// Binary Tree in C++

#include <stdlib.h>

#include <iostream>

using namespace std;

struct node {

int data;

struct node \*left;

struct node \*right;

};

// New node creation

struct node \*newNode(int data) {

struct node \*node = (struct node \*)malloc(sizeof(struct node));

node->data = data;

node->left = NULL;

node->right = NULL;

return (node);

}

// Traverse Preorder

void traversePreOrder(struct node \*temp) {

if (temp != NULL) {

cout << " " << temp->data;

traversePreOrder(temp->left);

traversePreOrder(temp->right);

}

}

// Traverse Inorder

void traverseInOrder(struct node \*temp) {

if (temp != NULL) {

traverseInOrder(temp->left);

cout << " " << temp->data;

traverseInOrder(temp->right);

}

}

// Traverse Postorder

void traversePostOrder(struct node \*temp) {

if (temp != NULL) {

traversePostOrder(temp->left);

traversePostOrder(temp->right);

cout << " " << temp->data;

}

}

int main() {

struct node \*root = newNode(1);

root->left = newNode(2);

root->right = newNode(3);

root->left->left = newNode(4);

cout << "preorder traversal: ";

traversePreOrder(root);

cout << "\nInorder traversal: ";

traverseInOrder(root);

cout << "\nPostorder traversal: ";

traversePostOrder(root);

}

// Binary Search Tree operations in C++

#include <iostream>

using namespace std;

struct node {

int key;

struct node \*left, \*right;

};

// Create a node

struct node \*newNode(int item) {

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// Inorder Traversal

void inorder(struct node \*root) {

if (root != NULL) {

// Traverse left

inorder(root->left);

// Traverse root

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

// Traverse right

inorder(root->right);

}

}

// Insert a node

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

// Return a new node if the tree is empty

if (node == NULL) return newNode(key);

// Traverse to the right place and insert the node

if (key < node->key)

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

else

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

return node;

}

// Find the inorder successor

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

// Find the leftmost leaf

while (current && current->left != NULL)

current = current->left;

return current;

}

// Deleting a node

struct node \*deleteNode(struct node \*root, int key) {

// Return if the tree is empty

if (root == NULL) return root;

// Find the node to be deleted

if (key < root->key)

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

else if (key > root->key)

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

else {

// If the node is with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

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

struct node \*temp = root->left;

free(root);

return temp;

}

// If the node has two children

struct node \*temp = minValueNode(root->right);

// Place the inorder successor in position of the node to be deleted

root->key = temp->key;

// Delete the inorder successor

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

}

return root;

}

// Driver code

int main() {

struct node \*root = NULL;

root = insert(root, 8);

root = insert(root, 3);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 7);

root = insert(root, 10);

root = insert(root, 14);

root = insert(root, 4);

cout << "Inorder traversal: ";

inorder(root);

cout << "\nAfter deleting 10\n";

root = deleteNode(root, 10);

cout << "Inorder traversal: ";

inorder(root);

}