Data Structures & Algorithms (C++)

Binary Tree ADT

Introduction to Binary Tree ADT

A **Binary Tree Abstract Data Type (ADT)** defines a tree structure where each node can have at most **two children**, known as the left child and right child.

It focuses on **what operations are allowed** (insert, traverse, delete) rather than how they are implemented. This abstraction helps separate interface from implementation.

Binary trees are widely used in searching, sorting, expression evaluation, compilers, and decision-making systems.

Example Binary Tree

1. C++ Binary Tree Interface

Introduction

A binary tree interface defines the **set of operations** that a binary tree must support. It does not specify how these operations are implemented, only what they should do.

This is useful in object-oriented programming where multiple implementations (linked list, array, vector) can share the same interface.

It improves **modularity, reusability, and abstraction**.

C++ Interface Example

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

```
    virtual void postorder() = 0;
};
```

Explanation

- virtual functions enforce implementation in derived classes
- = 0 makes them abstract (pure virtual functions)
- Ensures all tree implementations follow the same structure

2. Properties of Binary Trees

Introduction

Binary trees have specific mathematical and structural properties that define their behavior and performance.

These properties help in analyzing complexity and designing efficient algorithms.

They are also essential in understanding balanced trees and search efficiency.

Key Properties

- Each node has **maximum 2 children**
- A tree with **n nodes has n−1 edges**
- Maximum nodes at level i = **2^i**
- Maximum nodes in height h = **2^(h+1) − 1**
- Minimum height = $\log_2(n)$

Types of Binary Trees

Type	Description
Full Binary Tree	Every node has 0 or 2 children
Complete Binary Tree	All levels filled except last
Perfect Binary Tree	All levels completely filled
Skewed Tree	Nodes only on one side

3. Linked Structure for Binary Trees

Introduction

A linked structure represents a binary tree using **dynamic memory (pointers)** instead of arrays.

Each node contains:

- Data
- Pointer to left child
- Pointer to right child

This structure is flexible and widely used in real-world applications.

C++ Node Structure

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

   Node(int value) {
      data = value;
      left = right = NULL;
   }
};
```

Advantages

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

Disadvantages

- Pointer complexity
- Extra memory for pointers
- Slower traversal compared to arrays

4. Vector-Based Binary Tree Structure

Introduction

A vector-based binary tree stores nodes in an **array or vector**, usually following index-based rules.

This structure is commonly used in **heap implementations**.

It avoids pointers and uses mathematical relationships for navigation.

Index Rules

If index = i:

- Left child = 2i + 1
- Right child = 2i + 2
- Parent = (i - 1) / 2

C++ Example

```
#include <vector>
using namespace std;

class BinaryTreeVector {
   vector<int> tree;

public:
   void insert(int value) {
      tree.push_back(value);
   }

   void display() {
      for (int i = 0; i < tree.size(); i++) {
         cout << tree[i] << " ";
      }
   }
};
```

Advantages

- Fast indexing
- No pointer overhead
- Efficient for complete trees (heaps)

Disadvantages

- Wastes space for sparse trees
- Fixed structure limitations
- Not good for general trees

5. Traversals of a Binary Tree

Introduction

Tree traversal means visiting all nodes in a specific order.

Traversals are essential for searching, evaluating expressions, and processing data.

There are three main DFS traversals.

Types of Traversals

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

C++ Example

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

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

Explanation

- Preorder: used for copying tree
- Inorder: used for BST sorting
- Postorder: used for deletion

6. Template Function Pattern

Introduction

Template functions in C++ allow writing **generic algorithms** that work with different data types.

In binary trees, templates allow nodes to store integers, floats, strings, or custom objects.

This improves code reusability and flexibility.

C++ Template Example

```
template <typename T>
struct Node {
   T data;
   Node* left;
   Node* right;

   Node(T value) {
      data = value;
      left = right = NULL;
   }
};
```

Template Traversal Function

```
template <typename T>
void preorder(Node<T>* root) {
   if (root == NULL) return;

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

Advantages

- Reusable code
- Type flexibility
- Reduces duplication
- Used in STL-like structures

7. Representing General Trees with Binary Trees

Introduction

A general tree allows a node to have more than two children, but a binary tree restricts it to two.

We can still represent a general tree using a binary tree technique called:
Left-Child Right-Sibling Representation

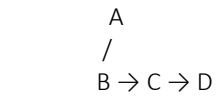
Concept

- Left pointer → first child
- Right pointer → next sibling

Example

General Tree:

Binary Representation:



C++ Structure

```
struct Node {
   int data;
   Node* leftChild;
   Node* rightSibling;
};
```

Advantages

- Converts general tree to binary form
- Saves memory
- Simplifies traversal algorithms

Disadvantages

- Less intuitive structure
- Complex pointer handling

Summary Table

Topic	Purpose
Binary Tree ADT	Defines operations only
Interface	Ensures abstraction
Linked Structure	Dynamic tree using pointers
Vector Structure	Array-based implementation
Traversals	Visit nodes systematically
Templates	Generic tree design
General Tree Representation	Converts n-ary tree to binary

C++ Implementation

```
#include <iostream>
#include <vector>
using namespace std;
/*
=====================================================
1. TEMPLATE-BASED NODE STRUCTURE (GENERAL ADT IDEA)
=====================================================
*/
template <typename T>
struct Node {
   T data;
   Node* left;
   Node* right;

   Node(T value) {
      data = value;
      left = right = NULL;
   }
};
/*
=====================================================
2. BINARY TREE INTERFACE (ABSTRACTION)
=====================================================
*/
template <typename T>
class BinaryTreeInterface {
public:
   virtual void insert(T value) = 0;
   virtual void preorder(Node<T>* node) = 0;
   virtual void inorder(Node<T>* node) = 0;
   virtual void postorder(Node<T>* node) = 0;
};
/*
=====================================================
3. LINKED BINARY TREE IMPLEMENTATION
=====================================================
*/
template <typename T>
class BinaryTree : public BinaryTreeInterface<T> {
private:
   Node<T>* root;
public:
   BinaryTree() {
      root = NULL;
   }
```

```
Node<T>* getRoot() {
    return root;
}
/*
Simple BST-style insertion for demonstration
*/
Node<T>* insertNode(Node<T>* node, T value) {
    if (node == NULL)
        return new Node<T>(value);
    if (value < node->data)
        node->left = insertNode(node->left, value);
    else
        node->right = insertNode(node->right, value);
    return node;
}
void insert(T value) override {
    root = insertNode(root, value);
}
/*
==========================
TRAVERSALS
==========================
*/
void preorder(Node<T>* node) override {
    if (node == NULL) return;
    cout << node->data << " ";
    preorder(node->left);
    preorder(node->right);
}
void inorder(Node<T>* node) override {
    if (node == NULL) return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}
void postorder(Node<T>* node) override {
    if (node == NULL) return;
    postorder(node->left);
    postorder(node->right);
    cout << node->data << " ";
}
/*
==========================
HEIGHT OF TREE
==========================
*/
int height(Node<T>* node) {
    if (node == NULL)
```

```
        return -1;
    return max(height(node->left), height(node->right)) + 1;
  }
  /*
  ========================
  DEPTH OF NODE
  ========================
  */
  int depth(Node<T>* node, T value, int d) {
    if (node == NULL)
      return -1;
    if (node->data == value)
      return d;
    int left = depth(node->left, value, d + 1);
    if (left != -1)
      return left;
    return depth(node->right, value, d + 1);
  }
};
/*
=====================================================
4. VECTOR-BASED TREE STRUCTURE (ARRAY REPRESENTATION)
=====================================================
*/
class VectorTree {
private:
  vector<int> tree;
public:
  void insert(int value) {
    tree.push_back(value);
  }
  void display() {
    cout << "\nVector-Based Tree: ";
    for (int i : tree)
      cout << i << " ";
    cout << endl;
  }
};
/*
=====================================================
5. GENERAL TREE REPRESENTATION (LEFT-CHILD RIGHT-SIBLING IDEA)
=====================================================
*/
struct GeneralNode {
  int data;
  GeneralNode* firstChild;
  GeneralNode* nextSibling;
  GeneralNode(int value) {
```

```
        data = value;
        firstChild = nextSibling = NULL;
    }
};
/*
======================================================
MAIN FUNCTION (DEMO OF ALL CONCEPTS)
======================================================
*/
int main() {
    cout << "================ BINARY TREE ADT DEMO ================\n";
    BinaryTree<int> tree;
    /*
        Insert values (BST-style for simplicity)
        Tree structure example:
              10
             /   \
            5     15
           / \     \
          3   7     20
    */
    tree.insert(10);
    tree.insert(5);
    tree.insert(15);
    tree.insert(3);
    tree.insert(7);
    tree.insert(20);
    cout << "\nPreorder Traversal: ";
    tree.preorder(tree.getRoot());
    cout << "\nInorder Traversal: ";
    tree.inorder(tree.getRoot());
    cout << "\nPostorder Traversal: ";
    tree.postorder(tree.getRoot());
    cout << "\n\nHeight of Tree: " << tree.height(tree.getRoot());
    cout << "\nDepth of node 7: " << tree.depth(tree.getRoot(), 7, 0);
    cout << "\nDepth of node 20: " << tree.depth(tree.getRoot(), 20, 0);
    cout << "\nDepth of node 100: " << tree.depth(tree.getRoot(), 100, 0);
    /*
    ======================================================
    VECTOR-BASED TREE DEMO
    ======================================================
    */
    VectorTree vtree;
    vtree.insert(10);
    vtree.insert(5);
    vtree.insert(15);
    vtree.insert(20);
    vtree.display();
```

```
    /*
    ========================================================
    GENERAL TREE STRUCTURE DEMO (JUST STRUCTURE IDEA)
    ========================================================
    */
    GeneralNode* root = new GeneralNode(1);
    root->firstChild = new GeneralNode(2);
    root->firstChild->nextSibling = new GeneralNode(3);
    root->firstChild->nextSibling->nextSibling = new GeneralNode(4);
    cout << "\nGeneral Tree (Left-Child Right-Sibling):\n";
    cout << "Root: " << root->data << endl;
    cout << "Children: 2 -> 3 -> 4\n";
    cout << "\n================ END OF DEMO ================\n";
    return 0;
}
```

What This Program Covers (Exam Mapping)

Binary Tree ADT

- Abstract class (interface)
- Implementation class

Linked Structure

- Node with left/right pointers

Traversals

- Preorder
- Inorder
- Postorder

Tree Functions

- insert()
- height()
- depth()

Vector-Based Structure

- Array-based tree representation

Template Pattern

- Generic tree using template <typename T>

General Tree Representation

- Left-child / right-sibling model

Key Takeaways

- A **Binary Tree ADT** defines operations without implementation details.
- Binary trees support at most **two children per node**.
- They can be implemented using:
    - Linked structures (pointers)
    - Vector/array structures (index-based)
- Traversals are essential:
    - Preorder (Root-first)
    - Inorder (Sorted order in BST)
    - Postorder (Root-last)
- C++ **templates** allow generic tree implementation.
- General trees can be represented using **left-child right-sibling method**.
- Binary trees are foundational for:
    - BST
    - Heaps
    - Expression trees
    - Compilers and AI systems