**DATA STRUCTURE AND ALGORITHMS**

**PROJECT**

**TITLE :**

**B-TREE IMPLENTATION**

**Created By : Laiba Banoori (4654)**

**Submitted to : Miss Arshi Parvaiz**

### Binary Tree Implementation in C++

A **Binary Tree** is a hierarchical data structure in which each node has at most two children referred to as the **left child** and **right child**. It is often used for searching, sorting, and storing hierarchical data. In C++, we can implement a binary tree using structures or classes, and provide functions for basic operations like insertion, traversal, deletion, etc.

Let's break down the key components of binary tree implementation in C++.

### 1. ****Binary Tree Node Structure****

Each node in a binary tree stores a data element and two pointers to its left and right children.

#include <iostream>

using namespace std;

struct Node {

int data; // Data to be stored in the node

Node\* left; // Pointer to left child

Node\* right; // Pointer to right child

// Constructor to initialize a new node

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

### 2. ****Binary Tree Class****

Now we define a class for Binary Tree operations. This class will contain a pointer to the root of the tree and various methods for tree operations such as insertion, traversal, etc.

class BinaryTree {

public:

Node\* root;

BinaryTree() {

root = nullptr; // Initially, the tree is empty

}

// Insert a new node in the binary tree

Node\* insert(Node\* node, int value) {

// If the tree is empty, create the root node

if (node == nullptr) {

return new Node(value);

}

// Otherwise, traverse the tree and insert the node

if (value < node->data) {

node->left = insert(node->left, value);

} else {

node->right = insert(node->right, value);

}

return node;

}

// Public method to insert a node with the given value

void insert(int value) {

root = insert(root, value);

}

// In-order traversal (Left, Root, Right)

void inorder(Node\* node) {

if (node != nullptr) {

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

}

// Pre-order traversal (Root, Left, Right)

void preorder(Node\* node) {

if (node != nullptr) {

cout << node->data << " ";

preorder(node->left);

preorder(node->right);

}

}

// Post-order traversal (Left, Right, Root)

void postorder(Node\* node) {

if (node != nullptr) {

postorder(node->left);

postorder(node->right);

cout << node->data << " ";

}

}

// Wrapper functions for traversal

void inorder() { inorder(root); }

void preorder() { preorder(root); }

void postorder() { postorder(root); }

};

### 3. ****Tree Operations****

**Insertion :**

We can insert nodes into the tree by comparing values and placing them in the correct left or right position according to the binary tree property:

* If the value to be inserted is smaller than the current node, it goes to the left.
* If the value is larger, it goes to the right.

**Traversals :**

Traversal of a binary tree refers to visiting each node in a specific order. There are three common types of depth-first traversal:

* **In-order Traversal**: Left, Root, Right
* **Pre-order Traversal**: Root, Left, Right
* **Post-order Traversal**: Left, Right, Root

Here are the traversal functions implemented in the BinaryTree class above.

**Example :**

int main() {

BinaryTree tree;

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

cout << "In-order Traversal: ";

tree.inorder();

cout << endl;

cout << "Pre-order Traversal: ";

tree.preorder();

cout << endl;

cout << "Post-order Traversal: ";

tree.postorder();

cout << endl;

return 0;

}

### 4. ****Binary Tree Deletion (Optional)****

Deleting a node from a binary tree involves three possible cases:

* If the node to be deleted has no children (leaf node), simply remove it.
* If the node has one child, remove the node and replace it with its child.
* If the node has two children, find the in-order successor or predecessor (the smallest node in the right subtree or the largest node in the left subtree) and replace the deleted node with this successor.

### 5. ****Complexity Analysis****

* **Insertion**: O(log n) for a balanced tree, O(n) for a skewed tree.
* **Traversal**: O(n) as each node must be visited once.
* **Search/Deletion**: O(log n) for a balanced tree, O(n) for a skewed tree.

### 6. ****Conclusion****

The binary tree is a versatile data structure used in various applications such as searching, sorting, and building hierarchical structures. Its implementation in C++ is straightforward using structures or classes. The most common operations (insertion, traversal) are easy to implement and optimize for efficiency. For large-scale or performance-critical applications, a balanced binary tree (e.g., AVL or Red-Black Tree) is preferred to avoid performance degradation in unbalanced trees.