclude the current Root from the `Post-order` sequence.
  - Based on the *counts* of Left and Right Subtree nodes (from In-order), split the remaining `Post-order` into two parts:
    - The first part forms the `Post-order` for the **Left Subtree**.
    - The second part forms the `Post-order` for the **Right Subtree**.
  - **Example:** `Post-order` (excluding 8): (9, 1, 2, 12, 7, 5), (3, 11, 4)
    - Left Subtree Post-order (6 nodes): $9, 1, 2, 12, 7, 5$
    - Right Subtree Post-order (3 nodes): $3, 11, 4$
- **Recursive Construction:**
  - Recursively apply Steps 1, 2, and 3 to the **Left Subtree's sequences** and the **Right Subtree's sequences**.
  - Continue until all nodes are placed and sub-sequences are empty.
  - For each recursive call, the last element of its respective Post-order sub-sequence becomes the new root.

Final Binary Tree

Visual Representation