1. Linear Search

#include<iostream>

#include<vector>

using namespace std;

int main() {

int n, s;

cout << "Enter size of vector" << endl;

cin >> n;

vector<int> arr(n);

cout << "Enter " << n << " elements:" << endl;

for (int i = 0; i < n; i++) {

cin >> arr[i];

}

cout << "Enter Element For Search" << endl;

cin >> s;

// Searching Operation

bool found = false;

for (int i = 0; i < n; i++) {

if (arr[i] == s) {

cout << "Element Found at Index " << i << endl;

found = true;

break;

}

}

if (!found) {

cout << "Element Not Found" << endl;

}

return 0;}



Time Complexity O(n)

Space Complexity O(1)

Stable yes

In-place YES

1. Binary Search

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> arr = {2,5,8,12,16,23,38,56,72,91};

int target = 23;

int left = 0, right = arr.size() - 1;

// Binary Search

while (left <= right) {

int mid = left + (right - left) / 2;

if (arr[mid] == target) {

cout << "Element found at index: " << mid << endl;

return 0;

}

else if (arr[mid] > target) {

right = mid - 1;

}

else {

left = mid + 1;

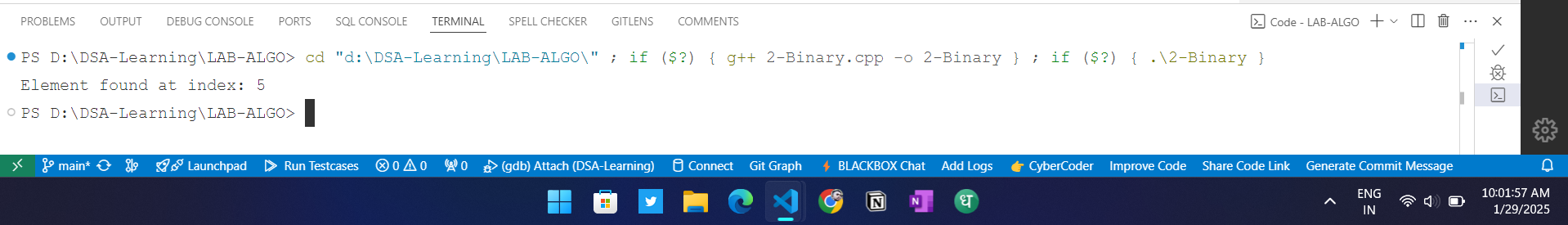
}

}

cout << "Element not found" << endl;

return 0;

}



Time Complexity O(log n)

Space Complexity O(1)

Stable NO

In-place YES

1. BUBBLE SORT

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

for (int i = 0; i < n; i++) {

cin >> arr[i];

}

// Bubble Sort

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

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

}

}

}

// Print Sorted Values

cout << "Sorted Elements: ";

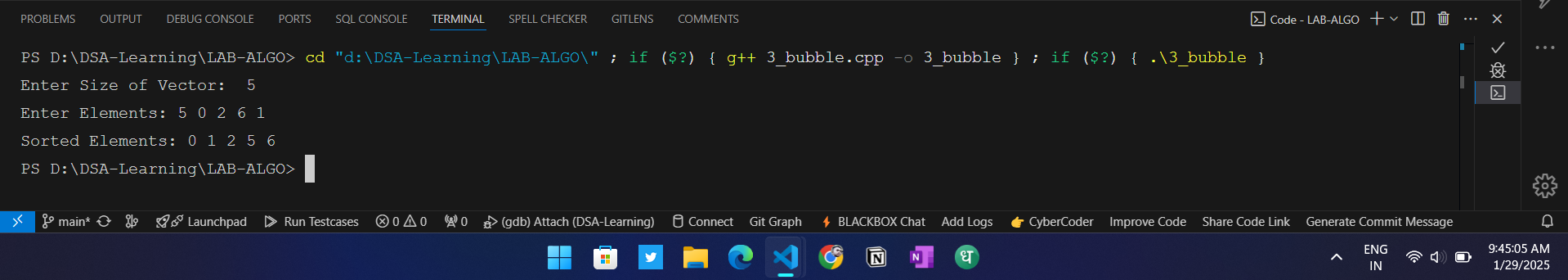
for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

cout << endl;

return 0;

}

Time Complexity O(n^2)

Space Complexity O(1)

Stable YES

In-place YES

1. Insertation Sort

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

for (int i = 0; i < n; i++) {

cin >> arr[i];

}

// Insertion Sort

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

// Print Sorted Values

cout << "Sorted Elements: ";

for (int i = 0; i < n; i++) {

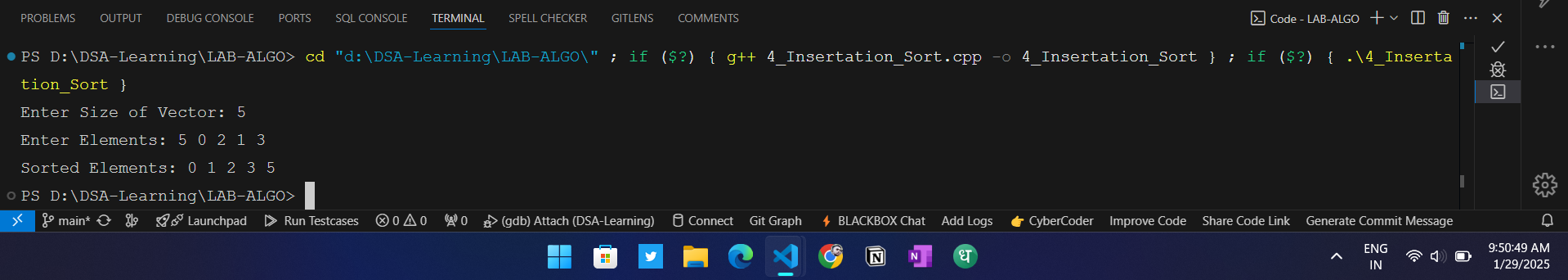
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



Time Complexity O(n^2)

Space Complexity O(1)

Stable YES

In-place YES

1. Selection Sort

#include <iostream>

#include <vector>

using namespace std;

int main() {

int n;

cout << "Enter Size of Vector: ";

cin >> n;

vector<int> arr(n);

// Element Entry

cout << "Enter Elements: ";

for (int i = 0; i < n; i++) {

cin >> arr[i];

}

// Selection Sort

for (int i = 0; i < n - 1; i++) {

int minIndex = i;

for (int j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

swap(arr[i], arr[minIndex]);

}

// Print Sorted Values

cout << "Sorted Elements: ";

for (int i = 0; i < n; i++) {

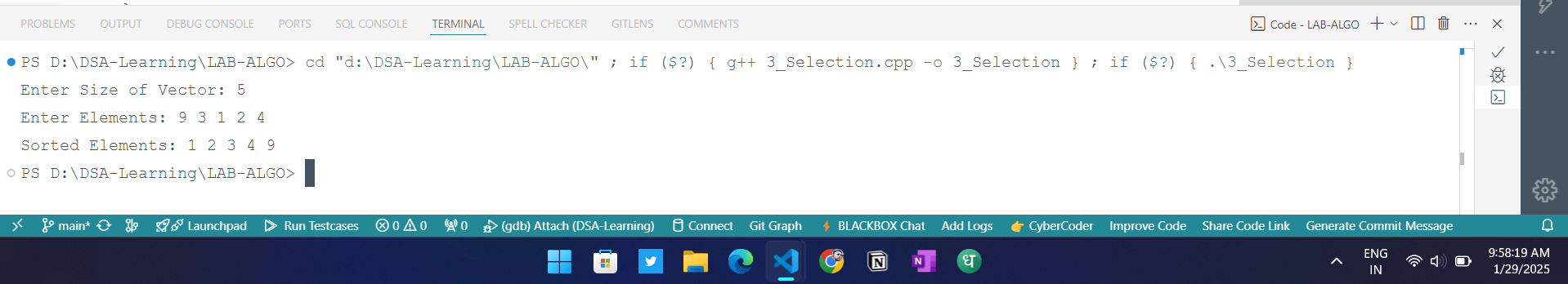
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



Time Complexity O(n^2)

Space Complexity O(1)

Stable NO

In-place YES

6 Quick Sort

#include<iostream>

using namespace std;

// Partition

int partition(int arr[], int start, int end){

int pos = start;

for(int i=start; i<=end; i++){

if(arr[i] <= arr[end]){

swap(arr[i] , arr[pos]);

pos++;

}

}

return pos - 1;

}

// Quick Sort

void quickSort(int arr[], int start, int end){

if(start >= end)

return;

int pivot = partition(arr, start, end);

// Left Side

quickSort(arr, start, pivot - 1);

// Right Side

quickSort(arr, pivot, end);

}

int main(){

int arr[] = {10,3,4,1,5,6,3,2,11,9};

quickSort(arr, 0, 9);

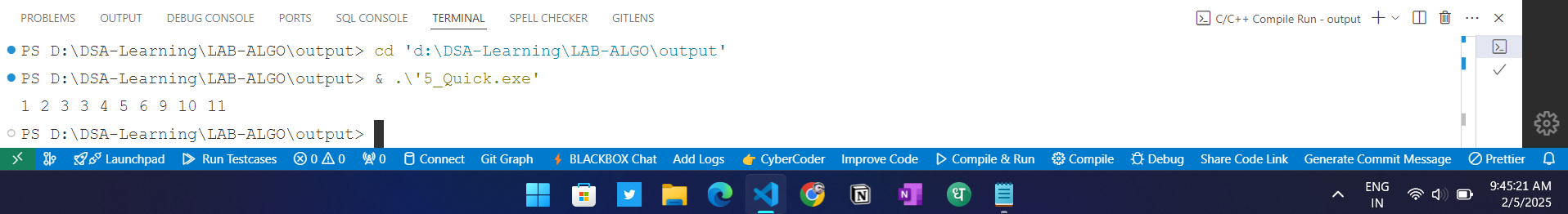
for(int i=0;i<10;i++){

cout << arr[i] << " ";

}

return 0;

}



**Time Complexity**

* **Best Case**: *O*(*n*log*n*)
* **Worst Case**: O(n2)
* **Average Case**:  *O*(*n*log*n*)

**Space Complexity**

* **Space Complexity**: *O*(log*n*) (due to recursive stack space)

Stability

* **Stable**: No (does not preserve the relative order of equal elements)
* **In-Place**: Yes (requires a constant amount of additional space)

Why is Quick Sort Not Stable?

Quick Sort is not stable because it can swap elements that are equal during the partitioning process. For example, if two equal elements are in different partitions, their relative order may change after the sorting process.

1. Merge Sort

#include <iostream>

#include <vector>

using namespace std;

void Merge(int arr[], int start, int mid, int end) {

vector<int> temp(end - start + 1);

int left = start, right = mid + 1, index = 0;

// Merging the two halves

while (left <= mid && right <= end) {

if (arr[left] <= arr[right]) {

temp[index++] = arr[left++];

} else {

temp[index++] = arr[right++];

}

}

while (left <= mid) {

temp[index++] = arr[left++];

}

while (right <= end) {

temp[index++] = arr[right++];

}

for (int i = 0; i < temp.size(); i++) {

arr[start + i] = temp[i];

}

}

// Merge Sort function

void MergeSort(int arr[], int start, int end) {

if (start < end) { // Change condition to allow sorting

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

MergeSort(arr, start, mid); // Sort left half

MergeSort(arr, mid + 1, end); // Sort right half

Merge(arr, start, mid, end); // Merge both halves

}

}

int main() {

int arr[] = {6, 3, 1, 2, 8, 9, 10, 7, 3, 10};

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

MergeSort(arr, 0, n - 1);

cout << "Sorted array: ";

for (int i = 0; i < n; i++) {

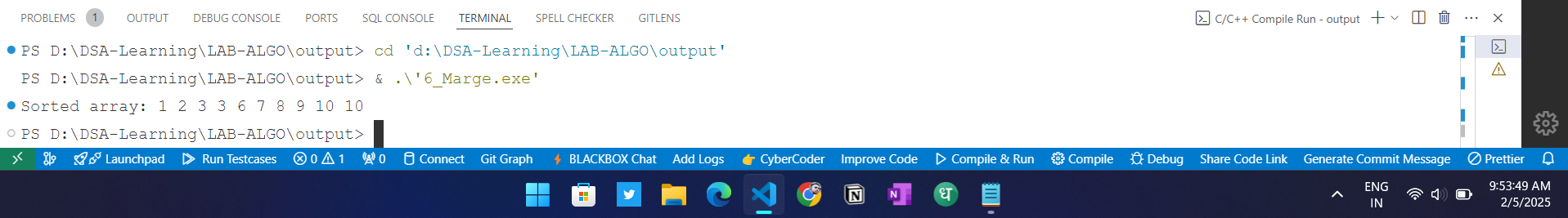
cout << arr[i] << " ";

}

cout << endl;

return 0;

}



**Time Complexity**

**Best Case**: *O*(*n*log*n*)

**Worst Case**: *O*(*n*log*n*)

**Average Case**: *O*(*n*log*n*)

**Space Complexity**

**Space Complexity**: *O*(1) (in-place sorting, no additional array needed)

**Stability**

**Stable**: No (does not preserve the relative order of equal elements)

**In-Place**: Yes (does not require additional storage for sorting)

**Why is Heap Sort Not Stable?**

Heap Sort is not stable because it may change the relative order of equal elements when they are moved around in the heap structure. During the Heapification process, equal elements can be swapped, thus disrupting their original order. These sections summarize the characteristics of Quick Sort and Heap Sort, highlighting their time and space complexities, stability, and reasons for stability or instability.

1. Heap Sort

#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i) {

int largest = i;

int left = 2 \* i + 1;

int right = 2 \* i + 2;

// If left child is larger than root

if (left < n && arr[left] > arr[largest]) {

largest = left;

}

// If right child is larger than largest so far

if (right < n && arr[right] > arr[largest]) {

largest = right;

}

// If largest is not root

if (largest != i) {

swap(arr[i], arr[largest]);

heapify(arr, n, largest);

}

}

void heapSort(int arr[], int n) {

for (int i = n / 2 - 1; i >= 0; i--) {

heapify(arr, n, i);

}

// One by one extract elements from heap

for (int i = n - 1; i >= 0; i--) {

swap(arr[0], arr[i]);

// Call max heapify on the reduced heap

heapify(arr, i, 0);

}

}

void printArray(int arr[], int n) {

for (int i = 0; i < n; i++) {

cout << arr[i] << " ";

}

cout << endl;

}

int main() {

int arr[] = {12, 11, 13, 5, 6, 7};

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

cout << "Original array: ";

printArray(arr, n);

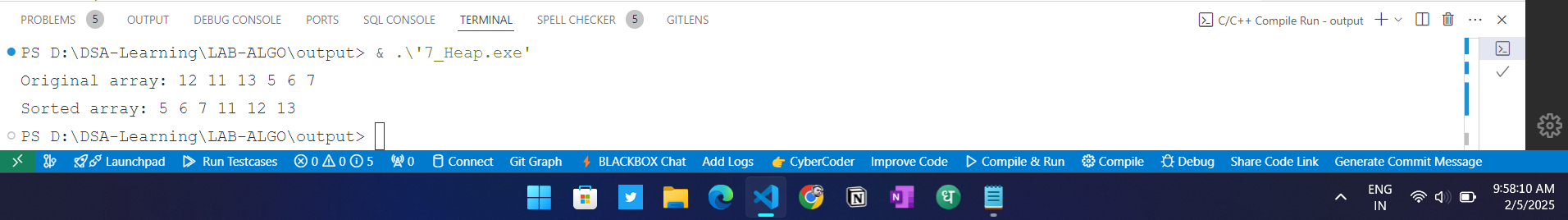
heapSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}



8 AVL Tree

#include <iostream>

using namespace std;

// Node structure for AVL Tree

struct Node {

int key;

Node\* left;

Node\* right;

int height;

Node(int value) {

key = value;

left = right = nullptr;

height = 1;

}

};

// Get the height of a node

int getHeight(Node\* node) {

return node ? node->height : 0;

}

// Get the balance factor of a node

int getBalanceFactor(Node\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

// Update the height of a node

void updateHeight(Node\* node) {

if (node)

node->height = 1 + max(getHeight(node->left), getHeight(node->right));

}

// Right rotate (LL Rotation)

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

// Rotation

x->right = y;

y->left = T2;

// Update heights

updateHeight(y);

updateHeight(x);

return x;

}

// Left rotate (RR Rotation)

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

// Rotation

y->left = x;

x->right = T2;

// Update heights

updateHeight(x);

updateHeight(y);

return y;

}

// Balance the tree after insertion or deletion

Node\* balance(Node\* node) {

int balanceFactor = getBalanceFactor(node);

// Left Heavy (LL case)

if (balanceFactor > 1 && getBalanceFactor(node->left) >= 0)

return rightRotate(node);

// Left-Right Case (LR case)

if (balanceFactor > 1 && getBalanceFactor(node->left) < 0) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

// Right Heavy (RR case)

if (balanceFactor < -1 && getBalanceFactor(node->right) <= 0)

return leftRotate(node);

// Right-Left Case (RL case)

if (balanceFactor < -1 && getBalanceFactor(node->right) > 0) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node; // Balanced

}

// Insert a node in AVL Tree

Node\* insert(Node\* root, int key) {

if (!root) return new Node(key);

if (key < root->key)

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

else if (key > root->key)

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

else

return root; // No duplicate keys

updateHeight(root);

return balance(root);

}

// Find the node with the smallest value

Node\* getMinNode(Node\* node) {

while (node->left)

node = node->left;

return node;

}

// Delete a node from AVL Tree

Node\* deleteNode(Node\* root, int key) {

if (!root) return nullptr;

// Perform standard BST delete

if (key < root->key)

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

else if (key > root->key)

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

else {

if (!root->left || !root->right) { // One child or no child

Node\* temp = root->left ? root->left : root->right;

delete root;

return temp;

}

else { // Two children

Node\* temp = getMinNode(root->right);

root->key = temp->key;

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

}

}

updateHeight(root);

return balance(root);

}

// Search for a key in AVL Tree

bool search(Node\* root, int key) {

if (!root) return false;

if (root->key == key) return true;

if (key < root->key) return search(root->left, key);

return search(root->right, key);

}

// Inorder Traversal (Sorted Order)

void inorder(Node\* root) {

if (!root) return;

inorder(root->left);

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

inorder(root->right);

}

// Driver Code

int main() {

Node\* root = nullptr;

// Insert nodes

root = insert(root, 60);

root = insert(root, 40);

root = insert(root, 80);

root = insert(root, 20);

root = insert(root, 55);

root = insert(root, 75);

root = insert(root, 95);

cout << "Inorder traversal: ";

inorder(root);

cout << endl;

// Delete node

root = deleteNode(root, 40);

cout << "After deleting 40: ";

inorder(root);

cout << endl;

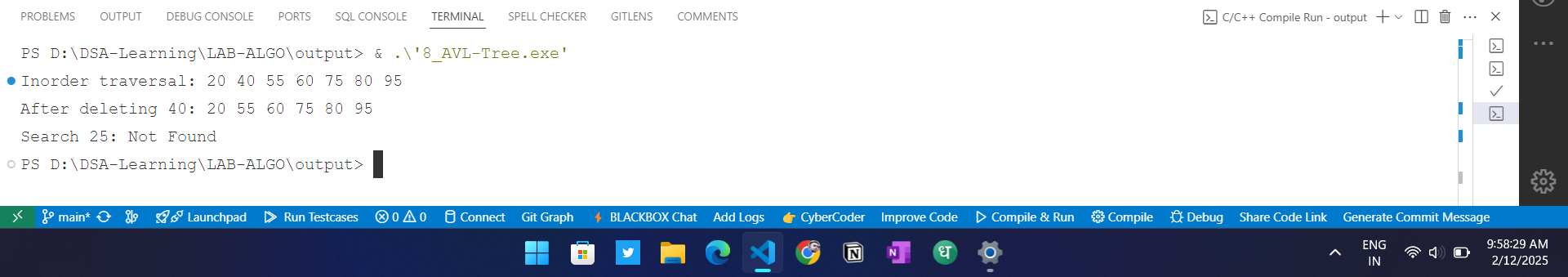
// Search for a value

int searchKey = 55;

cout << "Search " << searchKey << ": " << (search(root, searchKey) ? "Found" : "Not Found") << endl;

return 0;

}



**Time Complexity**

**Space Complexity**

1. BST

#include <iostream>

using namespace std;

struct Node {

int key;

Node\* left;

Node\* right;

Node(int value) {

key = value;

left = right = nullptr;

}

};

// Insert a node in BST

Node\* insert(Node\* root, int key) {

if (!root) return new Node(key);

if (key < root->key)

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

else if (key > root->key)

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

return root;

}

// Inorder Traversal (Left, Root, Right)

void inorder(Node\* root) {

if (!root) return;

inorder(root->left);

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

inorder(root->right);

}

// Preorder Traversal (Root, Left, Right)

void preorder(Node\* root) {

if (!root) return;

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

preorder(root->left);

preorder(root->right);

}

// Postorder Traversal (Left, Right, Root)

void postorder(Node\* root) {

if (!root) return;

postorder(root->left);

postorder(root->right);

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

}

int main() {

Node\* root = nullptr;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 70);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 60);

root = insert(root, 80);

cout << "Inorder Traversal: ";

inorder(root);

cout << endl;

cout << "Preorder Traversal: ";

preorder(root);

cout << endl;

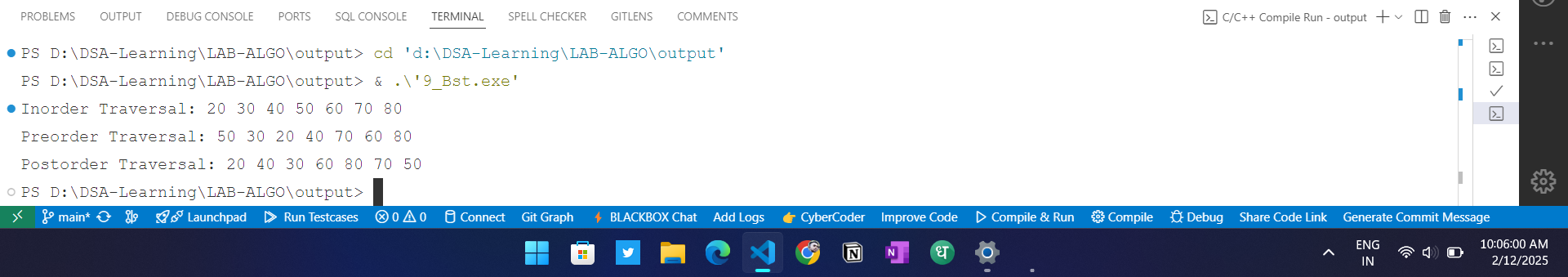
cout << "Postorder Traversal: ";

postorder(root);

cout << endl;

return 0;

}



**Time Complexity**

**Space Complexity**