Data Structures & Algorithms (C++)

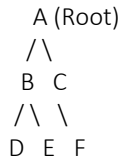
Introduction to Trees

A **tree** is a non-linear hierarchical data structure used to represent relationships between elements. Unlike arrays or linked lists, trees organize data in a parent–child structure.

Trees are widely used in **databases, file systems, compilers, AI, and networking algorithms**. They help in fast searching, sorting, and hierarchical representation.

A tree starts with a **root node** and branches into multiple child nodes, forming a structure similar to a real-world tree.

Tree Structure Example



1. Tree Definitions and Properties

Introduction

A tree is defined as a collection of nodes connected by edges with a hierarchical relationship. It has no cycles and always has a single root node.

Each node can have zero or more child nodes, and every node except the root has exactly one parent.

Trees are recursive in nature, meaning each subtree is also a tree.

Important Properties of Trees

- A tree with **N nodes has N-1 edges**
- There is exactly **one path between any two nodes**
- No cycles exist in a tree
- A single root node exists

Types of Nodes

Node Type	Description
Root	Top-most node
Leaf	Node with no children
Internal Node	Node with at least one child
Parent	Node with children
Child	Descendant of a node

2. Tree Functions

Introduction

Tree functions are operations used to manipulate and analyze tree structures. These include insertion, deletion, searching, and traversal operations.

In C++, tree functions are usually implemented using recursion due to the hierarchical nature of trees.

These functions form the foundation of all advanced tree algorithms like BST, AVL, and heaps.

Common Tree Functions

- Insert node
- Delete node
- Search node
- Traverse tree
- Compute height

Example Structure in C++

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

Key Idea

Each node contains:

- Data

- Pointer to left child
- Pointer to right child

3. C++ Tree Interface

Introduction

A tree interface defines how tree operations are structured in C++. It provides function declarations for insertion, traversal, and other operations.

Interfaces help in separating **implementation from definition**, improving modularity and code organization.

In real-world systems, tree interfaces are used in libraries and frameworks for reusable code.

Basic Interface Example

```
class Tree {
public:
   virtual void insert(int value) = 0;
   virtual void inorder() = 0;
};
```

Explanation

- `class Tree` defines abstract structure
- `virtual` functions enforce implementation in derived classes
- `= 0` makes it a pure virtual function (abstract)

Key Features

- Supports abstraction
- Enables polymorphism
- Encourages reusable design
- Used in OOP-based DSA design

4. Linked Structure for General Trees

Introduction

A linked structure represents a tree using dynamically allocated nodes connected via pointers.

Instead of fixed arrays, each node stores pointers to its children, making it flexible and memory efficient.

This structure is widely used in real-world applications like XML parsing and file systems.

Node Structure

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

Explanation

- Each node stores data
- left pointer refers to left child
- right pointer refers to right child
- Dynamic memory allows flexible tree growth

Advantages

- Dynamic size
- Efficient memory usage
- Easy insertion/deletion
- No fixed limit

Disadvantage

- Complex pointer management
- Requires careful memory handling

5. Tree Traversal Algorithms

Introduction

Tree traversal refers to the process of visiting all nodes in a tree in a specific order.

Traversal is essential for searching, printing, and processing tree data.

There are two main types:

- Depth First Search (DFS)
- Breadth First Search (BFS)

Types of Traversal

Type	Order
Preorder	Root → Left → Right
Inorder	Left → Root → Right
Postorder	Left → Right → Root

Example Tree

6. Depth and Height

Introduction

Depth and height are important measurements in tree structures that define node positioning.

- **Depth** = distance from root to a node
- **Height** = longest path from node to leaf

These concepts are critical in balancing trees and analyzing performance.

Definitions

- Root depth = 0
- Leaf height = 0
- Tree height = height of root node

Example

Depth of D = 2
Height of B = 1
Height of tree = 2

Importance

- Used in AVL trees
- Used in BST balancing
- Helps analyze algorithm efficiency

7. Preorder Traversal

Introduction

Preorder traversal visits nodes in the order:
Root → Left → Right

It is useful for copying trees and generating prefix expressions.

Preorder is implemented using recursion in C++.

Algorithm

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

C++ Code

```
void preorder(Node* root) {
   if (root == NULL)
      return;

   cout << root->data << " ";
   preorder(root->left);
   preorder(root->right);
}
```

Example Output

For tree:

```
    A
   / \
  B   C
```

Output:

```
A B C
```

8. Postorder Traversal

Introduction

Postorder traversal visits nodes in the order:
Left → Right → Root

It is widely used in deleting trees and evaluating expressions.

This traversal ensures children are processed before the parent node.

Algorithm

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

C++ Code

```
void postorder(Node* root) {
   if (root == NULL)
      return;

   postorder(root->left);
   postorder(root->right);
   cout << root->data << " ";
}
```

Example Output

For tree:

```
    A
   / \
  B   C
```

Output:

B C A

Comparison Table (Traversals)

Traversal	Order	Use Case

Preorder	Root → Left → Right	Copying tree
Postorder	Left → Right → Root	Deleting tree
Inorder	Left → Root → Right	BST sorting

C++ Implementation

```
#include <iostream>
using namespace std;
/*
=====================================================
TREE NODE STRUCTURE (Linked Representation)
=====================================================
Each node has:
- data
- left child pointer
- right child pointer
=====================================================
*/
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) {
        data = value;
        left = right = NULL;
    }
};
/*
=====================================================
TREE CLASS (Interface + Implementation)
=====================================================
Includes:
- Insert (simple level-based insertion)
- Preorder traversal
- Postorder traversal
- Height calculation
- Depth calculation helper
=====================================================
*/
class Tree {
private:
    Node* root;
    // Helper: Insert node (level-order style)
    Node* insert(Node* node, int value) {
        if (node == NULL)
```

```
        return new Node(value);

    if (value < node->data)
        node->left = insert(node->left, value);
    else
        node->right = insert(node->right, value);
    return node;
  }
  // Preorder: Root -> Left -> Right
  void preorder(Node* node) {
    if (node == NULL)
        return;
    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
  }
  // Postorder: Left -> Right -> Root
  void postorder(Node* node) {
    if (node == NULL)
        return;
    postorder(node->left);
    postorder(node->right);
    cout << node->data << " ";
  }
  // Height of tree
  int height(Node* node) {
    if (node == NULL)
        return -1;
    int leftHeight = height(node->left);
    int rightHeight = height(node->right);

    return max(leftHeight, rightHeight) + 1;
  }
  // Depth of a node (search-based)
  int depth(Node* node, int value, int currentDepth) {
    if (node == NULL)
        return -1;
    if (node->data == value)
        return currentDepth;

    int left = depth(node->left, value, currentDepth + 1);
    if (left != -1)
        return left;
    return depth(node->right, value, currentDepth + 1);
  }
public:
  Tree() {
    root = NULL;
```

```
    }
    void insert(int value) {
        root = insert(root, value);
    }
    void showPreorder() {
        cout << "Preorder Traversal: ";
        preorder(root);
        cout << endl;
    }
    void showPostorder() {
        cout << "Postorder Traversal: ";
        postorder(root);
        cout << endl;
    }
    void showHeight() {
        cout << "Height of Tree: " << height(root) << endl;
    }
    void showDepth(int value) {
        int d = depth(root, value, 0);
        if (d == -1)
            cout << "Value not found in tree" << endl;
        else
            cout << "Depth of " << value << " = " << d << endl;
    }
};
/*
=======================================================
MAIN FUNCTION (TESTING ALL CONCEPTS)
=======================================================
*/
int main() {
    Tree t;
    /*
    Creating Tree:
           10
          /  \
         5    15
        / \     \
       2   7     20
    */
    t.insert(10);
    t.insert(5);
    t.insert(15);
    t.insert(2);
    t.insert(7);
    t.insert(20);
    cout << "==============================" << endl;
    t.showPreorder();   // Root Left Right
```

```
    t.showPostorder();  // Left Right Root
    cout << "================================" << endl;
    t.showHeight();
    cout << "================================" << endl;
    t.showDepth(7);
    t.showDepth(20);
    t.showDepth(100);
    cout << "================================" << endl;
    return 0;
}
```

Key Takeaways

- A **tree** is a hierarchical non-linear data structure.
- Trees consist of nodes connected via parent-child relationships.
- C++ represents trees using **structures and pointers**.
- Tree interfaces help in abstraction and modular design.
- Linked tree structures allow dynamic memory usage.
- Traversal algorithms are essential for processing tree data.
- **Preorder** processes root first, **Postorder** processes root last.
- Depth and height are key metrics for analyzing tree efficiency.
- Trees are fundamental in compilers, databases, AI, and operating systems.