Assignment #3

21i0759

Task 1:

#include<iostream>

using namespace std;

class Node

{

public:

int number;

Node\* left;

Node\* right;

Node()

{

}

Node(int data)

{

this->number = data;

}

};

template <class T>

class BST

{

private:

Node\* root;

public:

int Height(Node\* root)

{

if (root == nullptr)

{

cout<< "Tree Is Empty" << endl;

return 0;

}

int LHeight = Height(root->left);

int RHeight = Height(root->right);

return max(LHeight, RHeight) + 1;

}

int treeNodeCount(Node\*root)

{

if (root = nullptr)

{

return 0;

}

else

{

return treeNodeCount(root->left) + treeNodeCount(root->right) + 1;

}

}

int treeLeavesCount(Node\* root)

{

if (root == NULL)

return 0;

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

return 1;

else

return treeLeavesCount(root->left) + treeLeavesCount(root->right);

}

void Insert(int number)

{

//Create A New Node

Node\* newnode = new Node;

newnode->number = number;

newnode->left = nullptr;

newnode->right = nullptr;

//Case 1

//If Tree Is Empty

if (!root)

{

root = newnode;

}

else //If Tree is not empty

{

Node\* ptr = new Node;

while (true)

{

if (number < ptr->number)

{

// Left subtree

if (ptr->left)

{

ptr = ptr->left;

}

else

{

ptr->left = newnode;

return;

}

}

else if (number > ptr->number)

{

// Left subtree

if (ptr->right)

{

ptr = ptr->right;

}

else

{

ptr->right = newnode;

return;

}

}

else

{

cout << "Value Already Exsists" << endl;

break;

}

}

}

}

void deleteNode(T number, Node\*& ptr)

{

if (ptr == nullptr)

{

cout << "Tree is Empty" << endl;

}

else if (number < ptr->number)

{

deleteNode(number, ptr->left);

}

else if (number > ptr->number)

{

deleteNode(number, ptr->right);

}

else

deleted(ptr);

}

void deleted(Node\*& nodePtr)

{

Node\* tempNodePtr;

if (nodePtr->right == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->left;

delete tempNodePtr;

}

else if (nodePtr->left == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->right; // Reattach the right child

delete tempNodePtr;

}

else

{

tempNodePtr = nodePtr->right;

while (tempNodePtr->left)

{

tempNodePtr = tempNodePtr->left;

}

tempNodePtr->left = nodePtr->left;

tempNodePtr = nodePtr;

nodePtr = nodePtr->right;

delete tempNodePtr;

}

}

void search(T number)

{

Node\* ptr = root;

while (ptr)

{

if (ptr->number == number)

{

cout << "Value Found" << endl;

}

else if (number < ptr->number)

{

ptr = ptr->left;

}

else if (number < ptr->number)

{

ptr = ptr->right;

}

}

return;

}

int findmax(Node\* root)

{

if (root == nullptr)

{

cout << "Tree is Empty" << endl;

return -1;

}

Node\* current = root;

while (current->right != nullptr)

{

current = current->right;

}

return current->number;

}

int findmin(Node\* root)

{

if (root == nullptr)

{

cout << "Tree is Empty" << endl;

return -1;

}

Node\* current = root;

while (current->left != nullptr)

{

current = current->left;

}

return current->number;

}

void PostOrder(Node\* root)

{

if (root != nullptr)

{

PostOrder(root->left);

PostOrder(root->right);

cout << root->number << " ";

}

}

void InOrder(Node\* root)

{

if (root != nullptr)

{

InOrder(root->left);

cout << root->number << " ";

InOrder(root->right);

}

}

void PreOrder(Node\* root)

{

if (root != nullptr)

{

cout << root->number << " ";

PreOrder(root->left);

PreOrder(root->right);

}

}

void Mirror(Node\* p)

{

if (p == nullptr)

{

return;

}

else

{

Node\* t = p;

Mirror(p->left);

Mirror(p->right);

t = p->left;

p->left = p->right;

p->right = t;

}

}

Node\* arraytobst(int array[], int s, int e)

{

if (s > e)

{

return nullptr;

}

//Fnding the Middle Elmeent Of an Array

// Then Make The Middle Element of an Array as Root

int m = (s + e) / 2;

Node\* newnode = new Node(m);

newnode->left = arraytobst(array, s, m - 1);

newnode->right = arraytobst(array, m + 1, e);

return newnode;

}

int sumofallnodes(Node\* root)

{

if (root == nullptr)

{

return 0;

}

else

{

return root->number + sumofallnodes(root->left) + sumofallnodes(root->right);

}

}

Node\* converttogreaterbst(Node\* root)

{

int sum = 0;

if (root == nullptr)

{

return nullptr;

}

else

{

converttogreaterbst(root->right);

sum = sum + root->number;

root->number = sum;

converttogreaterbst(root->left);

return root;

}

}

};

#include<iostream>

using namespace std;

class Node

{

public:

int number;

Node\* left;

Node\* right;

Node()

{

}

Node(int data)

{

this->number = data;

}

};

template <class T>

class BST

{

private:

Node\* root;

public:

int Height(Node\* root)

{

if (root == nullptr)

{

cout<< "Tree Is Empty" << endl;

return 0;

}

int LHeight = Height(root->left);

int RHeight = Height(root->right);

return max(LHeight, RHeight) + 1;

}

int treeNodeCount(Node\*root)

{

if (root = nullptr)

{

return 0;

}

else

{

return treeNodeCount(root->left) + treeNodeCount(root->right) + 1;

}

}

int treeLeavesCount(Node\* root)

{

if (root == NULL)

return 0;

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

return 1;

else

return treeLeavesCount(root->left) + treeLeavesCount(root->right);

}

void Insert(int number)

{

//Create A New Node

Node\* newnode = new Node;

newnode->number = number;

newnode->left = nullptr;

newnode->right = nullptr;

//Case 1

//If Tree Is Empty

if (!root)

{

root = newnode;

}

else //If Tree is not empty

{

Node\* ptr = new Node;

while (true)

{

if (number < ptr->number)

{

// Left subtree

if (ptr->left)

{

ptr = ptr->left;

}

else

{

ptr->left = newnode;

return;

}

}

else if (number > ptr->number)

{

// Left subtree

if (ptr->right)

{

ptr = ptr->right;

}

else

{

ptr->right = newnode;

return;

}

}

else

{

cout << "Value Already Exsists" << endl;

break;

}

}

}

}

void deleteNode(T number, Node\*& ptr)

{

if (ptr == nullptr)

{

cout << "Tree is Empty" << endl;

}

else if (number < ptr->number)

{

deleteNode(number, ptr->left);

}

else if (number > ptr->number)

{

deleteNode(number, ptr->right);

}

else

deleted(ptr);

}

void deleted(Node\*& nodePtr)

{

Node\* tempNodePtr;

if (nodePtr->right == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->left;

delete tempNodePtr;

}

else if (nodePtr->left == NULL)

{

tempNodePtr = nodePtr;

nodePtr = nodePtr->right; // Reattach the right child

delete tempNodePtr;

}

else

{

tempNodePtr = nodePtr->right;

while (tempNodePtr->left)

{

tempNodePtr = tempNodePtr->left;

}

tempNodePtr->left = nodePtr->left;

tempNodePtr = nodePtr;

nodePtr = nodePtr->right;

delete tempNodePtr;

}

}

void search(T number)

{

Node\* ptr = root;

while (ptr)

{

if (ptr->number == number)

{

cout << "Value Found" << endl;

}

else if (number < ptr->number)

{

ptr = ptr->left;

}

else if (number < ptr->number)

{

ptr = ptr->right;

}

}

return;

}

int findmax(Node\* root)

{

if (root == nullptr)

{

cout << "Tree is Empty" << endl;

return -1;

}

Node\* current = root;

while (current->right != nullptr)

{

current = current->right;

}

return current->number;

}

int findmin(Node\* root)

{

if (root == nullptr)

{

cout << "Tree is Empty" << endl;

return -1;

}

Node\* current = root;

while (current->left != nullptr)

{

current = current->left;

}

return current->number;

}

void PostOrder(Node\* root)

{

if (root != nullptr)

{

PostOrder(root->left);

PostOrder(root->right);

cout << root->number << " ";

}

}

void InOrder(Node\* root)

{

if (root != nullptr)

{

InOrder(root->left);

cout << root->number << " ";

InOrder(root->right);

}

}

void PreOrder(Node\* root)

{

if (root != nullptr)

{

cout << root->number << " ";

PreOrder(root->left);

PreOrder(root->right);

}

}

void Mirror(Node\* p)

{

if (p == nullptr)

{

return;

}

else

{

Node\* t = p;

Mirror(p->left);

Mirror(p->right);

t = p->left;

p->left = p->right;

p->right = t;

}

}

Node\* arraytobst(int array[], int s, int e)

{

if (s > e)

{

return nullptr;

}

//Fnding the Middle Elmeent Of an Array

// Then Make The Middle Element of an Array as Root

int m = (s + e) / 2;

Node\* newnode = new Node(m);

newnode->left = arraytobst(array, s, m - 1);

newnode->right = arraytobst(array, m + 1, e);

return newnode;

}

int sumofallnodes(Node\* root)

{

if (root == nullptr)

{

return 0;

}

else

{

return root->number + sumofallnodes(root->left) + sumofallnodes(root->right);

}

}

Node\* converttogreaterbst(Node\* root)

{

int sum = 0;

if (root == nullptr)

{

return nullptr;

}

else

{

converttogreaterbst(root->right);

sum = sum + root->number;

root->number = sum;

converttogreaterbst(root->left);

return root;

}

}

};

Task3

#include <iostream>

using namespace std;

class TreeNode {

public:

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

class MinimalHeightBST {

public:

// Function to convert a sorted array to a BST of minimal height

TreeNode\* sortedArrayToBST(int sortedArray[], int start, int end) {

if (start > end)

return nullptr;

// Find the middle element of the sorted array

int mid = (start + end) / 2;

// Create a new node with the middle element as the root

TreeNode\* root = new TreeNode(sortedArray[mid]);

// Recursively construct the left and right subtrees

root->left = sortedArrayToBST(sortedArray, start, mid - 1);

root->right = sortedArrayToBST(sortedArray, mid + 1, end);

return root;

}

};

// Function to perform an inorder traversal of the BST (for testing purposes)

void inorderTraversal(TreeNode\* root) {

if (root == nullptr)

return;

inorderTraversal(root->left);

cout << root->val << " ";

inorderTraversal(root->right);

}

int main() {

int sortedArray[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int n = sizeof(sortedArray) / sizeof(sortedArray[0]);

MinimalHeightBST converter;

TreeNode\* root = converter.sortedArrayToBST(sortedArray, 0, n - 1);

cout << "Inorder traversal of the minimal height BST: ";

inorderTraversal(root);

cout << endl;

while (root != nullptr) {

TreeNode\* temp = root;

root = root->right;

delete temp;

}

system("pause");

return 0;

}

Output:



Task4:

#include <iostream>

using namespace std;

class TreeNode {

public:

int val;

TreeNode\* left;

TreeNode\* right;

TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

class BinaryTree {

public:

TreeNode\* root;

BinaryTree() : root(nullptr) {}

// Recursive function to calculate the sum of all nodes in the binary tree

int calculateSum(TreeNode\* root) {

if (root == nullptr)

return 0;

int leftSum = calculateSum(root->left);

int rightSum = calculateSum(root->right);

return root->val + leftSum + rightSum;

}

// Recursive function for inorder traversal of the binary tree

void inorderTraversal(TreeNode\* root) {

if (root == nullptr)

return;

inorderTraversal(root->left);

cout << root->val << " ";

inorderTraversal(root->right);

}

};

int main() {

// Create a sample binary tree

BinaryTree tree;

tree.root = new TreeNode(8);

tree.root->left = new TreeNode(4);

tree.root->right = new TreeNode(12);

tree.root->left->left = new TreeNode(2);

tree.root->left->right = new TreeNode(6);

tree.root->right->left = new TreeNode(10);

tree.root->right->right = new TreeNode(14);

cout << "Inorder traversal of the binary tree: ";

tree.inorderTraversal(tree.root);

cout << endl;

int sum = tree.calculateSum(tree.root);

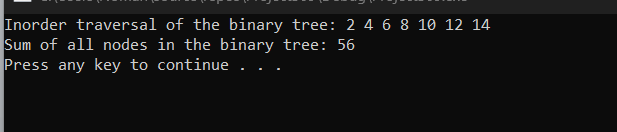
cout << "Sum of all nodes in the binary tree: " << sum << endl;

system("pause");

return 0;

}

Output:



Task 5(part1):

#include<iostream>

using namespace std;

class BST

{

public:

int\* array;

int\* arr;

int size;

int tree[10];

BST(int s)

{

size = s;

array = new int[size];

arr = new int[size];

for (int i = 0; i < size; i++)

{

array[i] = arr[i] = -1;

}

}

void input()

{

cout << "ENTER ELEMENTS OF AN ARRAY : " << endl;

for (int i = 0; i < size; i++)

{

cin >> array[i];

}

for (int i = 0; i < size; i++)

{

arr[i] = array[i];

}

int parent, parent1;

for (int i = 0; i < 10; i++)

{

tree[i] = -1;

}

tree[1] = array[0];

parent = parent1 = 1;

for (int i = 1; i < size; i++)

{

if (array[i] > tree[parent])

{

if (array[i] < tree[parent])

{

tree[parent \* 2] = array[i];

parent = parent \* 2;

}

else

{

tree[(parent \* 2) + 1] = array[i];

parent = (parent \* 2) + 1;

}

}

}

parent = 1;

for (int i = 1; i < size; i++)

{

if (array[i] < tree[parent1])

{

if (array[i] < tree[parent])

{

tree[parent \* 2] = array[i];

parent = parent \* 2;

}

else if (array[i] > tree[parent])

{

tree[(parent \* 2) + 1] = array[i];

parent = (parent \* 2) + 1;

}

}

}

cout << " THE TREE IS : " << endl;

for (int i = 0; i < 10; i++)

{

cout << i << " --> " << tree[i] << endl;

}

}

void permutation()

{

for (int i = 1; i < 5 - 1; i++)

{

swap(arr[1], arr[i + 1]);

int parent, parent1;

int tree1[10];

for (int i = 0; i < 10; i++)

{

tree1[i] = -1;

}

tree1[1] = arr[0];

parent = parent1 = 1;

for (int i = 1; i < size; i++)

{

if (arr[i] > tree1[parent])

{

if (arr[i] < tree1[parent])

{

tree1[parent \* 2] = arr[i];

parent = parent \* 2;

}

else

{

tree1[(parent \* 2) + 1] = arr[i];

parent = (parent \* 2) + 1;

}

}

}

parent = 1;

for (int i = 1; i < size; i++)

{

if (arr[i] < tree1[parent1])

{

if (arr[i] < tree1[parent])

{

tree1[parent \* 2] = arr[i];

parent = parent \* 2;

}

else if (arr[i] > tree[parent])

{

tree1[(parent \* 2) + 1] = arr[i];

parent = (parent \* 2) + 1;

}

}

}

for (int i = 0; i < 10; i++)

{

if (tree1[i] == tree[i])

{

if (i == 9)

{

for (int i = 0; i < size; i++)

{

cout << arr[i];

}

}

}

else

{

break;

}

}

}

}

};

int main()

{

int s;

cout << " ENTER NO OF ELEMENTS IN AN ARRAY : " << endl;

cin >> s;

BST obj(s);

cout << "=============================================================" << endl;

obj.input();

cout << "=============================================================" << endl;

cout << " THE PERMUTATION ARE GIVEN BVELOW " << endl;

for (int i = 0; i < 5; i++)

{

obj.permutation();

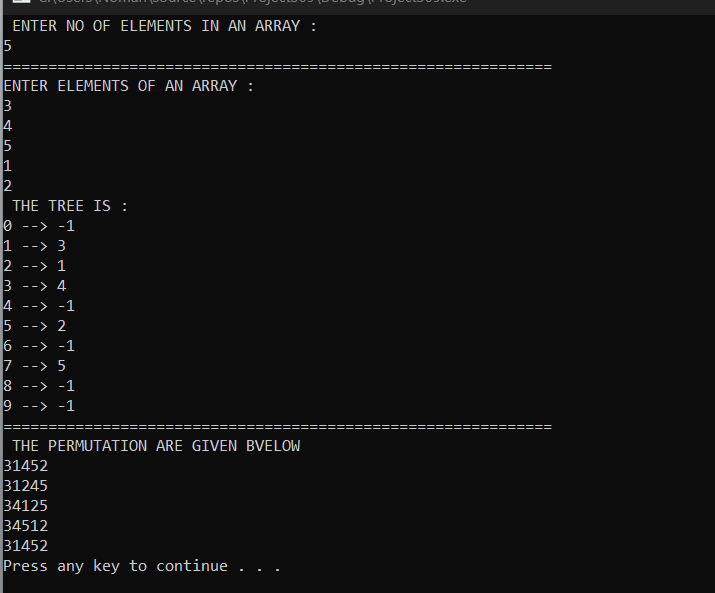
cout << endl;

}

system("pause");

}

Output:



Task #6:

#include <iostream>

using namespace std;

class TreeNode

{

public:

int val; //attributes

TreeNode\* left;

TreeNode\* right;

TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

class BinaryTree

{

public:

TreeNode\* root;

BinaryTree() : root(nullptr) {}

// Function to perform a reverse inorder traversal of the BST

void reverseInorder(TreeNode\* root)

{

if (root == nullptr)

return;

reverseInorder(root->right);

cout << root->val << " ";

reverseInorder(root->left);

}

};

int main()

{

// Create a sample binary search tree

BinaryTree tree;

tree.root = new TreeNode(1);

tree.root->left = new TreeNode(2);

tree.root->right = new TreeNode(3);

tree.root->left->left = new TreeNode(4);

tree.root->left->right = new TreeNode(5);

tree.root->right->left = new TreeNode(6);

tree.root->right->right = new TreeNode(8);

cout << "BST in descending order: ";

tree.reverseInorder(tree.root);

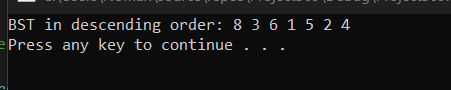
cout << endl;

system("pause");

return 0;

}

Output:



Task #7:

#include <iostream>

using namespace std;

class TreeNode

{

public:

int val; //attributes

TreeNode\* left;

TreeNode\* right;

TreeNode(int v) : val(v), left(nullptr), right(nullptr) {}

};

class BinaryTree

{

public:

TreeNode\* root;

BinaryTree()

{

root = nullptr;

}

// Function to delete the given value and its factors from the BST

TreeNode\* deletevalandFactors(TreeNode\* root, int value)

{

if (root == nullptr)

return nullptr;

if (value < root->val) {

root->left = deletevalandFactors(root->left, value);

}

else if (value > root->val) {

root->right = deletevalandFactors(root->right, value);

}

else {

// Found the value in the tree, delete it and its factors

root->left = deletevalandFactors(root->left, value);

root->right = deletevalandFactors(root->right, value);

delete root;

return nullptr; // Return nullptr to remove the value from the tree

}

return root;

}

// Function for inorder traversal of the BST (left -> root -> right)

void inorderTraversal(TreeNode\* root)

{

if (root == nullptr)

return;

inorderTraversal(root->left);

cout << root->val << " ";

inorderTraversal(root->right);

}

};

int main() {

// Create a sample binary search tree

BinaryTree obj;

obj.root = new TreeNode(1);

obj.root->left = new TreeNode(3);

obj.root->right = new TreeNode(5);

obj.root->left->left = new TreeNode(7);

obj.root->left->right = new TreeNode(8);

obj.root->right->left = new TreeNode(10);

obj.root->right->right = new TreeNode(12);

int valueToDelete = 7;

cout << "Original tree = ";

obj.inorderTraversal(obj.root);

cout << endl;

obj.root = obj.deletevalandFactors(obj.root, valueToDelete);

cout << "Tree after deleting " << valueToDelete << " and its factors = ";

obj.inorderTraversal(obj.root);

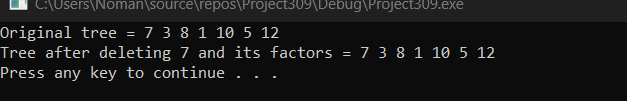
cout << endl;

system("pause");

return 0;

}

Output:



Task #8:

#include <iostream>

using namespace std;

struct TreeNode

{

int data; // attributes

TreeNode\* left;

TreeNode\* right;

TreeNode(int v) : data(v), left(nullptr), right(nullptr) {} // intialzing using intializer list

};

int largest(TreeNode\* root)

{

if (root == nullptr)

return 0;

int maxVal = root->data;

int leftMax = largest(root->left);

int rightMax = largest(root->right);

if (leftMax > maxVal)

maxVal = leftMax;

if (rightMax > maxVal)

maxVal = rightMax;

return maxVal;

}

// function to delete the node with the maximum value in the binary tree

TreeNode\* deletelargest(TreeNode\* root, int maxVal) {

if (root == nullptr)

return nullptr;

if (root->data == maxVal) {

if (root->left == nullptr) {

TreeNode\* temp = root->right;

delete root;

return temp;

}

if (root->right == nullptr) {

TreeNode\* temp = root->left;

delete root;

return temp;

}

// find the inorder successor (smallest value in the right subtree)

// and replace the node with it.

TreeNode\* smallestvalue = root->right;

while (smallestvalue->left != nullptr)

smallestvalue = smallestvalue->left;

root->data = smallestvalue->data;

root->right = deletelargest(root->right, smallestvalue->data);

}

else if (root->data < maxVal)

{

root->right = deletelargest(root->right, maxVal);

}

else

{

root->left = deletelargest(root->left, maxVal);

}

return root;

}

void inorder(TreeNode\* root)

{

if (root == nullptr)

return;

inorder(root->left);

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

inorder(root->right);

}

int main() {

// Create a sample binary tree

TreeNode\* root = new TreeNode(8);

root->left = new TreeNode(4);

root->right = new TreeNode(12);

root->left->left = new TreeNode(2);

root->left->right = new TreeNode(6);

root->right->left = new TreeNode(9);

root->right->right = new TreeNode(14);

cout << "Original tree: ";

inorder(root);

cout <<endl;

int maxVal = largest(root);

root = deletelargest(root, maxVal);

cout << "Tree after deleting the maximum element = " << maxVal << ": ";

inorder(root);

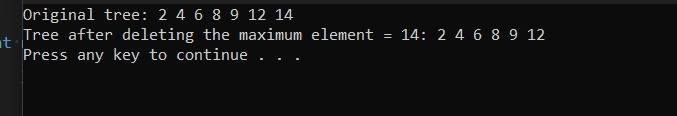
cout <<endl;

system("pause");

return 0;

}

Output:



Task #9:

#include <iostream>

using namespace std;

struct TreeNode

{

int val; //attributes

TreeNode\* left;

TreeNode\* right;

TreeNode(int v)

{

val=v;

left=nullptr;

right = nullptr;

}

};

TreeNode\* removeLeafNodes(TreeNode\* root)

{

if (root == nullptr)

{

return nullptr;

}

// Check if the node is a leaf node

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

{

delete root;

return nullptr;

}

// Recursively remove leaf nodes from the left and right subtrees

root->left = removeLeafNodes(root->left);

root->right = removeLeafNodes(root->right);

// If the current node has become a leaf node after removing its children, delete it

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

{

delete root;

return nullptr;

}

return root;

}

void inorderTraversal(TreeNode\* root) {

if (root == nullptr) return;

inorderTraversal(root->left);

cout << root->val << " ";

inorderTraversal(root->right);

}

int main() {

// Create a sample binary tree

TreeNode\* root = new TreeNode(1);

root->left = new TreeNode(2);

root->right = new TreeNode(3);

root->left->left = new TreeNode(4);

root->left->right = new TreeNode(5);

root->right->left = new TreeNode(6);

root->right->right = new TreeNode(7);

cout << "Original tree: ";

inorderTraversal(root);

cout << endl;

// Remove leaf nodes

root = removeLeafNodes(root);

cout << "Tree after removing leaf nodes: ";

inorderTraversal(root);

cout << endl;

system("pause");

return 0;

}

Output:

