#include <iostream>

using namespace std;

struct Node

{

int key, height;

Node\* L, \* R;

};

class BST

{

private:

Node\* root;

public:

BST()

{

root = NULL;

};

Node\* getRoot()

{

return root;

}

Node\* insert(int num, Node\* temp)

{

if (temp == NULL)

{

temp = new Node;

temp->key = num;

temp->L = temp->R = NULL;

}

else if (num < temp->key)

{

temp->L = insert(num, temp->L);

}

else if (num > temp->key)

{

temp->R = insert(num, temp->R);

}

return temp;

}

Node\* insert1(int num, Node\* temp)

{

if (temp == NULL)

{

temp = new Node;

temp->key = num;

temp->L = temp->R = NULL;

}

else if (num < temp->key)

{

temp->L = insert(num, temp->L);

}

else if (num > temp->key)

{

temp->R = insert(num, temp->R);

}

return temp;

}

//print display of tree

void display(Node\* r, int space)

{

if (r == NULL)

{

return;

}

display(r->L, space);

cout << r->key << "\n";

display(r->R, space);

return;

}

//print inorder of tree

void inorderPrint(Node\* T)

{

if (T == NULL)

{

return;

}

else

{

inorderPrint(T->L);

cout << T->key << " ";

inorderPrint(T->R);

}

}

//print preorder of tree

void print\_preorder(Node\* node)

{

if (node == NULL)

{

return;

}

else

{

cout << node->key << " ";

inorderPrint(node->L);

inorderPrint(node->R);

}

}

//print postorder of tree

void print\_postorder(Node\* node)

{

if (node == NULL)

{

return;

}

else

{

inorderPrint(node->L);

inorderPrint(node->R);

cout << node->key << " ";

}

}

void printtree(Node\* temp, int space)

{

int COUNT = 10;

if (temp == NULL)

{

return;

}

space += COUNT;

printtree(temp->R, space);

cout << endl;

for (int i = COUNT; i < space; i++)

cout << " ";

cout << temp->key << "\n";

printtree(temp->L, space);

}

Node\* minval(struct Node\* node)

{

Node\* temp = node;

while (temp && temp->L != NULL)

{

temp = temp->L;

}

return temp;

}

Node\* deleteNode(Node\* temp1, int key)

{

if (temp1 == NULL)

{

return temp1;

}

if (key < temp1->key)

{

temp1->L = deleteNode(temp1->L, key);

}

else if (key > temp1->key)

{

temp1->R = deleteNode(temp1->R, key);

}

else

{

if (temp1->L == NULL)

{

Node\* temp = temp1->L;

delete temp1->L;

delete temp1->R;

cout << temp1->R;

temp = NULL;

return temp;

}

else if (temp1->R == NULL)

{

Node\* temp = temp1->L;

delete temp1->R;

temp->L = NULL;

temp->R = NULL;

temp = NULL;

return temp;

}

Node\* temp = minval(temp1->R);

temp1->key = temp->key;

temp1->R = deleteNode(temp1->R, temp->key);

}

return temp1;

}

int height(Node\* t)

{

int h = 0;

if (t != NULL)

{

int l\_height = height(t->L);

int r\_height = height(t->R);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int depth(struct Node\* node, int data, int level)

{

if (node == NULL)

{

return 0;

}

if (node->key == data)

return level;

int next\_level = depth(node->L, data, level + 1);

if (next\_level != 0)

{

return next\_level;

}

next\_level = depth(node->R, data, level + 1);

return next\_level;

}

bool tree\_balance(Node\* temp)

{

if (!temp)

{

return true;

}

int l\_height = height(temp->L);

int r\_height = height(temp->R);

if (abs(l\_height - r\_height) > 1)

return false;

return tree\_balance(temp->L) && tree\_balance(temp->R);

}

~BST()

{

empty(root);

}

void empty(Node\* temp)

{

if (NULL(temp))

{

return;

}

else

{

empty(temp->L);

empty(temp->R);

delete temp;

}

}

int Breadth(Node\* temp, int level)

{

if (temp == 0)

{

return 0;

}

if (level == 1)

{

return 1;

}

else if (level > 1)

{

return Breadth(temp->L, level - 1) + Breadth(temp->R, level - 1);

}

}

int predecessor(Node\* temp, int key)

{

Node\* predecessor;

if (temp == NULL)

{

return NULL;

}

if (temp->key == key)

{

if (temp->L != NULL)

{

Node\* temp1 = temp->L;

while (temp1->R)

{

temp1 = temp1->R;

}

predecessor = temp1;

return predecessor->key;

cout << endl;

}

}

}

int Successor(Node\* temp, int data)

{

Node\* succesor;

if (temp == NULL)

{

return NULL;

}

if (temp->R != NULL)

{

Node\* temp1 = temp->R;

while (temp1->L)

{

temp1 = temp1->L;

}

if (temp1->L == NULL && temp1->R == NULL)

{

cout << "There is no successor";

}

succesor = temp1;

return succesor->key;

}

}

Node\* minval1(struct Node\* node)

{

Node\* temp = node;

while (temp && temp->L != NULL)

{

temp = temp->L;

}

return temp;

}

Node\* trimbelowk(Node\* temp1, int key)

{

if (temp1 == NULL)

{

return temp1;

}

if (key < temp1->key)

{

temp1->L = trimbelowk(temp1->L, key);

}

else if (key > temp1->key)

{

temp1->R = trimbelowk(temp1->R, key);

}

else

{

if (temp1->L == NULL)

{

Node\* temp = temp1->L;

delete temp1->L;

delete temp1->R;

cout << temp1->R;

temp = NULL;

return temp;

}

else if (temp1->R == NULL)

{

Node\* temp = temp1->L;

delete temp1->R;

temp->L = NULL;

temp->R = NULL;

temp = NULL;

return temp;

}

Node\* temp = minval1(temp1->R);

temp1->key = temp->key;

temp1->R = trimbelowk(temp1->R, temp->key);

}

return temp1;

}

//Node\* sortedArrayToBST(int arr[],

// int start, int end)

//{

// /\* Base Case \*/

// if (start > end)

// return NULL;

// /\* Get the middle element and make it root \*/

// int mid = (start + end) / 2;

// Node\* temp = insert(arr[mid]);

// /\* Recursively construct the left subtree

// and make it left child of root \*/

// temp->L = sortedArrayToBST(arr, start,

// mid - 1);

// /\* Recursively construct the right subtree

// and make it right child of root \*/

// temp->R = sortedArrayToBST(arr, mid + 1, end);

// return temp;

//}

void operator=(const Node& op)

{

int key;

key = op.key;

Node\* temp = NULL;

Node\* l = temp->L;

temp->L = new Node(\*op.L);

delete l;

Node\* r = temp->R;

temp->R = new Node(\*op.R);

delete r;

}

Node\* copy(Node\* copy)

{

Node\* temp = new Node(\*copy);

return temp;

}

bool areIdentical(Node\* temp, Node\* temp1)

{

if (temp == NULL && temp1 == NULL)

{

return true;

}

if (temp == NULL || temp1 == NULL)

{

return false;

}

return (temp->key == temp1->key) && areIdentical(temp->L, temp1->L) &&

areIdentical(temp->R, temp1->R);

}

bool isSubtree(Node\* T, Node\* S)

{

if (S == NULL)

{

return true;

}

if (T == NULL)

{

return false;

}

if (areIdentical(T, S))

{

return true;

}

return isSubtree(T->L, S) || isSubtree(T->R, S);

}

};

int main()

{

BST tree;

Node\* T = tree.getRoot();

int opt, num;

bool exit = false;

while (!exit)

{

cout << "Press 1 for Inserting values" << endl;

cout << "Press 2 for display trees(Inorder,Preorder,postorder and tree form)" << endl;

cout << "press 3 to delete element " << endl;

cout << "press 4 to height of a tree " << endl;

cout << "press 5 to depth of a tree " << endl;

cout << "press 6 to check whether tree is balanced or NOT " << endl;

cout << "press 7 to check width of a tree " << endl;

cout << "press 8 to find predecossor and succesor of a tree " << endl;

cout << "press 9 to check whether tree is a sub tree or NOT " << endl;

cout << "press 10 to delete the nodes below " << endl;

cout << "press 11 for copy constructor " << endl;

cout << "Press 12 to Exit" << endl;

cout << "Your Option is: ";

cin >> opt;

{

if (opt == 1)

{

cout << "Enter element: ";

cin >> num;

T = tree.insert(num, T);

}

else if (opt == 2)

{

int op;

cout << "Press 1 for display" << endl;

cout << "Press 2 for Inorder display" << endl;

cout << "Press 3 for preorder display" << endl;

cout << "Press 4 for postorder display" << endl;

cout << "Press 5 for tree form display" << endl;

cout << "Enter your choice :";

cin >> op;

if (op == 1)

{

cout << "Display of Tree: ";

tree.display(T, 5);

cout << endl;

}

else if (op == 2)

{

cout << "Inorder print of Tree: ";

tree.inorderPrint(T);

cout << endl;

}

else if (op == 3)

{

cout << "Preorder print of Tree: ";

tree.print\_preorder(T);

cout << endl;

}

else if (op == 4)

{

cout << "postorder print of Tree: ";

tree.print\_postorder(T);

cout << endl;

}

else if (op == 5)

{

tree.printtree(T, 0);

cout << endl;

}

}

else if (opt == 3)

{

int a;

cout << "Enter element to delete :";

cin >> a;

tree.deleteNode(T, a);

cout << endl;

}

else if (opt == 4)

{

tree.height(T);

cout << endl;

}

else if (opt == 5)

{

int d;

cout << "Enter number whose depth you want to find :";

cin >> d;

tree.depth(T, d, 1);

cout << endl;

}

else if (opt == 6)

{

if (tree.tree\_balance(T))

{

cout << "Tree is balanced" << endl;

}

else

{

cout << "Tree is not Balanced" << endl;

}

cout << endl;

}

else if (opt == 7)

{

int l;

cout << "Enter level whose breadth you wanna find :";

cin >> l;

cout << "Breadth is :";

cout << tree.Breadth(T, l);

cout << endl;

}

else if (opt == 8)

{

int key, key1;

cout << "Enter key to find predecossr :";

cin >> key;

cout << "Enter key to find successor :";

cin >> key1;

cout << "Predecossor of key is :";

cout << tree.predecessor(T, key);

cout << "Successor of key is :";

cout << tree.Successor(T, key1);

cout << endl;

}

else if (opt == 9)

{

int num1;

Node\* T1 = tree.getRoot();

while (1)

{

int o;

cout << "Enter element: ";

cin >> num1;

T1 = tree.insert1(num1, T1);

cout << "Press any key to continue" << endl;

cout << "Press -1 to exit :";

cin >> o;

if (o == -1)

{

break;

}

}

if (tree.isSubtree(T1, T))

{

cout << "\nTree is subtree";

}

else

{

cout << "\nTree is not subtree";

}

}

else if (opt == 10)

{

int a;

cout << "Enter element to delete below :";

cin >> a;

tree.trimbelowk(T, a);

cout << endl;

}

else if (opt == 11)

{

tree.copy(T);

cout << endl;

}

else if (opt == 12)

{

exit = true;

break;

}

else

{

cout << "Option is invalid!" << endl;

}

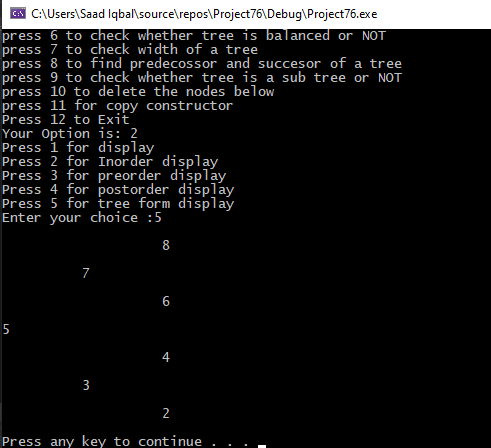
}

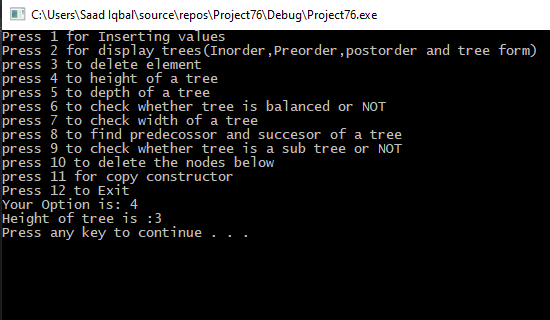
system("pause");

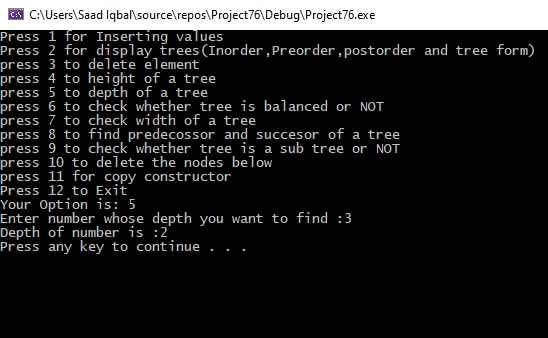
system("cls");

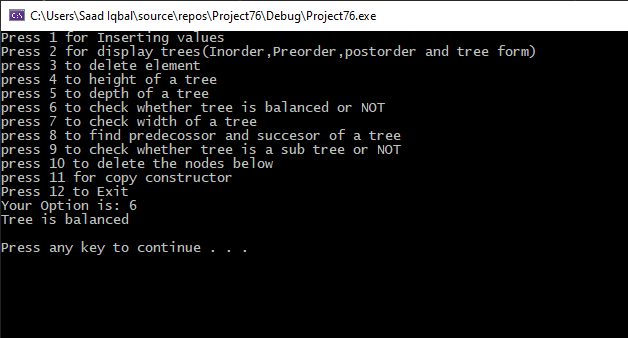
}

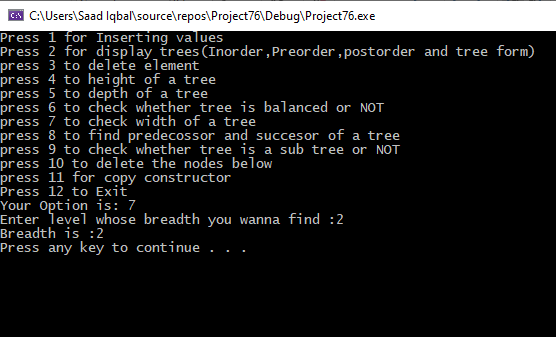
}

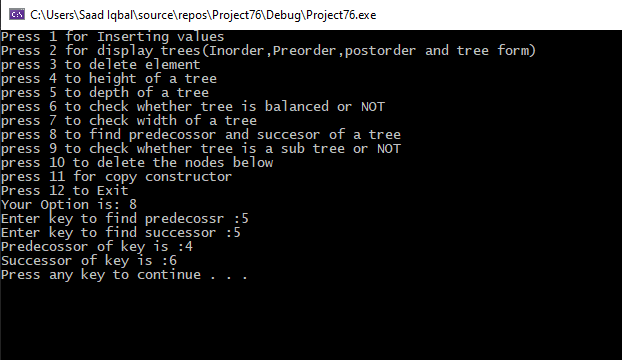




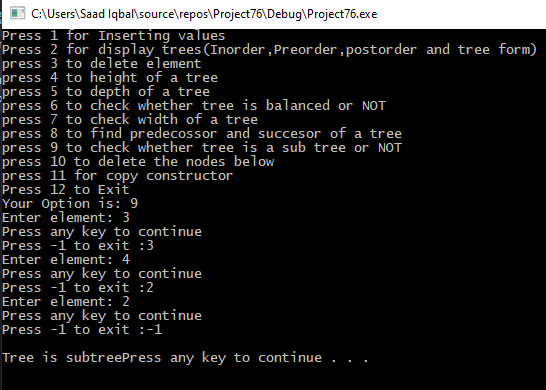
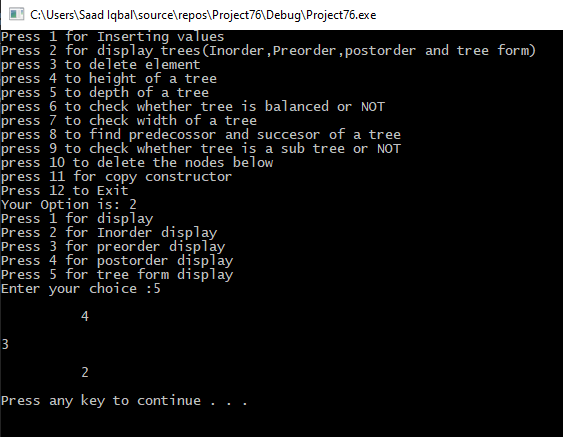








***OUTPUT for ISsubtree:***



***OUTPUT for Trim below:***

