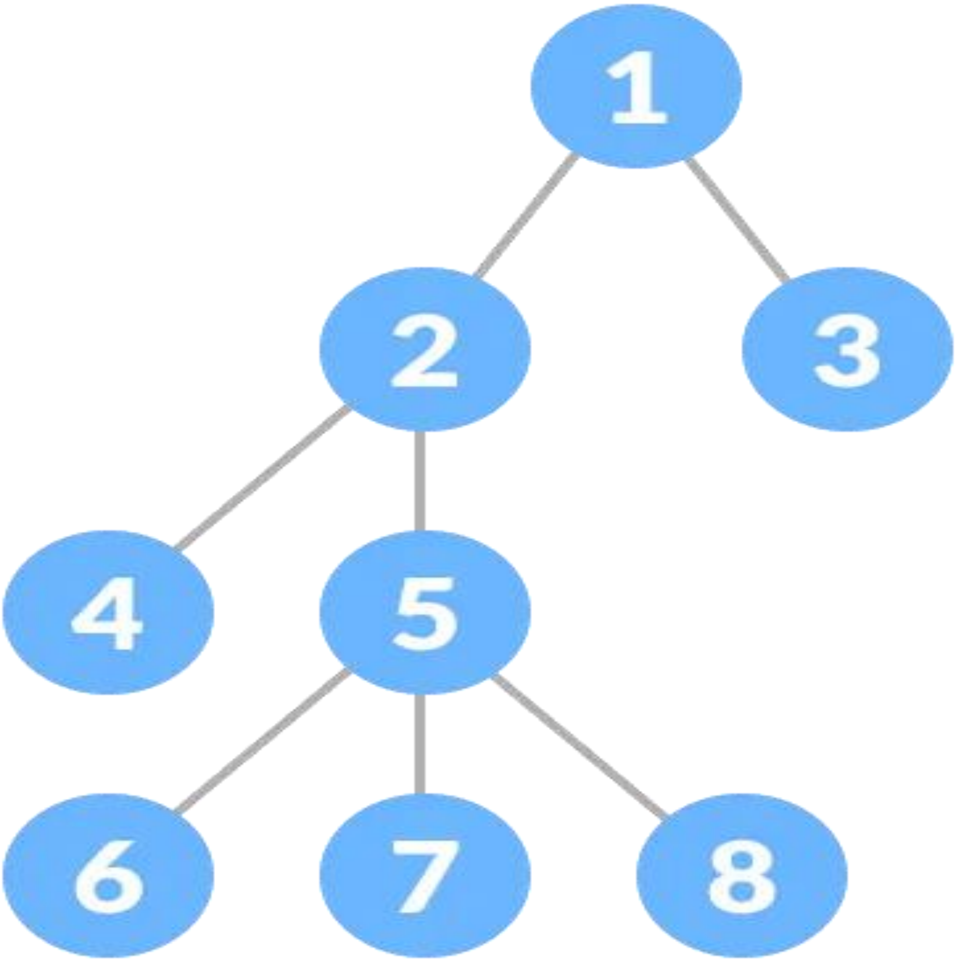
Tree Data Structures

Table of Contents

INTRODUCTION TO TREES

A **Tree** is a hierarchical data structure that consists of nodes connected by edges. It's a non-linear data structure (data is organized at multiple levels) that simulates a tree structure with a root value and subtrees of children, represented as a set of linked nodes.



Key Characteristics:

- **Hierarchical Structure**: Data is organized in a hierarchy
- **Non-linear**: Unlike arrays or linked lists, trees don't store data in a sequential manner
- **Recursive Nature**: Each subtree is also a tree
- **Connected Graph**: All nodes are connected, but there are no cycles

Real-world Examples:

- **File System**: Folders and subfolders
- **Organization Chart**: Company hierarchy
- **Decision Trees**: Decision-making processes
- **HTML DOM**: Web page structure
- **Family Tree**: Genealogical relationships

BASIC TREE TERMINOLOGY

Essential Terms:

Node: Basic unit of a tree containing data and references to child nodes

```
[A] ← Node
```

Root: The topmost node of the tree (has no parent)

```
  [A] ← Root
  / | \
 B  C  D
```

Parent: A node that has one or more child nodes **Child**: A node that has a parent node **Siblings**: Nodes that share the same parent

Leaf/Terminal Node: A node with no children

```
  [A]
  / | \
 B  C  D ← All are leaf nodes
```

Internal Node: A node with at least one child

Edge: Connection between two nodes **Path**: Sequence of nodes connected by edges **Height**: Number of edges on the longest path from root to leaf **Depth/Level**: Number of edges from root to a particular node **Degree**: Number of children of a node

Child: A node that is directly connected to a parent node and is a descendant of that parent.

Siblings: Nodes that share the same parent.

Leaf Node (External Node/Terminal Node): A node that has no children. It is at the lowest level of a particular branch.

Internal Node: A node that has at least one child.

Subtree: A portion of a tree that is itself a tree, rooted at one of the original tree's nodes.

Degree of a Node: The number of children a node has.

Degree of a Tree: The maximum degree among all nodes in the tree.

Path: A sequence of nodes and edges from one node to another.

Ancestor: Any node on the path from the root to a given node (excluding the node itself).

Descendant: Any node reachable by following paths from a given node towards its children.

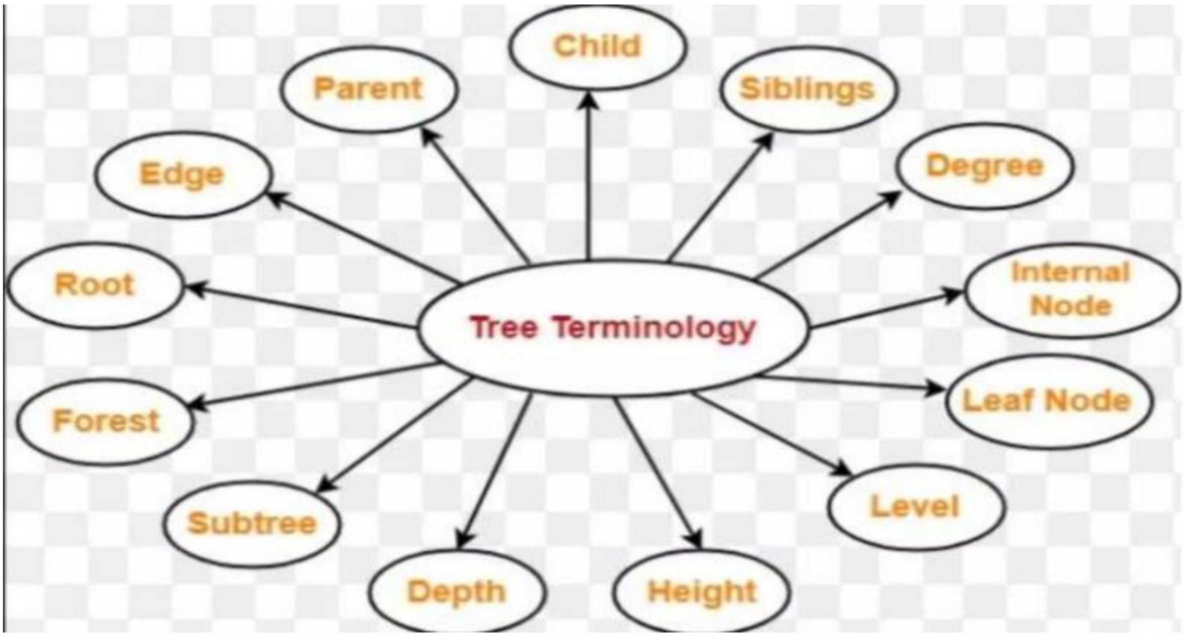
Level: The depth of a node, often starting with the root at level 0 or 1. If a node is at level L, its children are at level L+1.

Depth of a Node: The number of edges on the path from the root to that node.

Height of a Node: The number of edges on the longest path from the node to a leaf descendant.

Height of a Tree: The height of the root node, which is the maximum depth of any node in the tree.

Forest: A collection of disjoint trees

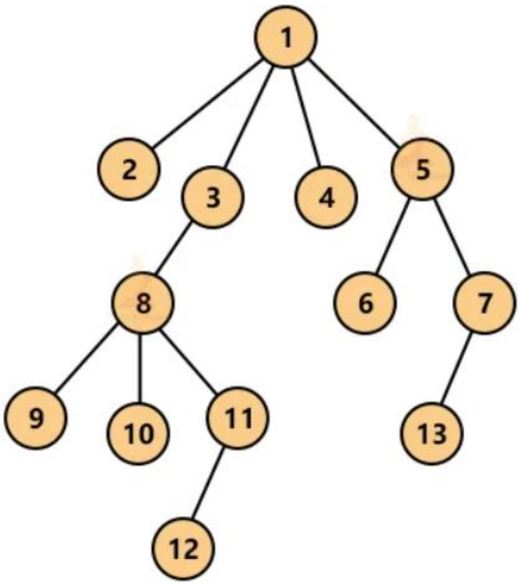


TYPES OF TREES

1. General Tree

- A **General Tree** is a type of tree where each node can have **any number of children**, not just two like in a binary tree.
- A common real-life example of this is a **file system** on your computer — where folders can contain many files and subfolders.
- While it's a flexible structure, **general trees aren't used very often** in programming because they can be **tricky to implement and manage** compared to more specific tree types.

Diagram:



2. Binary Tree

- A **Binary Tree** is a tree where **each node can have up to two children** — usually called the **left** and **right** child.

- It's one of the most common and important types of trees in computer science.

- Think of it like a family tree where each person can have **no more than two children**.

- **Binary trees form the base** for many advanced data structures, like Binary Search Trees and Heaps.

Properties:

- Maximum nodes at level $i = 2^i$
- Maximum nodes in a binary tree of height h = 2^(h+1) - 1

3. Full Binary Tree

- In a **Full Binary Tree**, **every node** either has **exactly two children** or **no children at all**.
- That means you'll never find a node with **just one child** — it's either a parent of two, or it has none.
- You can think of it like a strict rule: if a node wants to have children, it must have **both** left and right.

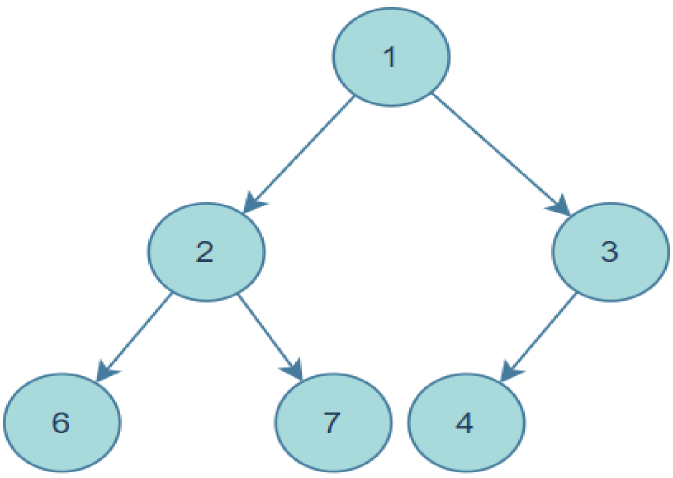
Example:

```
      1
     / \
    2   3
```

4. Complete Binary Tree

- A **Complete Binary Tree** is neatly organized — **all levels are fully filled**, except maybe the **last level**.
- And even if the last level isn't full, the nodes are always filled **from left to right** without any gaps in between.
- You can picture it like filling seats in a theater row by row — every row is full, and the last row is being filled from **left to right**.

Example:

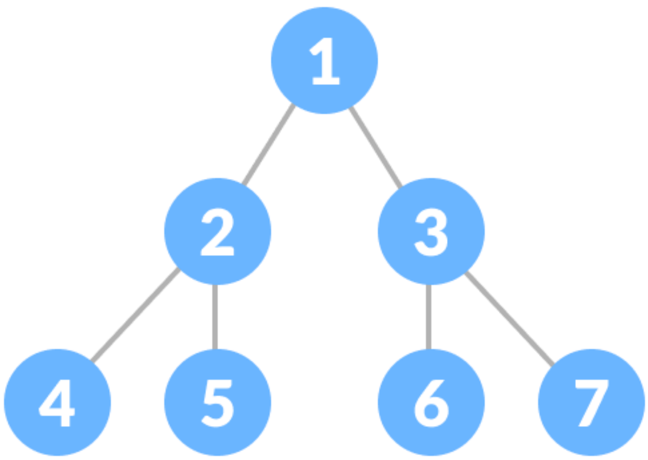


Complete Binary Tree

5. Perfect Binary Tree

- A **Perfect Binary Tree** is as balanced and symmetrical as it gets.
- **Every internal node** (non-leaf node) has **exactly two children**, and **all the leaf nodes** (the ones with no children) are at the **same level**.
- Imagine a pyramid made out of people where **each layer is full**, and everyone in the bottom row stands at the **same height** — that's how perfect it is!

Example:



6. Balanced Binary Tree

- In a **Balanced Binary Tree**, the tree stays nicely leveled — the **height difference** between the **left and right subtrees** of any node is **no more than 1**.

- This balance helps the tree stay efficient, so operations like **searching, inserting, or deleting** data are done quickly.
- You can think of it like keeping both sides of a scale even — it makes everything run smoother and faster!

7. Binary Search Tree (BST)

- A **Binary Search Tree** is a special kind of binary tree that follows a simple but powerful rule:

    1. **Left child** has values **less than** the parent node.
    2. **Right child** has values **greater than** the parent node.

- This structure keeps everything in **sorted order**, which makes finding, inserting, or deleting values much **faster — usually in O(log n) time**.
- You can think of it like a phonebook — everything is arranged so you can **quickly jump to what you're looking for** instead of going through everything one by one.

8. AVL Tree

- An **AVL Tree** is a type of **self-balancing Binary Search Tree**.

- After you **insert or delete** a node, the tree might become unbalanced — but the AVL Tree **automatically fixes itself** using special techniques called **rotations**.
- This way, it always stays balanced, which helps keep operations like **search, insert, and delete efficient**.
- Think of it like a tree that **adjusts its branches on its own** to stay neat and organized.

9. Heap Tree

- A **Heap Tree** is a **complete binary tree**, meaning it's fully filled level by level, except possibly the last, which is filled from left to right.

- It's used to build a structure called a **heap**, and comes in two main types:

  - **Min-Heap**: Every parent node is **less than or equal to** its children.
  - **Max-Heap**: Every parent node is **greater than or equal to** its children.

- Heaps are super useful in things like **Priority**

Queues, where the most important task gets handled first, and in algorithms like **Heap Sort**.

10. B-Trees and B+ Trees

- Used in **databases** and **file systems** for storing large blocks of data.
- Allow more than two children.
- B+ Trees store **all data in leaf nodes** and are optimized for **range queries**.

11. Trie (Prefix Tree)

- A special tree used to store **strings or words**.
- Each edge represents a character.
- Efficient for **searching prefixes**, **autocomplete**, and **dictionary** operations.

TREE TRAVERSAL

Process of visiting each node in a tree exactly once in a particular order.

There are two major types of tree traversal:

1. Depth-First Traversal (DFS)

In DFS, we go **as deep as possible** down one path before backing up and exploring other paths.

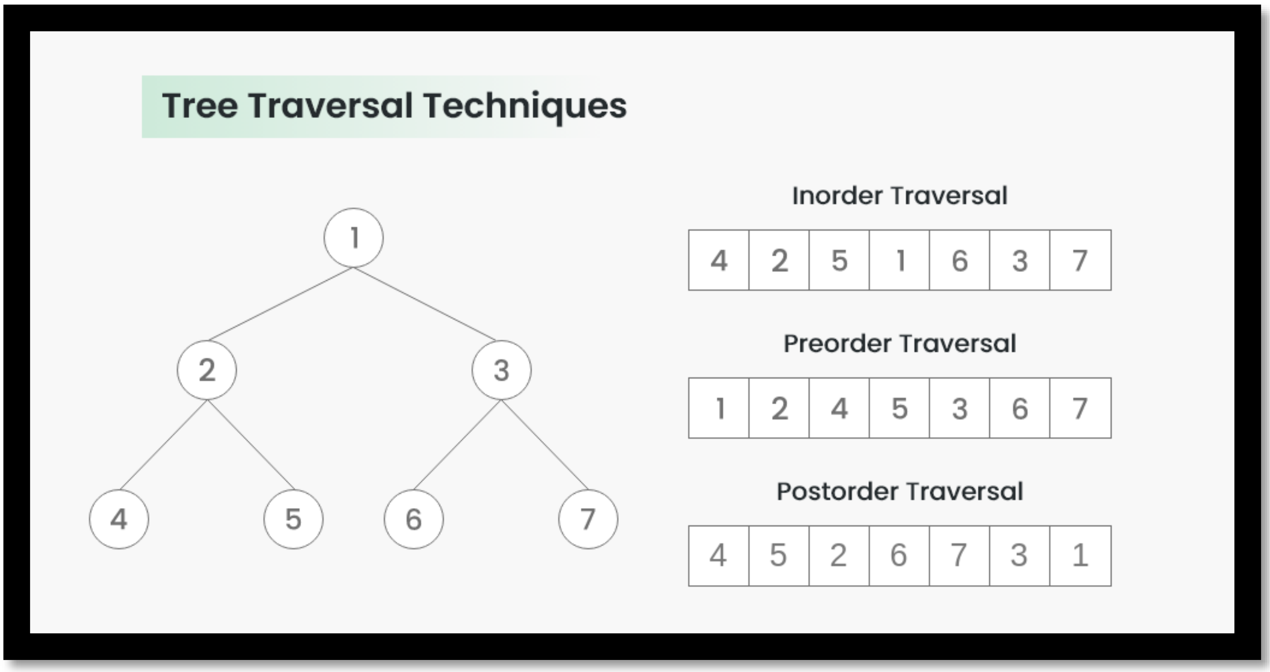
Types:

- **Inorder (Left, Root, Right)**
- **Preorder (Root, Left, Right)**
- **Postorder (Left, Right, Root)**

Example Tree:

```
    A
   / \
  B   C
 / \
D   E
```

Traversal Type	Order
Inorder	D B E A C
Preorder	A B D E C
Postorder	D E B C A



DFS CODE (C++) :

```
struct Node {
    char data;
    Node* left;
    Node* right;
};
void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

void preorder(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}
void postorder(Node* root) {
    if (root == nullptr) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}
```

2. Breadth-First Traversal (BFS)

It is also known as **Level Order Traversal**, where we visit nodes **level by level** from top to bottom and left to right.

Example Tree :

```
    A
   / \
  B   C
 / \
D   E
```

Level Order: A B C D E

BFS Code (C++ using Queue):

```cpp
#include <queue>
void levelOrder(Node* root) {
    if (root == nullptr) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* curr = q.front();
        q.pop();
        cout << curr->data << " ";
        if (curr->left) q.push(curr->left);
        if (curr->right) q.push(curr->right);
    }
}
```

5. Binary Trees

A **Binary Tree** is a non-linear data structure where each node has at most two children, commonly referred to as the **left child** and **right child**. It forms the basis of many advanced tree structures and algorithms.

Key Characteristics

- Recursive structure: each subtree is itself a binary tree.
- The maximum number of nodes at level i is 2^i.
- Total nodes in a binary tree of height h is 2^(h+1) - 1.

Structure:

```
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr), right(nullptr)
{}
};
```

---

6. Binary Search Trees (BST)

A **Binary Search Tree** maintains the property that:

- Left subtree nodes < Root node
- Right subtree nodes > Root node

This structure allows efficient **searching, insertion, and deletion** in O(log n) time on average.

BST Insert Example:

```
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->data)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}
```

---

7. AVL Trees

An **AVL Tree** is a self-balancing BST where the **balance factor** (height difference of left and right subtrees) is kept between -1 and 1.

Rotations:

To maintain balance, AVL Trees perform:

- Right Rotation (LL case)
- Left Rotation (RR case)
- Left-Right Rotation (LR case)
- Right-Left Rotation (RL case)

RR Rotation Example:

```
Node* leftRotate(Node* x) {
   Node* y = x->right;
   x->right = y->left;
   y->left = x;
   return y;
}
```

8. Heaps (Min and Max)

A **Heap** is a complete binary tree satisfying the **heap property**.

- **Min-Heap**: Each parent ≤ children
- **Max-Heap**: Each parent ≥ children

Applications:

- Priority Queues
- Heap Sort
- Task Scheduling

C++ STL Min-Heap:

```
priority_queue<int,vector<int>, greater<int>> minHeap;
```

---

9. Segment Trees

A **Segment Tree** is a binary tree used for answering range queries and updates efficiently.

Use Cases:

- Range sum queries
- Range minimum/maximum
- Frequency count in subarrays

Build Function:

```
void build(int index, int low, int high, int arr[], int
seg[]) {
    if (low == high) {
        seg[index] = arr[low];
        return;
    }
    int mid = (low + high) / 2;
    build(2*index+1, low, mid, arr, seg);
    build(2*index+2, mid+1, high, arr, seg);
    seg[index] = seg[2*index+1] + seg[2*index+2];
}
```

---

10. Trie (Prefix Tree)

A **Trie** is a tree-based structure used for storing strings efficiently, especially useful in **dictionary**, **autocomplete**, and **spell-checking** systems.

Structure & Insert:

```
struct TrieNode {
    TrieNode* children[26];
    bool isEnd = false;
};

void insert(TrieNode* root, string word) {
    TrieNode* node = root;
    for (char ch : word) {
        int idx = ch - 'a';
        if (!node->children[idx])
            node->children[idx] = new TrieNode();
        node = node->children[idx];
    }
    node->isEnd = true;
}
```

11. Advanced Tree Algorithms

Lowest Common Ancestor (LCA):

```
Node* LCA(Node* root, int a, int b) {
    if (!root || root->data == a || root->data == b)
return root;
    Node* left = LCA(root->left, a, b);
    Node* right = LCA(root->right, a, b);
    return (left && right) ? root : (left ? left : right);
}
```

Diameter of Tree:

```
int diameter(Node* root, int& res) {
    if (!root) return 0;
    int l = diameter(root->left, res);
    int r = diameter(root->right, res);
    res = max(res, l + r + 1);
    return 1 + max(l, r);
}
```

12. Sample Problems with Solutions

Problem 1: Height of a Tree

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

Problem 2: Check if a Tree is BST

```
bool isBST(Node* root, Node* min = NULL, Node* max = NULL) {
    if (!root) return true;
    if ((min && root->data <= min->data) || (max && root->data >= max->data))
        return false;
    return isBST(root->left, min, root) && isBST(root->right, root, max);
}
```

Problem 3: Count Leaf Nodes

```
int countLeaves(Node* root) {
    if (!root) return 0;
    if (!root->left && !root->right) return 1;
    return countLeaves(root->left) + countLeaves(root->right);
}
```
