**STACK BASED IMPLEMENTATIONS**

#include <iostream>

#include <stack>

#include <unordered\_map>

#include <unordered\_set>

using namespace std;

// 1. Implement Queue using Stack (Two Stacks - Enqueue Costly)

class QueueUsingStack {

    stack<int> s1, s2;

public:

    void enqueue(int x) {

        while (!s1.empty()) {

            s2.push(s1.top());

            s1.pop();

        }

        s1.push(x);

        while (!s2.empty()) {

            s1.push(s2.top());

            s2.pop();

        }

    }

    int dequeue() {

        if (s1.empty()) return -1;

        int x = s1.top();

        s1.pop();

        return x;

    }

};

// 2. Implement Deque using Stack

class DequeUsingStack {

    stack<int> frontStack, backStack;

public:

    void pushFront(int x) { frontStack.push(x); }

    void pushBack(int x) { backStack.push(x); }

    int popFront() {

        if (frontStack.empty()) return backStack.empty() ? -1 : backStack.top();

        int x = frontStack.top();

        frontStack.pop();

        return x;

    }

    int popBack() {

        if (backStack.empty()) return frontStack.empty() ? -1 : frontStack.top();

        int x = backStack.top();

        backStack.pop();

        return x;

    }

};

// 3. Implement Min Stack using Two Stacks

class MinStack {

    stack<int> s, minStack;

public:

    void push(int x) {

        s.push(x);

        if (minStack.empty() || x <= minStack.top()) minStack.push(x);

    }

    void pop() {

        if (s.top() == minStack.top()) minStack.pop();

        s.pop();

    }

    int getMin() { return minStack.top(); }

};

// 4. Implement Max Stack using Two Stacks

class MaxStack {

    stack<int> s, maxStack;

public:

    void push(int x) {

        s.push(x);

        if (maxStack.empty() || x >= maxStack.top()) maxStack.push(x);

    }

    void pop() {

        if (s.top() == maxStack.top()) maxStack.pop();

        s.pop();

    }

    int getMax() { return maxStack.top(); }

};

// 5. Implement Priority Queue using Stack

class PriorityQueueUsingStack {

    stack<int> s;

public:

    void push(int x) {

        stack<int> temp;

        while (!s.empty() && s.top() > x) {

            temp.push(s.top());

            s.pop();

        }

        s.push(x);

        while (!temp.empty()) {

            s.push(temp.top());

            temp.pop();

        }

    }

    int pop() {

        if (s.empty()) return -1;

        int x = s.top();

        s.pop();

        return x;

    }

};

// 6. Implement BST (Inorder Traversal) using Stack (Iterative DFS)

struct TreeNode {

    int val;

    TreeNode \*left, \*right;

    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

void inorderTraversal(TreeNode\* root) {

    stack<TreeNode\*> s;

    TreeNode\* curr = root;

    while (curr || !s.empty()) {

        while (curr) {

            s.push(curr);

            curr = curr->left;

        }

        curr = s.top();

        s.pop();

        cout << curr->val << " ";

        curr = curr->right;

    }

}

// 7. Implement Graph DFS using Stack (Iterative DFS)

void graphDFS(int start, unordered\_map<int, vector<int>>& adj) {

    stack<int> s;

    unordered\_set<int> visited;

    s.push(start);

    while (!s.empty()) {

        int node = s.top();

        s.pop();

        if (!visited.count(node)) {

            cout << node << " ";

            visited.insert(node);

            for (auto it = adj[node].rbegin(); it != adj[node].rend(); ++it) {

                if (!visited.count(\*it)) s.push(\*it);

            }

        }

    }

}

// Main Function to test the implementations

int main() {

    // 1. Queue Using Stack

    QueueUsingStack q;

    q.enqueue(1);

    q.enqueue(2);

    cout << "Queue dequeue: " << q.dequeue() << endl;

    // 2. Deque Using Stack

    DequeUsingStack dq;

    dq.pushFront(1);

    dq.pushBack(2);

    cout << "Deque popFront: " << dq.popFront() << endl;

    // 3. Min Stack

    MinStack minStack;

    minStack.push(3);

    minStack.push(1);

    cout << "Min Stack min: " << minStack.getMin() << endl;

    // 4. Max Stack

    MaxStack maxStack;

    maxStack.push(3);

    maxStack.push(5);

    cout << "Max Stack max: " << maxStack.getMax() << endl;

    // 5. Priority Queue Using Stack

    PriorityQueueUsingStack pq;

    pq.push(3);

    pq.push(1);

    cout << "Priority Queue pop: " << pq.pop() << endl;

    // 6. BST Inorder Traversal

    TreeNode\* root = new TreeNode(4);

    root->left = new TreeNode(2);

    root->right = new TreeNode(6);

    cout << "BST Inorder Traversal: ";

    inorderTraversal(root);

    cout << endl;

    // 7. Graph DFS Using Stack

    unordered\_map<int, vector<int>> adj = {

        {0, {1, 2}}, {1, {3, 4}}, {2, {5, 6}}, {3, {}}, {4, {}}, {5, {}}, {6, {}}

    };

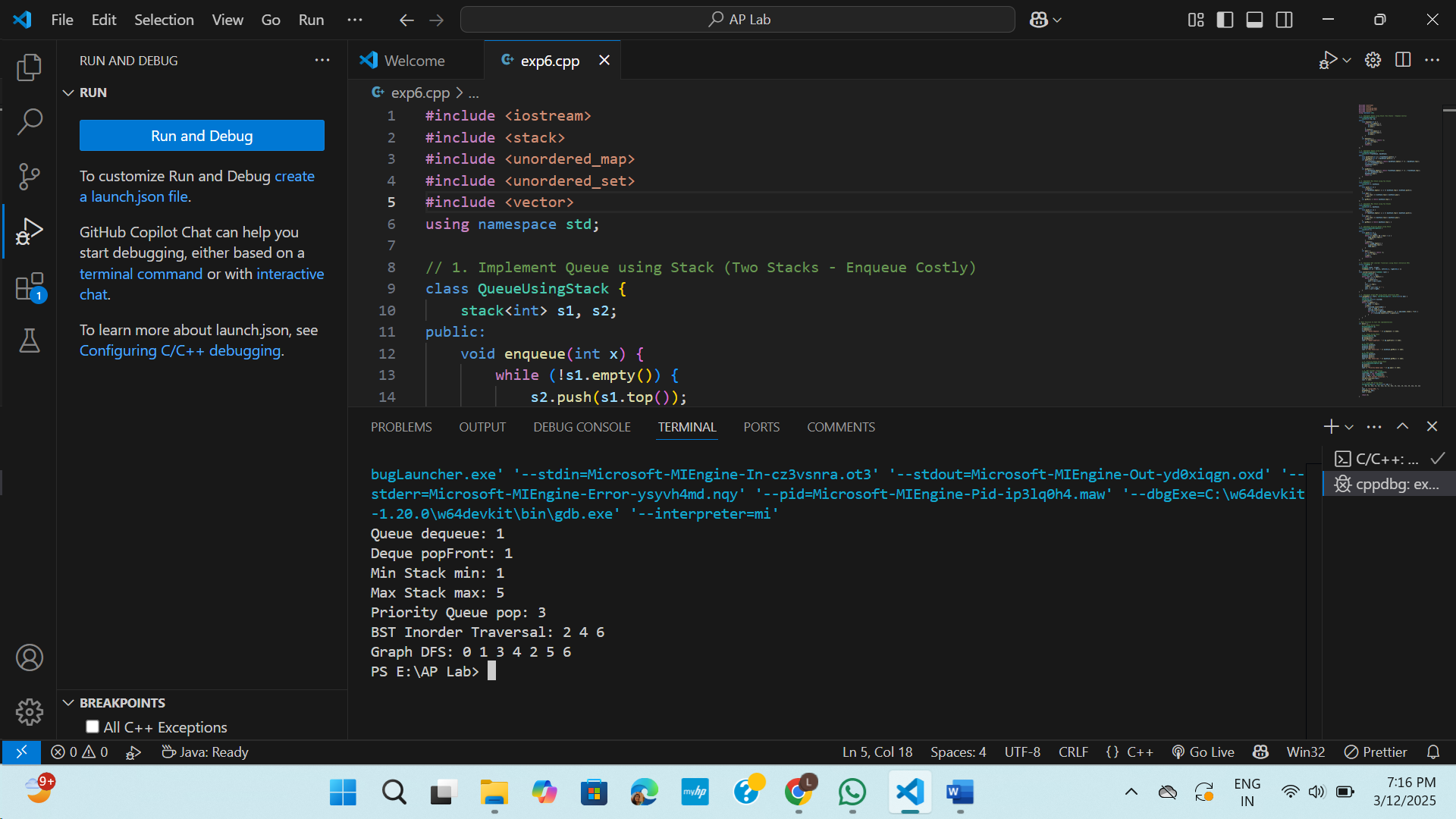
    cout << "Graph DFS: ";

    graphDFS(0, adj);

    cout << endl;

    return 0;

}



**QUEUE BASED IMPLEMENTATION**

#include <iostream>

#include <queue>

#include <stack>

#include <deque>

#include <vector>

#include <unordered\_map>

#include <unordered\_set>

using namespace std;

// 8. Implement Stack using Queue (Two Queues - Push Costly)

class StackUsingQueue {

    queue<int> q1, q2;

public:

    void push(int x) {

        q2.push(x);

        while (!q1.empty()) {

            q2.push(q1.front());

            q1.pop();

        }

        swap(q1, q2);

    }

    int pop() {

        if (q1.empty()) return -1;

        int x = q1.front();

        q1.pop();

        return x;

    }

    int top() { return q1.empty() ? -1 : q1.front(); }

};

// 9. Implement Deque using Queue

class DequeUsingQueue {

    deque<int> dq;

public:

    void pushFront(int x) { dq.push\_front(x); }

    void pushBack(int x) { dq.push\_back(x); }

    int popFront() {

        if (dq.empty()) return -1;

        int x = dq.front();

        dq.pop\_front();

        return x;

    }

    int popBack() {

        if (dq.empty()) return -1;

        int x = dq.back();

        dq.pop\_back();

        return x;

    }

};

// 10. Implement Circular Queue using Queue

class CircularQueue {

    vector<int> q;

    int front, rear, size, capacity;

public:

    CircularQueue(int k) : q(k), front(-1), rear(-1), size(0), capacity(k) {}

    bool enqueue(int x) {

        if (size == capacity) return false;

        if (front == -1) front = 0;

        rear = (rear + 1) % capacity;

        q[rear] = x;

        size++;

        return true;

    }

    bool dequeue() {

        if (size == 0) return false;

        if (front == rear) front = rear = -1;

        else front = (front + 1) % capacity;

        size--;

        return true;

    }

    int getFront() { return size == 0 ? -1 : q[front]; }

    int getRear() { return size == 0 ? -1 : q[rear]; }

};

// 11. Implement BST Level Order Traversal using Queue (BFS)

struct TreeNode {

    int val;

    TreeNode \*left, \*right;

    TreeNode(int x) : val(x), left(NULL), right(NULL) {}

};

void levelOrderTraversal(TreeNode\* root) {

    if (!root) return;

    queue<TreeNode\*> q;

    q.push(root);

    while (!q.empty()) {

        TreeNode\* curr = q.front();

        q.pop();

        cout << curr->val << " ";

        if (curr->left) q.push(curr->left);

        if (curr->right) q.push(curr->right);

    }

}

// 12. Implement Graph BFS using Queue (Iterative BFS)

void graphBFS(int start, unordered\_map<int, vector<int>>& adj) {

    queue<int> q;

    unordered\_set<int> visited;

    q.push(start);

    visited.insert(start);

    while (!q.empty()) {

        int node = q.front();

        q.pop();

        cout << node << " ";

        for (int neighbor : adj[node]) {

            if (!visited.count(neighbor)) {

                q.push(neighbor);

                visited.insert(neighbor);

            }

        }

    }

}

// Driver Code

int main() {

    // Testing Stack Using Queue

    StackUsingQueue stack;

    stack.push(1);

    stack.push(2);

    stack.push(3);

    cout << "Stack Top: " << stack.top() << endl;

    stack.pop();

    cout << "Stack Top after pop: " << stack.top() << endl;

    // Testing Deque Using Queue

    DequeUsingQueue dq;

    dq.pushFront(10);

    dq.pushBack(20);

    cout << "Deque Front: " << dq.popFront() << endl;

    cout << "Deque Back: " << dq.popBack() << endl;

    // Testing Circular Queue

    CircularQueue cq(3);

    cq.enqueue(1);

    cq.enqueue(2);

    cq.enqueue(3);

    cout << "Circular Queue Front: " << cq.getFront() << endl;

    cq.dequeue();

    cout << "Circular Queue Front after Dequeue: " << cq.getFront() << endl;

    // Testing BST Level Order Traversal

    TreeNode\* root = new TreeNode(5);

    root->left = new TreeNode(3);

    root->right = new TreeNode(7);

    cout << "BST Level Order Traversal: ";

    levelOrderTraversal(root);

    cout << endl;

    // Testing Graph BFS

    unordered\_map<int, vector<int>> adj;

    adj[1] = {2, 3};

    adj[2] = {4, 5};

    adj[3] = {6};

    adj[4] = {};

    adj[5] = {};

    adj[6] = {};

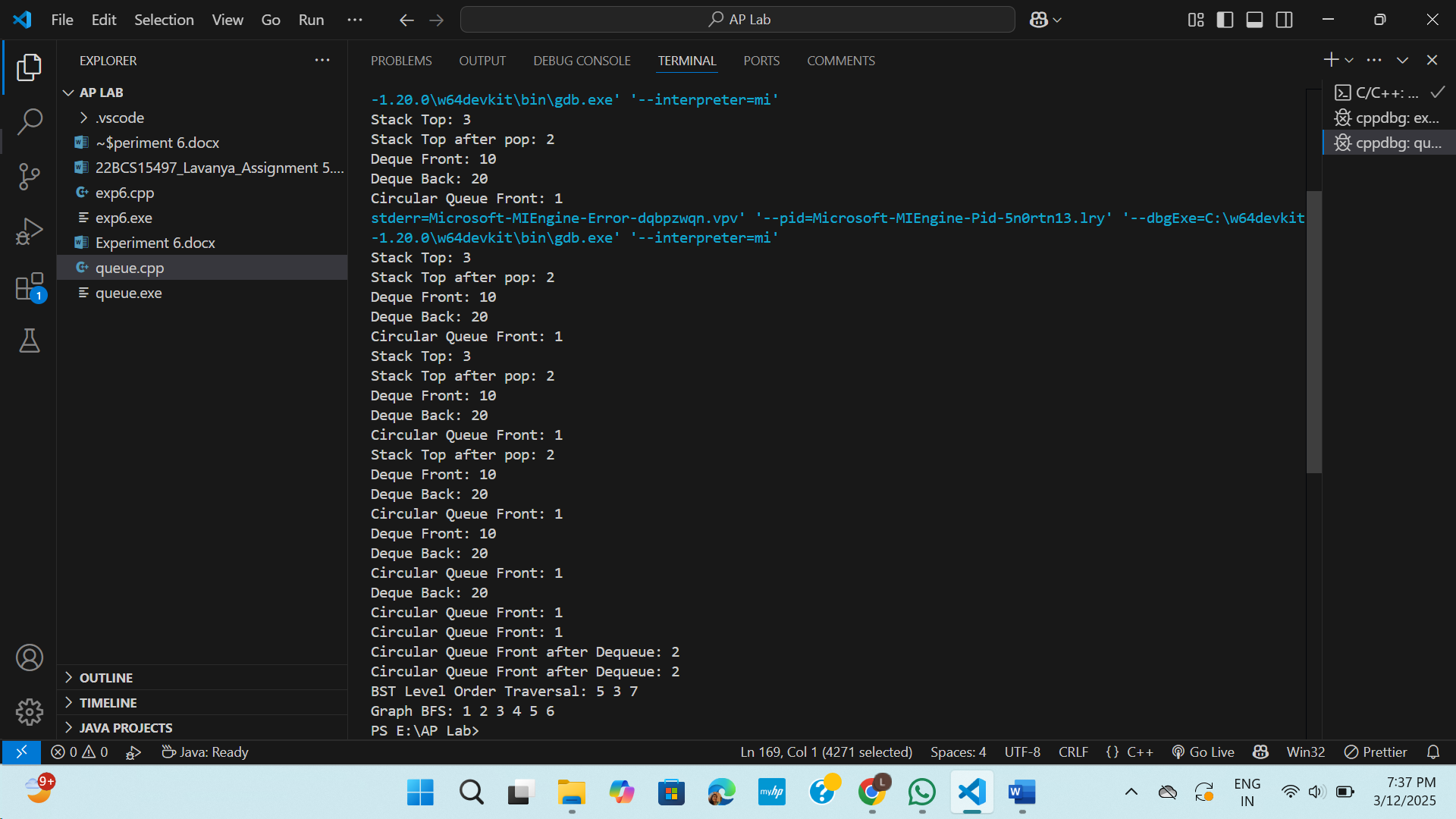
    cout << "Graph BFS: ";

    graphBFS(1, adj);

    cout << endl;

    return 0;

}



**ARRAY BASED IMPLEMENTATIONS**

#include <iostream>

#include <climits>

#include <vector>

#include <list>

#include <cstring>

using namespace std;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 13. Implement Stack using an Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class StackArray {

    int top, capacity;

    int \*arr;

public:

    StackArray(int size) {

        capacity = size;

        arr = new int[size];

        top = -1;

    }

    void push(int x) {

        if (top == capacity - 1) {

            cout << "Stack Overflow\n";

            return;

        }

        arr[++top] = x;

    }

    int pop() {

        if (top == -1) return -1;

        return arr[top--];

    }

    int peek() {

        if (top == -1) return -1;

        return arr[top];

    }

    void display() {

        if (top == -1) cout << "Stack is Empty\n";

        else {

            cout << "Stack Elements: ";

            for (int i = top; i >= 0; i--) cout << arr[i] << " ";

            cout << endl;

        }

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 14. Implement Queue using an Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class QueueArray {

    int front, rear, capacity;

    int \*arr;

public:

    QueueArray(int size) {

        capacity = size;

        arr = new int[size];

        front = rear = -1;

    }

    void enqueue(int x) {

        if (rear == capacity - 1) {

            cout << "Queue Overflow\n";

            return;

        }

        arr[++rear] = x;

        if (front == -1) front = 0;

    }

    int dequeue() {

        if (front == -1 || front > rear) return -1;

        return arr[front++];

    }

    void display() {

        if (front == -1 || front > rear) cout << "Queue is Empty\n";

        else {

            cout << "Queue Elements: ";

            for (int i = front; i <= rear; i++) cout << arr[i] << " ";

            cout << endl;

        }

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 20. Implement Min Heap using an Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class MinHeap {

    vector<int> heap;

public:

    void insert(int x) {

        heap.push\_back(x);

        int i = heap.size() - 1;

        while (i > 0 && heap[(i - 1) / 2] > heap[i]) {

            swap(heap[i], heap[(i - 1) / 2]);

            i = (i - 1) / 2;

        }

    }

    void display() {

        cout << "Min Heap: ";

        for (int i : heap) cout << i << " ";

        cout << endl;

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 21. Implement Max Heap using an Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class MaxHeap {

    vector<int> heap;

public:

    void insert(int x) {

        heap.push\_back(x);

        int i = heap.size() - 1;

        while (i > 0 && heap[(i - 1) / 2] < heap[i]) {

            swap(heap[i], heap[(i - 1) / 2]);

            i = (i - 1) / 2;

        }

    }

    void display() {

        cout << "Max Heap: ";

        for (int i : heap) cout << i << " ";

        cout << endl;

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 24. Implement Graph using Adjacency Matrix \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Graph {

    int adjMatrix[5][5];

public:

    Graph() { memset(adjMatrix, 0, sizeof(adjMatrix)); }

    void addEdge(int u, int v) { adjMatrix[u][v] = 1; }

    void display() {

        cout << "Adjacency Matrix:\n";

        for (int i = 0; i < 5; i++) {

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

                cout << adjMatrix[i][j] << " ";

            cout << endl;

        }

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAIN FUNCTION WITH OUTPUTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

    cout << "================ Stack using Array ================\n";

    StackArray stack(5);

    stack.push(10);

    stack.push(20);

    stack.push(30);

    stack.display();

    cout << "Popped Element: " << stack.pop() << endl;

    stack.display();

    cout << "\n================ Queue using Array ================\n";

    QueueArray queue(5);

    queue.enqueue(10);

    queue.enqueue(20);

    queue.enqueue(30);

    queue.display();

    cout << "Dequeued Element: " << queue.dequeue() << endl;

    queue.display();

    cout << "\n================ Min Heap using Array ================\n";

    MinHeap minHeap;

    minHeap.insert(40);

    minHeap.insert(10);

    minHeap.insert(30);

    minHeap.insert(5);

    minHeap.display();

    cout << "\n================ Max Heap using Array ================\n";

    MaxHeap maxHeap;

    maxHeap.insert(20);

    maxHeap.insert(50);

    maxHeap.insert(10);

    maxHeap.insert(70);

    maxHeap.display();

    cout << "\n================ Graph using Adjacency Matrix ================\n";

    Graph graph;

    graph.addEdge(0, 1);

    graph.addEdge(0, 2);

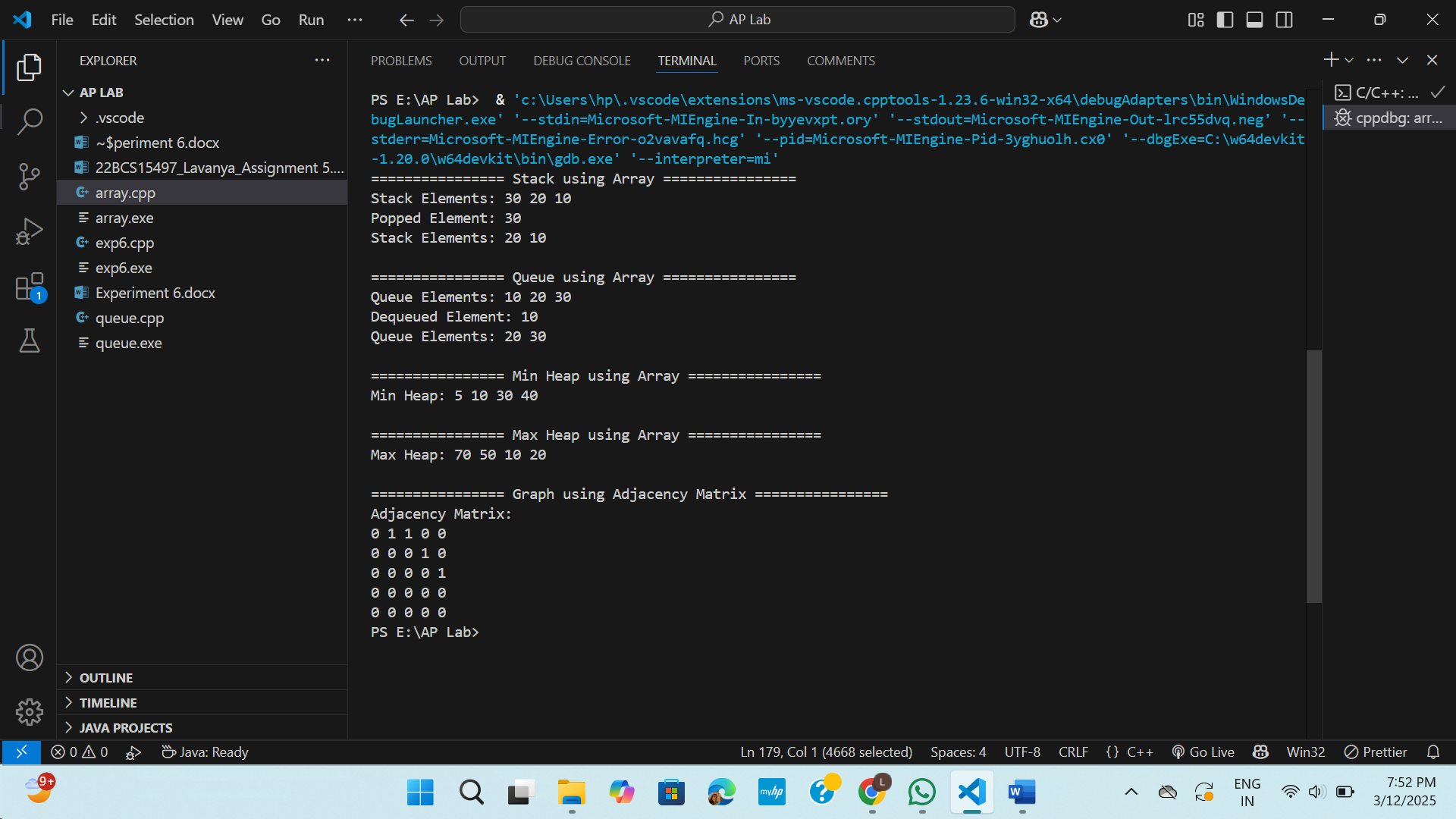
    graph.addEdge(1, 3);

    graph.addEdge(2, 4);

    graph.display();

    return 0;

}



**LINKED LIST BASED IMPLEMENTATION**

#include <iostream>

#include <climits>

#include <vector>

#include <list>

#include <unordered\_map>

using namespace std;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 25. Implement Stack using Linked List \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class StackLL {

struct Node {

int data;

Node\* next;

Node(int x) : data(x), next(nullptr) {}

};

Node\* top;

public:

StackLL() { top = nullptr; }

void push(int x) {

Node\* newNode = new Node(x);

newNode->next = top;

top = newNode;

}

int pop() {

if (!top) return -1;

int val = top->data;

Node\* temp = top;

top = top->next;

delete temp;

return val;

}

void display() {

Node\* temp = top;

cout << "Stack Elements: ";

while (temp) {

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

temp = temp->next;

}

cout << endl;

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 26. Implement Queue using Linked List \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class QueueLL {

struct Node {

int data;

Node\* next;

Node(int x) : data(x), next(nullptr) {}

};

Node \*front, \*rear;

public:

QueueLL() { front = rear = nullptr; }

void enqueue(int x) {

Node\* newNode = new Node(x);

if (!rear) front = rear = newNode;

else {

rear->next = newNode;

rear = newNode;

}

}

int dequeue() {

if (!front) return -1;

int val = front->data;

Node\* temp = front;

front = front->next;

if (!front) rear = nullptr;

delete temp;

return val;

}

void display() {

Node\* temp = front;

cout << "Queue Elements: ";

while (temp) {

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

temp = temp->next;

}

cout << endl;

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 30. Implement Hash Table using Linked List (Chaining) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class HashTable {

static const int SIZE = 10;

list<int> table[SIZE];

public:

int hashFunction(int key) { return key % SIZE; }

void insert(int key) {

int index = hashFunction(key);

table[index].push\_back(key);

}

bool search(int key) {

int index = hashFunction(key);

for (int val : table[index])

if (val == key) return true;

return false;

}

void display() {

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

cout << "Bucket " << i << ": ";

for (int val : table[i]) cout << val << " ";

cout << endl;

}

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 32. Implement Graph using Linked List (Adjacency List) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Graph {

unordered\_map<int, list<int>> adjList;

public:

void addEdge(int u, int v) {

adjList[u].push\_back(v);

adjList[v].push\_back(u); // For undirected graph

}

void display() {

for (auto &pair : adjList) {

cout << pair.first << " -> ";

for (int neighbor : pair.second) cout << neighbor << " ";

cout << endl;

}

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAIN FUNCTION WITH OUTPUTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

cout << "================ Stack using Linked List ================\n";

StackLL stack;

stack.push(10);

stack.push(20);

stack.push(30);

stack.display();

cout << "Popped Element: " << stack.pop() << endl;

stack.display();

cout << "\n================ Queue using Linked List ================\n";

QueueLL queue;

queue.enqueue(10);

queue.enqueue(20);

queue.enqueue(30);

queue.display();

cout << "Dequeued Element: " << queue.dequeue() << endl;

queue.display();

cout << "\n================ Hash Table using Linked List ================\n";

HashTable hashTable;

hashTable.insert(15);

hashTable.insert(25);

hashTable.insert(35);

hashTable.insert(45);

hashTable.display();

cout << "Search 25: " << (hashTable.search(25) ? "Found" : "Not Found") << endl;

cout << "\n================ Graph using Linked List (Adjacency List) ================\n";

Graph graph;

graph.addEdge(1, 2);

graph.addEdge(1, 3);

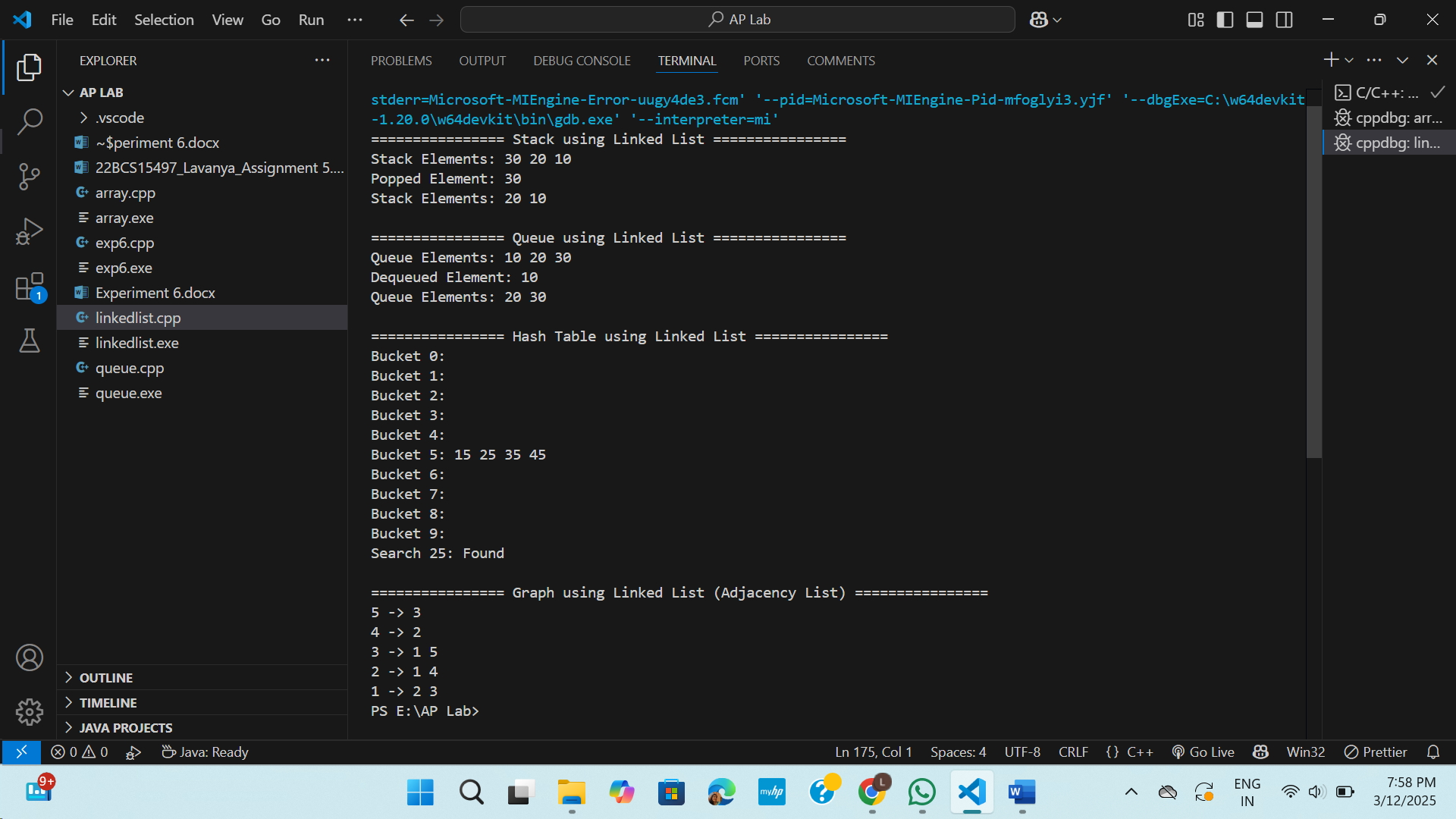
graph.addEdge(2, 4);

graph.addEdge(3, 5);

graph.display();

return 0;

}



**TREE BASED IMPLEMENTATIONS**

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <algorithm>

using namespace std;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 38. Implement BST using Linked List \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class BST {

struct Node {

int data;

Node\* left, \*right;

Node(int x) : data(x), left(nullptr), right(nullptr) {}

};

Node\* root;

Node\* insert(Node\* node, int data) {

if (!node) return new Node(data);

if (data < node->data) node->left = insert(node->left, data);

else node->right = insert(node->right, data);

return node;

}

void inorder(Node\* node) {

if (node) {

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

}

public:

BST() : root(nullptr) {}

void insert(int data) { root = insert(root, data); }

void display() {

cout << "Inorder Traversal: ";

inorder(root);

cout << endl;

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 39. Implement AVL Tree using BST \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class AVL {

struct Node {

int data, height;

Node\* left, \*right;

Node(int x) : data(x), height(1), left(nullptr), right(nullptr) {}

};

Node\* root;

int height(Node\* node) { return node ? node->height : 0; }

int getBalance(Node\* node) { return node ? height(node->left) - height(node->right) : 0; }

Node\* rotateRight(Node\* y) {

Node\* x = y->left;

Node\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

Node\* rotateLeft(Node\* x) {

Node\* y = x->right;

Node\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

Node\* insert(Node\* node, int data) {

if (!node) return new Node(data);

if (data < node->data) node->left = insert(node->left, data);

else node->right = insert(node->right, data);

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

int balance = getBalance(node);

if (balance > 1 && data < node->left->data) return rotateRight(node);

if (balance < -1 && data > node->right->data) return rotateLeft(node);

if (balance > 1 && data > node->left->data) {

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

return rotateRight(node);

}

if (balance < -1 && data < node->right->data) {

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

return rotateLeft(node);

}

return node;

}

void inorder(Node\* node) {

if (node) {

inorder(node->left);

cout << node->data << " ";

inorder(node->right);

}

}

public:

AVL() : root(nullptr) {}

void insert(int data) { root = insert(root, data); }

void display() {

cout << "AVL Inorder Traversal: ";

inorder(root);

cout << endl;

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 40. Implement Trie using HashMap \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class Trie {

struct Node {

unordered\_map<char, Node\*> children;

bool isEnd;

Node() : isEnd(false) {}

};

Node\* root;

public:

Trie() { root = new Node(); }

void insert(string word) {

Node\* node = root;

for (char ch : word) {

if (!node->children.count(ch))

node->children[ch] = new Node();

node = node->children[ch];

}

node->isEnd = true;

}

bool search(string word) {

Node\* node = root;

for (char ch : word) {

if (!node->children.count(ch)) return false;

node = node->children[ch];

}

return node->isEnd;

}

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 42. Implement Segment Tree using an Array \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class SegmentTree {

vector<int> tree;

int n;

void build(vector<int>& arr, int node, int start, int end) {

if (start == end) tree[node] = arr[start];

else {

int mid = (start + end) / 2;

build(arr, 2 \* node, start, mid);

build(arr, 2 \* node + 1, mid + 1, end);

tree[node] = tree[2 \* node] + tree[2 \* node + 1];

}

}

int sumQuery(int node, int start, int end, int L, int R) {

if (R < start || end < L) return 0;

if (L <= start && end <= R) return tree[node];

int mid = (start + end) / 2;

return sumQuery(2 \* node, start, mid, L, R) + sumQuery(2 \* node + 1, mid + 1, end, L, R);

}

public:

SegmentTree(vector<int>& arr) {

n = arr.size();

tree.resize(4 \* n);

build(arr, 1, 0, n - 1);

}

int query(int L, int R) { return sumQuery(1, 0, n - 1, L, R); }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAIN FUNCTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

cout << "================ BST using Linked List ================\n";

BST bst;

bst.insert(30);

bst.insert(20);

bst.insert(40);

bst.insert(10);

bst.display();

cout << "\n================ AVL Tree using BST ================\n";

AVL avl;

avl.insert(10);

avl.insert(20);

avl.insert(30);

avl.insert(40);

avl.insert(50);

avl.display();

cout << "\n================ Trie using HashMap ================\n";

Trie trie;

trie.insert("hello");

trie.insert("world");

cout << "Search 'hello': " << (trie.search("hello") ? "Found" : "Not Found") << endl;

cout << "Search 'data': " << (trie.search("data") ? "Found" : "Not Found") << endl;

cout << "\n================ Segment Tree using Array ================\n";

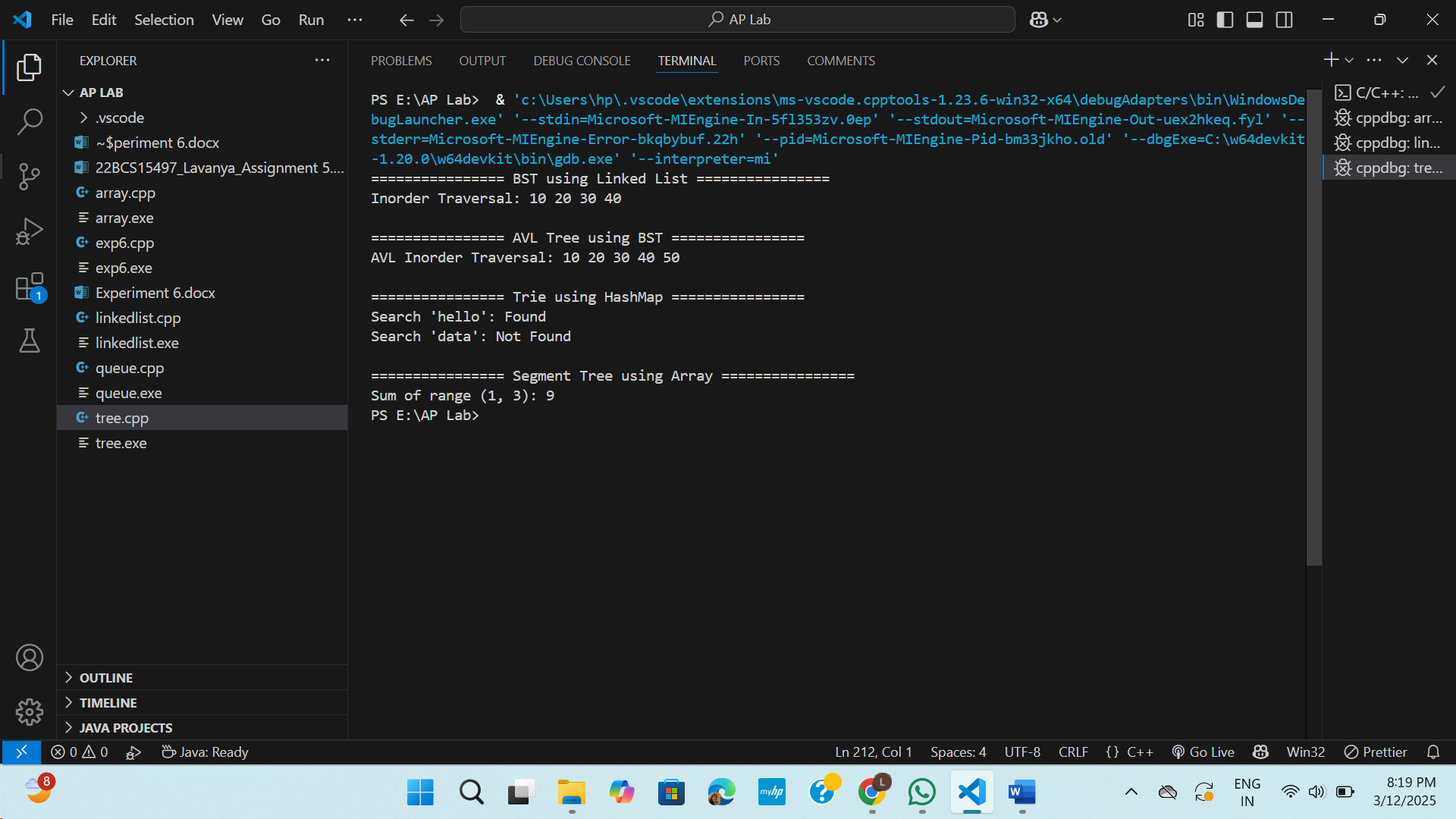
vector<int> arr = {1, 2, 3, 4, 5, 6};

SegmentTree segTree(arr);

cout << "Sum of range (1, 3): " << segTree.query(1, 3) << endl;

return 0;

}



**GRAPH BASED IMPLEMENTATION**

#include <iostream>

#include <vector>

#include <queue>

#include <stack>

#include <unordered\_map>

using namespace std;

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 49. Implement Graph using Adjacency List \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

class GraphAdjList {

public:  // FIX: Made adjList public

    unordered\_map<int, vector<int>> adjList;

    void addEdge(int u, int v) {

        adjList[u].push\_back(v);

        adjList[v].push\_back(u);

    }

    void display() {

        cout << "Graph (Adjacency List):\n";

        for (auto &pair : adjList) {

            cout << pair.first << " -> ";

            for (int v : pair.second)

                cout << v << " ";

            cout << endl;

        }

    }

};

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 51. Implement BFS using Graph + Queue \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void BFS(const unordered\_map<int, vector<int>>& adjList, int start) {

    queue<int> q;

    unordered\_map<int, bool> visited;

    q.push(start);

    visited[start] = true;

    cout << "BFS Traversal: ";

    while (!q.empty()) {

        int node = q.front();

        q.pop();

        cout << node << " ";

        for (int neighbor : adjList.at(node)) {

            if (!visited[neighbor]) {

                q.push(neighbor);

                visited[neighbor] = true;

            }

        }

    }

    cout << endl;

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 52. Implement DFS using Graph + Stack \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void DFS(const unordered\_map<int, vector<int>>& adjList, int start) {

    stack<int> s;

    unordered\_map<int, bool> visited;

    s.push(start);

    cout << "DFS Traversal: ";

    while (!s.empty()) {

        int node = s.top();

        s.pop();

        if (!visited[node]) {

            cout << node << " ";

            visited[node] = true;

        }

        for (int neighbor : adjList.at(node)) {

            if (!visited[neighbor]) {

                s.push(neighbor);

            }

        }

    }

    cout << endl;

}

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* MAIN FUNCTION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int main() {

    GraphAdjList graph;

    graph.addEdge(0, 1);

    graph.addEdge(1, 2);

    graph.addEdge(2, 3);

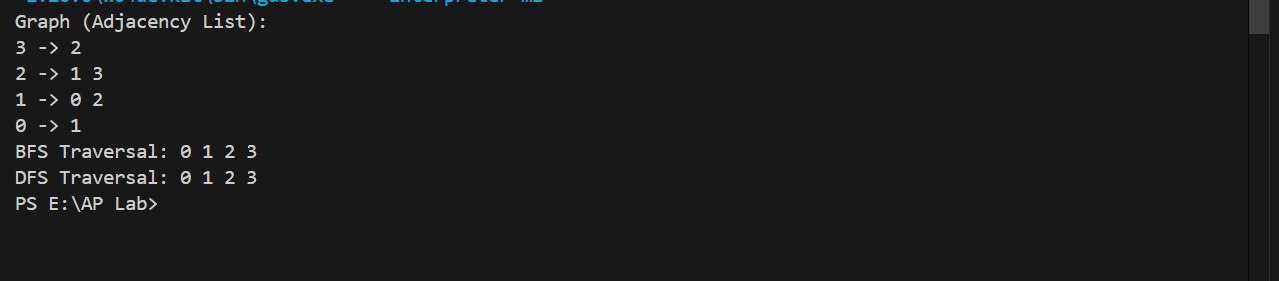
    graph.display();

    BFS(graph.adjList, 0);

    DFS(graph.adjList, 0);

    return 0;

}



**HEAP BASED IMPLEMENTATION**

#include <iostream>

#include <queue>

#include <vector>

using namespace std;

// ================== Priority Queue using Heap ==================

void priorityQueueExample() {

    priority\_queue<int> maxHeap;

    priority\_queue<int, vector<int>, greater<int>> minHeap;

    maxHeap.push(10);

    maxHeap.push(40);

    maxHeap.push(20);

    cout << "Priority Queue (Max Heap) - Top Element: " << maxHeap.top() << endl;

    minHeap.push(10);

    minHeap.push(40);

    minHeap.push(20);

    cout << "Priority Queue (Min Heap) - Top Element: " << minHeap.top() << endl;

}

// ==================  Min Heap using Max Heap ==================

class MinHeapUsingMaxHeap {

private:

    priority\_queue<int> maxHeap;

public:

    void push(int num) {

        maxHeap.push(-num);

    }

    int top() {

        return -maxHeap.top();

    }

    void pop() {

        maxHeap.pop();

    }

    bool empty() {

        return maxHeap.empty();

    }

};

void testMinHeapUsingMaxHeap() {

    MinHeapUsingMaxHeap minHeap;

    minHeap.push(10);

    minHeap.push(40);

    minHeap.push(20);

    cout << "Min Heap using Max Heap - Top Element: " << minHeap.top() << endl;

}

// ================== Max Heap using Min Heap ==================

class MaxHeapUsingMinHeap {

private:

    priority\_queue<int, vector<int>, greater<int>> minHeap;

public:

    void push(int num) {

        minHeap.push(-num);

    }

    int top() {

        return -minHeap.top();

    }

    void pop() {

        minHeap.pop();

    }

    bool empty() {

        return minHeap.empty();

    }

};

void testMaxHeapUsingMinHeap() {

    MaxHeapUsingMinHeap maxHeap;

    maxHeap.push(10);

    maxHeap.push(40);

    maxHeap.push(20);

    cout << "Max Heap using Min Heap - Top Element: " << maxHeap.top() << endl; // 40

}

// ==================  Median Finder using Two Heaps ==================

class MedianFinder {

private:

    priority\_queue<int> leftHalf;

    priority\_queue<int, vector<int>, greater<int>> rightHalf;

public:

    void addNum(int num) {

        if (leftHalf.empty() || num <= leftHalf.top()) {

            leftHalf.push(num);

        } else {

            rightHalf.push(num);

        }

        if (leftHalf.size() > rightHalf.size() + 1) {

            rightHalf.push(leftHalf.top());

            leftHalf.pop();

        } else if (rightHalf.size() > leftHalf.size()) {

            leftHalf.push(rightHalf.top());

            rightHalf.pop();

        }

    }

    double findMedian() {

        if (leftHalf.size() > rightHalf.size()) {

            return leftHalf.top();

        } else {

            return (leftHalf.top() + rightHalf.top()) / 2.0;

        }

    }

};

void testMedianFinder() {

    MedianFinder mf;

    mf.addNum(10);

    mf.addNum(20);

    cout << "Median: " << mf.findMedian() << endl;

    mf.addNum(30);

    cout << "Median: " << mf.findMedian() << endl;

}

// ==================  Kth Largest Element Finder using Heap ==================

class KthLargest {

private:

    priority\_queue<int, vector<int>, greater<int>> minHeap;

    int k;

public:

    KthLargest(int k, vector<int>& nums) {

        this->k = k;

        for (int num : nums) {

            add(num);

        }

    }

    int add(int num) {

        minHeap.push(num);

        if (minHeap.size() > k) {

            minHeap.pop();

        }

        return minHeap.top();

    }

};

void testKthLargest() {

    vector<int> nums = {10, 20, 30, 40, 50};

    KthLargest kthLargest(3, nums);

    cout << "3rd Largest Element: " << kthLargest.add(25) << endl;

}

// ================== Main Function to Run Tests ==================

int main() {

    cout << "=== Priority Queue ===" << endl;

    priorityQueueExample();

    cout << "\n=== Min Heap using Max Heap ===" << endl;

    testMinHeapUsingMaxHeap();

    cout << "\n=== Max Heap using Min Heap ===" << endl;

    testMaxHeapUsingMinHeap();

    cout << "\n=== Median Finder ===" << endl;

    testMedianFinder();

    cout << "\n=== Kth Largest Element Finder ===" << endl;

    testKthLargest();

    return 0;

}

