**Write a C++ program to perform the following operations:**

**a) Insert an element into a binary search tree.**

**b) Delete an element from a binary search tree. Give code in short**

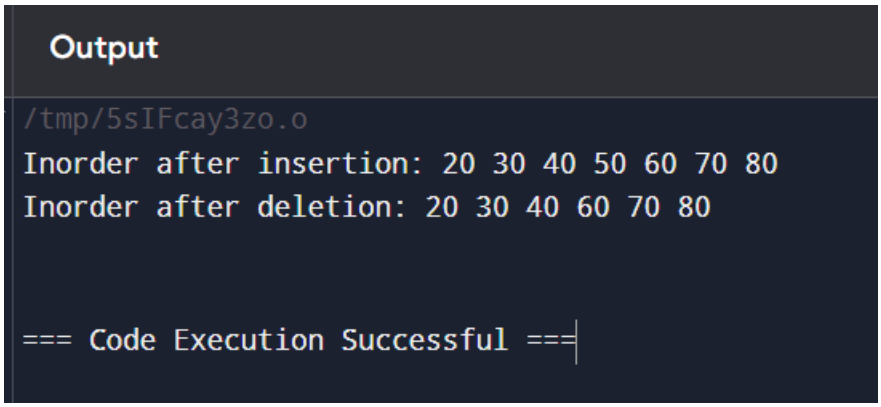
## Program:

```
#include <iostream>
using namespace std;
// Node structure for BST
struct Node {
    int data;
    Node* left;
    Node* right;

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

// Insert an element into BST
Node* insert(Node* root, int value) {
    if (!root) return new Node(value);
    if (value < root->data)
        root->left = insert(root->left, value);
    else
        root->right = insert(root->right, value);
    return root;
}

// Find minimum value node in the right subtree (used in deletion)
Node* findMin(Node* root) {
    while (root && root->left)
        root = root->left;
```

```
    return root;
}

// Delete an element from BST
Node* deleteNode(Node* root, int value) {
    if (!root) return root;
    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else { // Node to be deleted found
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = findMin(root->right); // Find min in right subtree
        root->data = temp->data; // Replace with inorder successor
        root->right = deleteNode(root->right, temp->data); // Delete successor
    }
    return root;
}

// Inorder traversal for testing
void inorder(Node* root) {
    if (root) {
        inorder(root->left);
```

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

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);

    cout << "Inorder after insertion: ";
    inorder(root);
    cout << endl;
    root = deleteNode(root, 50); // Deleting node 50
    cout << "Inorder after deletion: ";
    inorder(root);
    cout << endl;
    return 0;
}
```

Output:




## Explanation:

**1. Node Structure for BST**

The Node structure represents each node in the binary search tree.

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

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};
```

- **int data**: Stores the value of the node.
- **Node* left**: Points to the left child of the current node (values smaller than data).
- **Node* right**: Points to the right child of the current node (values greater than data).
- The constructor Node(int value) initializes the node's data and sets its left and right pointers to nullptr.

**2. Insert an Element into BST**

```
Node* insert(Node* root, int value) {
   if (!root) return new Node(value);
   if (value < root->data)
      root->left = insert(root->left, value);
   else
      root->right = insert(root->right, value);
   return root;
}
```

- **Goal**: To insert a new value into the tree while maintaining the BST property (left subtree has smaller values, right subtree has larger values).
- **How it works**:
    1. **Base case**: If root is nullptr, we create a new node with the given value and return it. This happens when we've found the appropriate empty spot to insert the value.
    2. **Recursive calls**: If the value is smaller than the current node's data, we recursively insert it into the left subtree. If the value is larger, we insert it into the right subtree.
    3. **Return the root**: After the recursive calls, we return the root to maintain the tree's structure.

**3. Find the Minimum Value Node in the Right Subtree**

This function helps to find the minimum value in a subtree. It's particularly useful when deleting a node with two children, where we need to replace the node with its inorder successor (smallest node in the right subtree).

```
Node* findMin(Node* root) {
   while (root && root->left)
      root = root->left;
   return root;
}
```

- **How it works**: Starting from the given root, the function keeps moving left until it finds the leftmost (smallest) node and returns it.

**4. Delete an Element from BST**

```
Node* deleteNode(Node* root, int value) {
    if (!root) return root;
    if (value < root->data)
        root->left = deleteNode(root->left, value);
    else if (value > root->data)
        root->right = deleteNode(root->right, value);
    else {
        if (!root->left) {
            Node* temp = root->right;
            delete root;
            return temp;
        } else if (!root->right) {
            Node* temp = root->left;
            delete root;
            return temp;
        }
        Node* temp = findMin(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
    return root;
}
```

- **Goal**: To delete a node with the given value while maintaining the BST structure.
- **Steps**:
    1. **Find the node to delete**: If value is smaller than the current node's data, we recursively search in the left subtree. If it's larger, we search in the right subtree.
    2. **Node to delete found**:
        - **Case 1: No left child**: If the node to be deleted has no left child, replace the node with its right child and delete the node.
        - **Case 2: No right child**: If the node has no right child, replace it with its left child and delete the node.

- **Case 3: Two children**: Find the node's inorder successor (the smallest node in its right subtree) using findMin(), copy the successor's value to the current node, and then delete the successor.

3. **Return the updated root**: This ensures the correct structure of the tree after the deletion.

**5. Inorder Traversal**

Inorder traversal prints the nodes of a BST in ascending order.

```
void inorder(Node* root) {
    if (root) {
        inorder(root->left);  // Visit left subtree
        cout << root->data << " ";  // Visit current node
        inorder(root->right);  // Visit right subtree
    }
}
```

- **How it works**:
    1. Visit the left subtree recursively.
    2. Print the current node's data.
    3. Visit the right subtree recursively.
    - This ensures that the values are printed in ascending order (since in a BST, left < root < right).

**6. Main Function**

```
int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 70);
    insert(root, 20);
    insert(root, 40);
    insert(root, 60);
    insert(root, 80);
```

```
    cout << "Inorder after insertion: ";
    inorder(root);
    cout << endl;

    root = deleteNode(root, 50); // Deleting node 50

    cout << "Inorder after deletion: ";
    inorder(root);
    cout << endl;

    return 0;
}
```

- **Tree construction**: The code first constructs the BST by inserting values into it using the insert function. The values inserted are: 50, 30, 70, 20, 40, 60, and 80.
- **Display the tree (inorder traversal)**: The inorder traversal after insertion prints the tree in ascending order.

Inorder after insertion: 20 30 40 50 60 70 80

- **Delete a node**: The code deletes the node with value 50. Since node 50 has two children, the deletion function finds the inorder successor (which is 60), replaces 50 with 60, and deletes 60.
- **Display the updated tree (inorder traversal)**: After deletion, the new inorder traversal prints:

Inorder after deletion: 20 30 40 60 70 80