Let's walk through creating a **binary tree** in C++, including a Tree class, some common operations (like traversal), and algorithms.

**Table of Contents:**

1. **Binary Tree Overview**
2. **Tree Class Definition**
3. **Tree Traversal Algorithms**
   * Inorder
   * Preorder
   * Postorder
   * Level Order
4. **Common Algorithms**
   * Insertion
   * Searching
   * Deletion
   * Height of Tree
5. **Code Example**

**1. Binary Tree Overview**

A **binary tree** is a tree data structure where each node has at most two children, referred to as the left child and the right child. The tree is usually built of **nodes**, each containing:

* A value (data)
* A pointer to the left child
* A pointer to the right child

**2. Tree Class Definition**

We'll start by defining the basic structure of a tree node and the tree itself.

#include <iostream>

#include <queue> // For level order traversal

using namespace std;

// TreeNode structure

struct TreeNode {

int data; // Value of the node

TreeNode\* left; // Pointer to the left child

TreeNode\* right; // Pointer to the right child

// Constructor to create a new node

TreeNode(int value) {

data = value;

left = right = nullptr;

}

};

// BinaryTree class

class BinaryTree {

private:

TreeNode\* root; // Pointer to the root node

public:

// Constructor

BinaryTree() {

root = nullptr;

}

// Get the root (used for internal algorithms)

TreeNode\* getRoot() {

return root;

}

// Insert a node (wrapper function for recursion)

void insert(int value) {

root = insertRec(root, value);

}

// Search for a value in the tree

bool search(int value) {

return searchRec(root, value);

}

// Delete a node from the tree

void deleteNode(int value) {

root = deleteRec(root, value);

}

// Get the height of the tree

int height() {

return heightRec(root);

}

// Traversal functions

void inorder() {

inorderRec(root);

cout << endl;

}

void preorder() {

preorderRec(root);

cout << endl;

}

void postorder() {

postorderRec(root);

cout << endl;

}

void levelOrder();

private:

// Recursive function to insert a node

TreeNode\* insertRec(TreeNode\* node, int value);

// Recursive function to search for a value

bool searchRec(TreeNode\* node, int value);

// Recursive function to delete a node

TreeNode\* deleteRec(TreeNode\* node, int value);

// Recursive function to get the height of the tree

int heightRec(TreeNode\* node);

// Traversal helper functions

void inorderRec(TreeNode\* node);

void preorderRec(TreeNode\* node);

void postorderRec(TreeNode\* node);

};

**3. Tree Traversal Algorithms**

Traversal means visiting all the nodes of the tree in some order. There are different types of traversals:

**Inorder Traversal (Left, Root, Right)**

In this traversal, we first visit the left subtree, then the root node, and finally the right subtree.

void BinaryTree::inorderRec(TreeNode\* node) {

if (node == nullptr)

return;

inorderRec(node->left); // Visit left subtree

cout << node->data << " "; // Visit root node

inorderRec(node->right); // Visit right subtree

}

**Preorder Traversal (Root, Left, Right)**

In preorder traversal, the root node is visited first, followed by the left subtree, and finally the right subtree.

void BinaryTree::preorderRec(TreeNode\* node) {

if (node == nullptr)

return;

cout << node->data << " "; // Visit root node

preorderRec(node->left); // Visit left subtree

preorderRec(node->right); // Visit right subtree

}

**Postorder Traversal (Left, Right, Root)**

In postorder traversal, we first visit the left subtree, then the right subtree, and finally the root node.

void BinaryTree::postorderRec(TreeNode\* node) {

if (node == nullptr)

return;

postorderRec(node->left); // Visit left subtree

postorderRec(node->right); // Visit right subtree

cout << node->data << " "; // Visit root node

}

**Level Order Traversal (BFS)**

In level-order traversal, we visit nodes level by level from left to right using a queue.

void BinaryTree::levelOrder() {

if (root == nullptr) return;

queue<TreeNode\*> q;

q.push(root);

while (!q.empty()) {

TreeNode\* current = q.front();

q.pop();

cout << current->data << " ";

// Enqueue left and right children

if (current->left != nullptr)

q.push(current->left);

if (current->right != nullptr)

q.push(current->right);

}

cout << endl;

}

**4. Common Algorithms**

**Insertion**

In a binary tree, the insertion process follows the rule of placing the value in the correct position, either left (if smaller) or right (if larger).

TreeNode\* BinaryTree::insertRec(TreeNode\* node, int value) {

if (node == nullptr) {

return new TreeNode(value);

}

// If value is less, go left; otherwise, go right

if (value < node->data) {

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

} else {

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

}

return node;

}

**Searching**

The search operation finds if a node with a particular value exists in the tree.

bool BinaryTree::searchRec(TreeNode\* node, int value) {

if (node == nullptr) {

return false;

}

if (node->data == value) {

return true;

}

if (value < node->data) {

return searchRec(node->left, value); // Search left

} else {

return searchRec(node->right, value); // Search right

}

}

**Deletion**

Deleting a node in a binary tree involves three cases:

1. Node is a leaf node.
2. Node has one child.
3. Node has two children (replace with the minimum value in the right subtree).

TreeNode\* BinaryTree::deleteRec(TreeNode\* node, int value) {

if (node == nullptr) return node;

// Traverse the tree to find the node to delete

if (value < node->data) {

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

} else if (value > node->data) {

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

} else {

// Node found, handle the deletion cases

if (node->left == nullptr) {

TreeNode\* temp = node->right;

delete node;

return temp;

} else if (node->right == nullptr) {

TreeNode\* temp = node->left;

delete node;

return temp;

}

// Node with two children, find the inorder successor

TreeNode\* temp = node->right;

while (temp && temp->left != nullptr) {

temp = temp->left;

}

// Copy inorder successor's data to this node

node->data = temp->data;

// Delete the inorder successor

node->right = deleteRec(node->right, temp->data);

}

return node;

}

**Height of the Tree**

The height of a tree is the longest path from the root to a leaf node.

int BinaryTree::heightRec(TreeNode\* node) {

if (node == nullptr)

return 0;

int leftHeight = heightRec(node->left);

int rightHeight = heightRec(node->right);

return 1 + max(leftHeight, rightHeight);

}

**5. Code Example**

Now let's see a complete example where we:

* Insert nodes
* Perform traversal
* Search for a value
* Delete a node

int main() {

BinaryTree tree;

// Insert nodes

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Traversals

cout << "Inorder Traversal: ";

tree.inorder(); // Expected: 20 30 40 50 60 70 80

cout << "Preorder Traversal: ";

tree.preorder(); // Expected: 50 30 20 40 70 60 80

cout << "Postorder Traversal: ";

tree.postorder(); // Expected: 20 40 30 60 80 70 50

cout << "Level Order Traversal: ";

tree.levelOrder(); // Expected: 50 30 70 20 40 60 80

// Search for a value

int key = 40;

if (tree.search(key))

cout << key << " found in the tree." << endl;

else

cout << key << " not found in the tree." << endl;

// Delete a node

tree.deleteNode(30);

cout << "Inorder Traversal after deleting 30: ";

tree.inorder(); // Expected: 20 40 50 60 70 80

// Height of tree

cout << "Height of tree: " << tree.height() << endl;

return 0;

}

**Output:**

Inorder Traversal: 20 30 40 50 60 70 80

Preorder Traversal: 50 30 20 40 70 60 80

Postorder Traversal: 20 40 30 60 80 70 50

Level Order Traversal: 50 30 70 20 40 60 80

40 found in the tree.

Inorder Traversal after deleting 30: 20 40 50 60 70 80

Height of tree: 3

**Explanation:**

* The tree is built by inserting nodes, and various traversal methods show the order in which nodes are visited.
* We search for a value (40) in the tree.
* After deleting the node with the value 30, we check the inorder traversal to confirm the deletion.
* The height function gives the depth of the binary tree.

This example demonstrates how to work with binary trees in C++ using recursion, traversal algorithms, and key operations like insertion, deletion, and searching.