Here's a **complete C++ program for Binary Search Tree (BST) operations**, including:

* Insertion
* Deletion
* Searching
* Inorder Traversal
* Preorder Traversal
* Postorder Traversal
* Minimum and Maximum value

**✅ Binary Search Tree Operations in C++**

#include <iostream>

using namespace std;

// Definition of Node

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Insert a node

Node\* insert(Node\* root, int key) {

if (root == nullptr) {

return new Node(key);

}

if (key < root->data) {

root->left = insert(root->left, key);

} else if (key > root->data) {

root->right = insert(root->right, key);

}

return root;

}

// Search a node

Node\* search(Node\* root, int key) {

if (root == nullptr || root->data == key)

return root;

if (key < root->data)

return search(root->left, key);

return search(root->right, key);

}

// Find minimum value node

Node\* findMin(Node\* root) {

while (root && root->left != nullptr)

root = root->left;

return root;

}

// Delete a node

Node\* deleteNode(Node\* root, int key) {

if (root == nullptr)

return root;

if (key < root->data) {

root->left = deleteNode(root->left, key);

} else if (key > root->data) {

root->right = deleteNode(root->right, key);

} else {

// Case 1: No child

if (root->left == nullptr && root->right == nullptr) {

delete root;

return nullptr;

}

// Case 2: One child

else if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

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

Node\* temp = root->left;

delete root;

return temp;

}

// Case 3: Two children

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Inorder Traversal (Left, Root, Right)

void inorder(Node\* root) {

if (root != nullptr) {

inorder(root->left);

cout << root->data << " ";

inorder(root->right);

}

}

// Preorder Traversal (Root, Left, Right)

void preorder(Node\* root) {

if (root != nullptr) {

cout << root->data << " ";

preorder(root->left);

preorder(root->right);

}

}

// Postorder Traversal (Left, Right, Root)

void postorder(Node\* root) {

if (root != nullptr) {

postorder(root->left);

postorder(root->right);

cout << root->data << " ";

}

}

// Find maximum value

Node\* findMax(Node\* root) {

while (root && root->right != nullptr)

root = root->right;

return root;

}

// Main function to demonstrate BST operations

int main() {

Node\* root = nullptr;

int choice, value;

do {

cout << "\n--- Binary Search Tree Operations ---\n";

cout << "1. Insert\n2. Delete\n3. Search\n4. Inorder Traversal\n";

cout << "5. Preorder Traversal\n6. Postorder Traversal\n7. Find Min\n";

cout << "8. Find Max\n9. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

root = insert(root, value);

break;

case 2:

cout << "Enter value to delete: ";

cin >> value;

root = deleteNode(root, value);

break;

case 3:

cout << "Enter value to search: ";

cin >> value;

if (search(root, value))

cout << value << " found in the tree.\n";

else

cout << value << " not found in the tree.\n";

break;

case 4:

cout << "Inorder Traversal: ";

inorder(root);

cout << endl;

break;

case 5:

cout << "Preorder Traversal: ";

preorder(root);

cout << endl;

break;

case 6:

cout << "Postorder Traversal: ";

postorder(root);

cout << endl;

break;

case 7:

if (root)

cout << "Minimum value: " << findMin(root)->data << endl;

else

cout << "Tree is empty.\n";

break;

case 8:

if (root)

cout << "Maximum value: " << findMax(root)->data << endl;

else

cout << "Tree is empty.\n";

break;

case 9:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice. Try again.\n";

}

} while (choice != 9);

return 0;

}

**✅ AVL Tree Implementation in C++**

#include <iostream>

#include <algorithm>

using namespace std;

// Node structure

struct Node {

int key;

Node\* left;

Node\* right;

int height;

};

// Utility function to get the height of a node

int height(Node\* n) {

return (n == nullptr) ? 0 : n->height;

}

// Utility function to get the balance factor of a node

int getBalance(Node\* n) {

return (n == nullptr) ? 0 : height(n->left) - height(n->right);

}

// Create a new node

Node\* createNode(int key) {

Node\* node = new Node();

node->key = key;

node->left = node->right = nullptr;

node->height = 1; // new node is initially added at leaf

return node;

}

// Right rotate

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Perform rotation

x->right = y;

y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

// Return new root

return x;

}

// Left rotate

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

// Return new root

return y;

}

// Insert a node in AVL tree

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

// 1. Perform the normal BST insertion

if (node == nullptr)

return createNode(key);

if (key < node->key)

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

else if (key > node->key)

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

else

return node; // No duplicates allowed

// 2. Update height

node->height = 1 + max(height(node->left), height(node->right));

// 3. Get balance factor

int balance = getBalance(node);

// 4. Balance the tree

// Case 1 - Left Left

if (balance > 1 && key < node->left->key)

return rightRotate(node);

// Case 2 - Right Right

if (balance < -1 && key > node->right->key)

return leftRotate(node);

// Case 3 - Left Right

if (balance > 1 && key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Case 4 - Right Left

if (balance < -1 && key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

// Return unchanged node

return node;

}

// Inorder traversal

void inorder(Node\* root) {

if (root != nullptr) {

inorder(root->left);

cout << root->key << " ";

inorder(root->right);

}

}

// Main function

int main() {

Node\* root = nullptr;

int choice, key;

do {

cout << "\n--- AVL Tree Operations ---\n";

cout << "1. Insert\n2. Inorder Traversal\n3. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter key to insert: ";

cin >> key;

root = insert(root, key);

break;

case 2:

cout << "Inorder Traversal: ";

inorder(root);

cout << endl;

break;

case 3:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice!\n";

}

} while (choice != 3);

return 0;

}