//Binary Search Tree Program

#include <iostream>

#include <cstdlib>

using namespace std;

class BinarySearchTree

{

private:

struct tree\_node

{

tree\_node\* left;

tree\_node\* right;

int data;

};

tree\_node\* root;

public:

BinarySearchTree()

{

root = NULL;

}

bool isEmpty() const { return root==NULL; }

void print\_inorder();

void inorder(tree\_node\*);

void print\_preorder();

void preorder(tree\_node\*);

void print\_postorder();

void postorder(tree\_node\*);

void insert(int);

void remove(int);

};

// Smaller elements go left

// larger elements go right

void BinarySearchTree::insert(int d)

{

tree\_node\* t = new tree\_node;

tree\_node\* parent;

t->data = d;

t->left = NULL;

t->right = NULL;

parent = NULL;

// is this a new tree?

if(isEmpty()) root = t;

else

{

//Note: ALL insertions are as leaf nodes

tree\_node\* curr;

curr = root;

// Find the Node's parent

while(curr)

{

parent = curr;

if(t->data > curr->data) curr = curr->right;

else curr = curr->left;

}

if(t->data < parent->data)

parent->left = t;

else

parent->right = t;

}

}

void BinarySearchTree::remove(int d)

{

//Locate the element

bool found = false;

if(isEmpty())

{

cout<<" This Tree is empty! "<<endl;

return;

}

tree\_node\* curr;

tree\_node\* parent;

curr = root;

while(curr != NULL)

{

if(curr->data == d)

{

found = true;

break;

}

else

{

parent = curr;

if(d>curr->data) curr = curr->right;

else curr = curr->left;

}

}

if(!found)

{

cout<<" Data not found! "<<endl;

return;

}

// 3 cases :

// 1. We're removing a leaf node

// 2. We're removing a node with a single child

// 3. we're removing a node with 2 children

// Node with single child

if((curr->left == NULL && curr->right != NULL)|| (curr->left != NULL

&& curr->right == NULL))

{

if(curr->left == NULL && curr->right != NULL)

{

if(parent->left == curr)

{

parent->left = curr->right;

delete curr;

}

else

{

parent->right = curr->right;

delete curr;

}

}

else // left child present, no right child

{

if(parent->left == curr)

{

parent->left = curr->left;

delete curr;

}

else

{

parent->right = curr->left;

delete curr;

}

}

return;

}

//We're looking at a leaf node

if( curr->left == NULL && curr->right == NULL)

{

if(parent->left == curr) parent->left = NULL;

else parent->right = NULL;

delete curr;

return;

}

//Node with 2 children

// replace node with smallest value in right subtree

if (curr->left != NULL && curr->right != NULL)

{

tree\_node\* chkr;

chkr = curr->right;

if((chkr->left == NULL) && (chkr->right == NULL))

{

curr = chkr;

delete chkr;

curr->right = NULL;

}

else // right child has children

{

//if the node's right child has a left child

// Move all the way down left to locate smallest element

if((curr->right)->left != NULL)

{

tree\_node\* lcurr;

tree\_node\* lcurrp;

lcurrp = curr->right;

lcurr = (curr->right)->left;

while(lcurr->left != NULL)

{

lcurrp = lcurr;

lcurr = lcurr->left;

}

curr->data = lcurr->data;

delete lcurr;

lcurrp->left = NULL;

}

else

{

tree\_node\* tmp;

tmp = curr->right;

curr->data = tmp->data;

curr->right = tmp->right;

delete tmp;

}

}

return;

}

}

void BinarySearchTree::print\_inorder()

{

inorder(root);

}

void BinarySearchTree::inorder(tree\_node\* p)

{

if(p != NULL)

{

if(p->left) inorder(p->left);

cout<<" "<<p->data<<" ";

if(p->right) inorder(p->right);

}

else return;

}

void BinarySearchTree::print\_preorder()

{

preorder(root);

}

void BinarySearchTree::preorder(tree\_node\* p)

{

if(p != NULL)

{

cout<<" "<<p->data<<" ";

if(p->left) preorder(p->left);

if(p->right) preorder(p->right);

}

else return;

}

void BinarySearchTree::print\_postorder()

{

postorder(root);

}

void BinarySearchTree::postorder(tree\_node\* p)

{

if(p != NULL)

{

if(p->left) postorder(p->left);

if(p->right) postorder(p->right);

cout<<" "<<p->data<<" ";

}

else return;

}

int main()

{

BinarySearchTree b;

int ch,tmp,tmp1;

while(1)

{

cout<<endl<<endl;

cout<<" Binary Search Tree Operations "<<endl;

cout<<" ----------------------------- "<<endl;

cout<<" 1. Insertion/Creation "<<endl;

cout<<" 2. In-Order Traversal "<<endl;

cout<<" 3. Pre-Order Traversal "<<endl;

cout<<" 4. Post-Order Traversal "<<endl;

cout<<" 5. Removal "<<endl;

cout<<" 6. Exit "<<endl;

cout<<" Enter your choice : ";

cin>>ch;

switch(ch)

{

case 1 : cout<<" Enter Number to be inserted : ";

cin>>tmp;

b.insert(tmp);

break;

case 2 : cout<<endl;

cout<<" In-Order Traversal "<<endl;

cout<<" -------------------"<<endl;

b.print\_inorder();

break;

case 3 : cout<<endl;

cout<<" Pre-Order Traversal "<<endl;

cout<<" -------------------"<<endl;

b.print\_preorder();

break;

case 4 : cout<<endl;

cout<<" Post-Order Traversal "<<endl;

cout<<" --------------------"<<endl;

b.print\_postorder();

break;

case 5 : cout<<" Enter data to be deleted : ";

cin>>tmp1;

b.remove(tmp1);

break;

case 6 :

return 0;

}

}

}