# Applied Programming



# Name: M. Hassan Sattar

# Roll No: 22F-3773

**Section: MS-1A**

# Submitted to: Sir. Muhammad Yousaf

**Question No. 1: Graphs (Robot Navigation System)**

#include <iostream>

#include <fstream>

#include <queue>

#include <stack>

using namespace std;

struct Point

{

int x, y;

};

int start[2] = { 0, 0 }, goal[2] = { 0, 0 };

void updatedMatric(int\*\* arr, int r, int c)

{

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

{

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

{

if (arr[i][j] == 9)

{

cout << "\* ";

}

else

{

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

}

}

cout << endl;

}

}

void copyArray(int\*\* to, int\*\* from, int r, int c)

{

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

{

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

{

to[i][j] = from[i][j];

}

}

}

void applyBFS(int\*\* arr, int r, int c)

{

int\*\* new\_arr;

new\_arr = new int\* [r];

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

{

new\_arr[i] = new int[c];

}

copyArray(new\_arr, arr, r, c);

queue<Point> q;

Point p;

int cost = 0;

q.push({ start[0], start[1] });

while (!q.empty())

{

p = q.front();

q.pop();

if (p.x == goal[0] && p.y == goal[1])

{

cout << "goal is found !!!" << endl;

break;

}

if (p.x - 1 >= 0 && !new\_arr[p.x - 1][p.y])

{

new\_arr[p.x - 1][p.y] = 9;

q.push({ p.x - 1, p.y });

cost += 2;

}

if ((p.x - 1) >= 0 && (p.y + 1) < c)

{

if (!new\_arr[p.x - 1][p.y + 1])

{

new\_arr[p.x - 1][p.y + 1] = 9;

q.push({ p.x - 1, p.y + 1 });

cost += 2;

}

}

if ((p.y + 1) < c && !new\_arr[p.x][p.y + 1])

{

new\_arr[p.x][p.y + 1] = 9;

q.push({ p.x, p.y + 1 });

cost += 2;

}

}

cout << "cost: " << cost << endl;

updatedMatric(new\_arr, r, c);

}

void applyDFS(int\*\* arr, int r, int c)

{

int\*\* new\_arr;

new\_arr = new int\* [r];

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

{

new\_arr[i] = new int[c];

}

copyArray(new\_arr, arr, r, c);

stack<Point> s;

Point p;

int cost = 0;

s.push({ start[0], start[1] });

while (!s.empty())

{

p = s.top();

s.pop();

if (p.x == goal[0] && p.y == goal[1])

{

cout << "goal is found !!!" << endl;

break;

}

if (p.x - 1 >= 0 && !new\_arr[p.x - 1][p.y])

{

new\_arr[p.x - 1][p.y] = 9;

s.push({ p.x - 1, p.y });

cost += 2;

}

if ((p.x - 1) >= 0 && (p.y + 1) < c)

{

if (!new\_arr[p.x - 1][p.y + 1])

{

new\_arr[p.x - 1][p.y + 1] = 9;

s.push({ p.x - 1, p.y + 1 });

cost += 2;

}

}

if ((p.y + 1) < c && !new\_arr[p.x][p.y + 1])

{

new\_arr[p.x][p.y + 1] = 9;

s.push({ p.x, p.y + 1 });

cost += 2;

}

}

cout << "cost: " << cost << endl;

updatedMatric(new\_arr, r, c);

}

bool fillArray(int\*\* arr, int r, int c);

int main()

{

int r = 15, c = 15;

int\*\* arr;

arr = new int\* [r];

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

{

arr[i] = new int[c];

}

if (!fillArray(arr, r, c))

{

return 0;

}

cout << "starting: ";

cin >> start[0] >> start[1];

cout << "goal: ";

cin >> goal[0] >> goal[1];

start[0] = 15 - start[0];

start[1]--;

goal[0] = 15 - goal[0];

goal[1] = 15;

cout << "BFS " << endl;

applyBFS(arr, r, c);

cout << endl;

cout << "DFS " << endl;

applyDFS(arr, r, c);

system("PAUSE");

return 0;

}

bool fillArray(int\*\* arr, int r, int c)

{

fstream fs;

fs.open("robot.txt");

if (!fs.is\_open())

{

cout << "file not exist...\n"

<< endl;

return false;

}

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

{

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

{

fs >> arr[i][j];

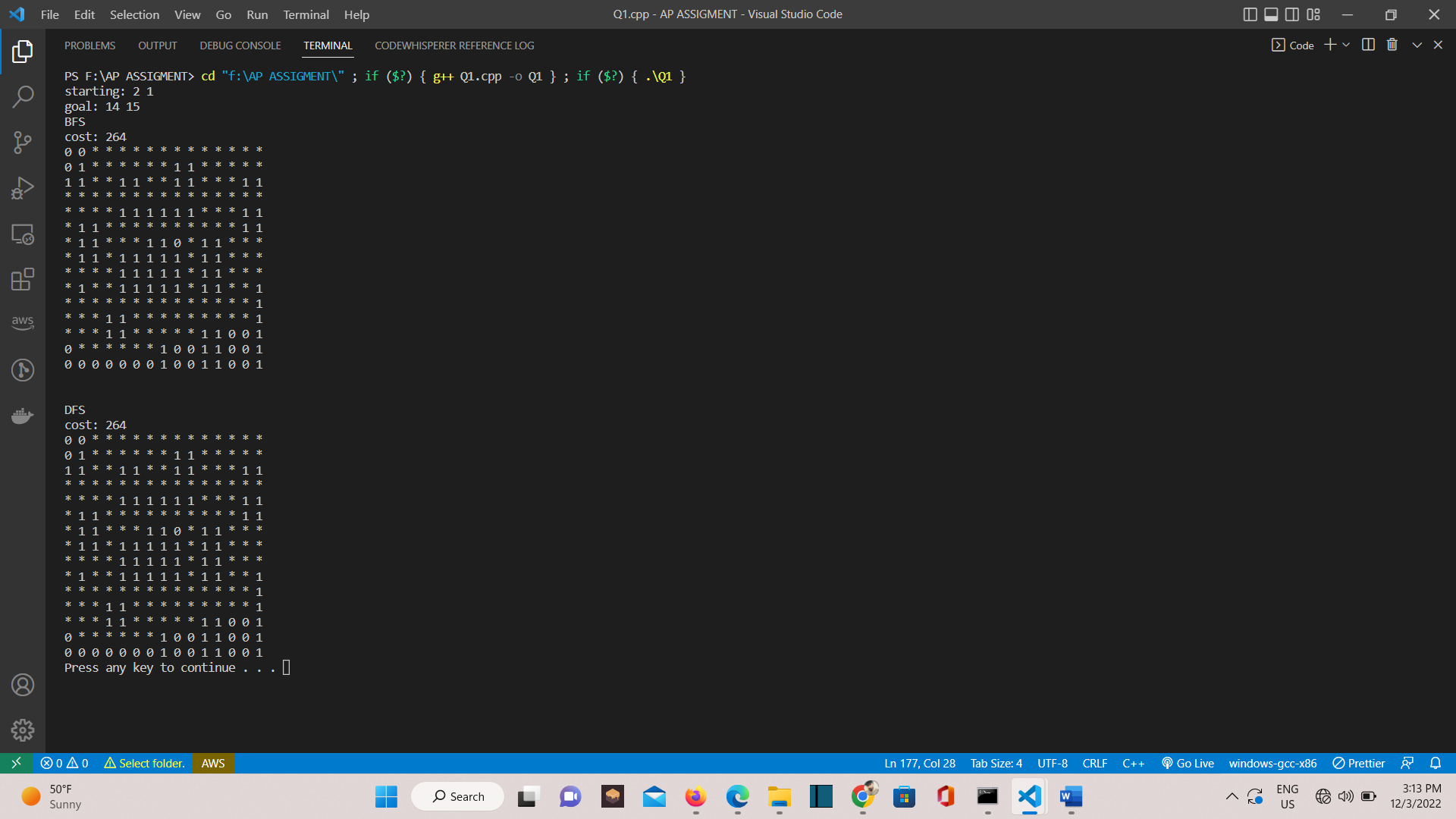
}

}

fs.close();

return true;

}

****

**robot.txt**

**0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**0 1 0 0 0 0 0 0 1 1 0 0 0 0 0**

**1 1 0 0 1 1 0 0 1 1 0 0 0 1 1**

**0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

**0 0 0 0 1 1 1 1 1 1 0 0 0 1 1**

**0 1 1 0 0 0 0 0 0 0 0 0 0 1 1**

**0 1 1 0 0 0 1 1 0 0 1 1 0 0 0**

**0 1 1 0 1 1 1 1 1 0 1 1 0 0 0**

**0 0 0 0 1 1 1 1 1 0 1 1 0 0 0**

**0 1 0 0 1 1 1 1 1 0 1 1 0 0 1**

**0 0 0 0 0 0 0 0 0 0 0 0 0 0 1**

**0 0 0 1 1 0 0 0 0 0 0 0 0 0 1**

**0 0 0 1 1 0 0 0 0 0 1 1 0 0 1**

**0 0 0 0 0 0 0 1 0 0 1 1 0 0 1**

**0 0 0 0 0 0 0 1 0 0 1 1 0 0 1**

**Question No. 2: Graphs**

**(a)**

#include <iostream>

#include <vector>

using namespace std;

const int v = 6;

int Mat[v][v];

struct Vertex

{

char point;

int cost;

};

struct Point

{

char point;

vector<Vertex> vec\_list;

};

void addEdge(Point\*& point, char src, char dest, int cost)

{

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

{

if (point[i].point == src)

{

point[i].vec\_list.push\_back({ dest, cost });

}

}

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

{

if (point[i].point == dest)

{

point[i].vec\_list.push\_back({ src, cost });

}

}

}

void displayGraph(Point\* point)

{

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

{

cout << point[i].point << " ---> ";

for (auto j : point[i].vec\_list)

cout << j.point << " ";

cout << endl;

}

}

void convertToAdjMat(Point\* point)

{

char\* names = new char[v];

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

{

names[i] = point[i].point;

}

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

{

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

Mat[i][j] = 0;

}

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

{

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

{

for (auto k : point[i].vec\_list)

{

if (k.point == names[j])

{

Mat[i][j] = 1;

break;

}

}

}

}

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

{

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

{

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

}

cout << endl;

}

}

int main()

{

Point\* points = new Point[v];

char names[v] = { 'a', 'b', 'c', 'd', 'e', 'f' };

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

{

points[i].point = names[i];

}

addEdge(points, 'a', 'b', 7);

addEdge(points, 'a', 'f', 14);

addEdge(points, 'a', 'c', 9);

addEdge(points, 'b', 'c', 10);

addEdge(points, 'b', 'd', 15);

addEdge(points, 'c', 'd', 11);

addEdge(points, 'c', 'f', 2);

addEdge(points, 'd', 'e', 6);

addEdge(points, 'e', 'f', 9);

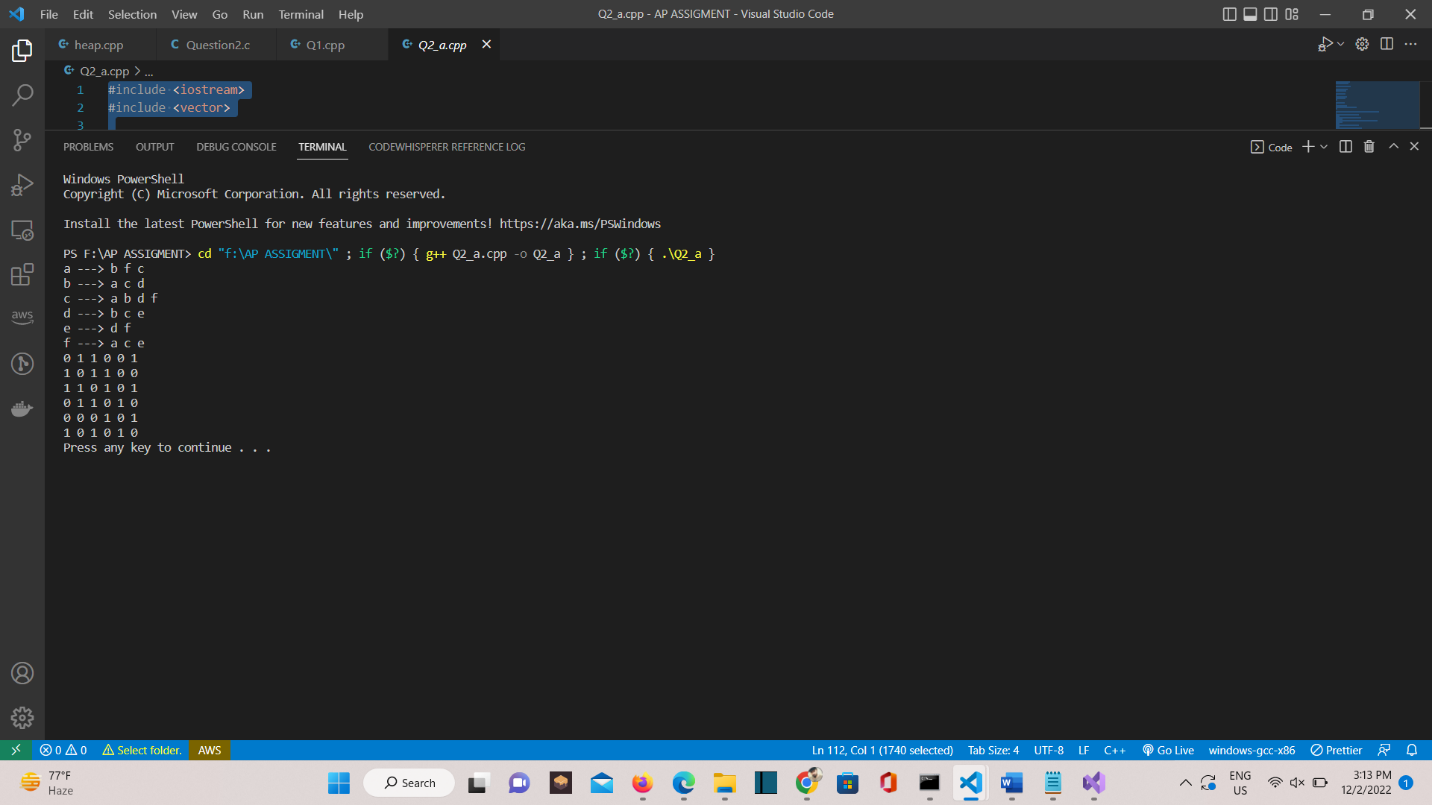
displayGraph(points);

convertToAdjMat(points);

system("PAUSE");

return 0;

}

****

**(b - 1)**

#include <stdio.h>

// V vertices and E Edges

int V, E;

void createAdjMatrix(int Adj[][V + 1],

int arr[][2])

{

for (int i = 0; i < V + 1; i++)

{

for (int j = 0; j < V + 1; j++)

{

Adj[i][j] = 0;

}

}

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

{

int x = arr[i][0];

int y = arr[i][1];

// Update value to 1

Adj[x][y] = 1;

Adj[y][x] = 1;

}

}

void printAdjMatrix(int Adj[][V + 1])

{

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

{

for (int j = 1; j < V + 1; j++)

{

printf(" %d ", Adj[i][j]);

}

printf("\n");

}

}

int main()

{

V = 8;

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

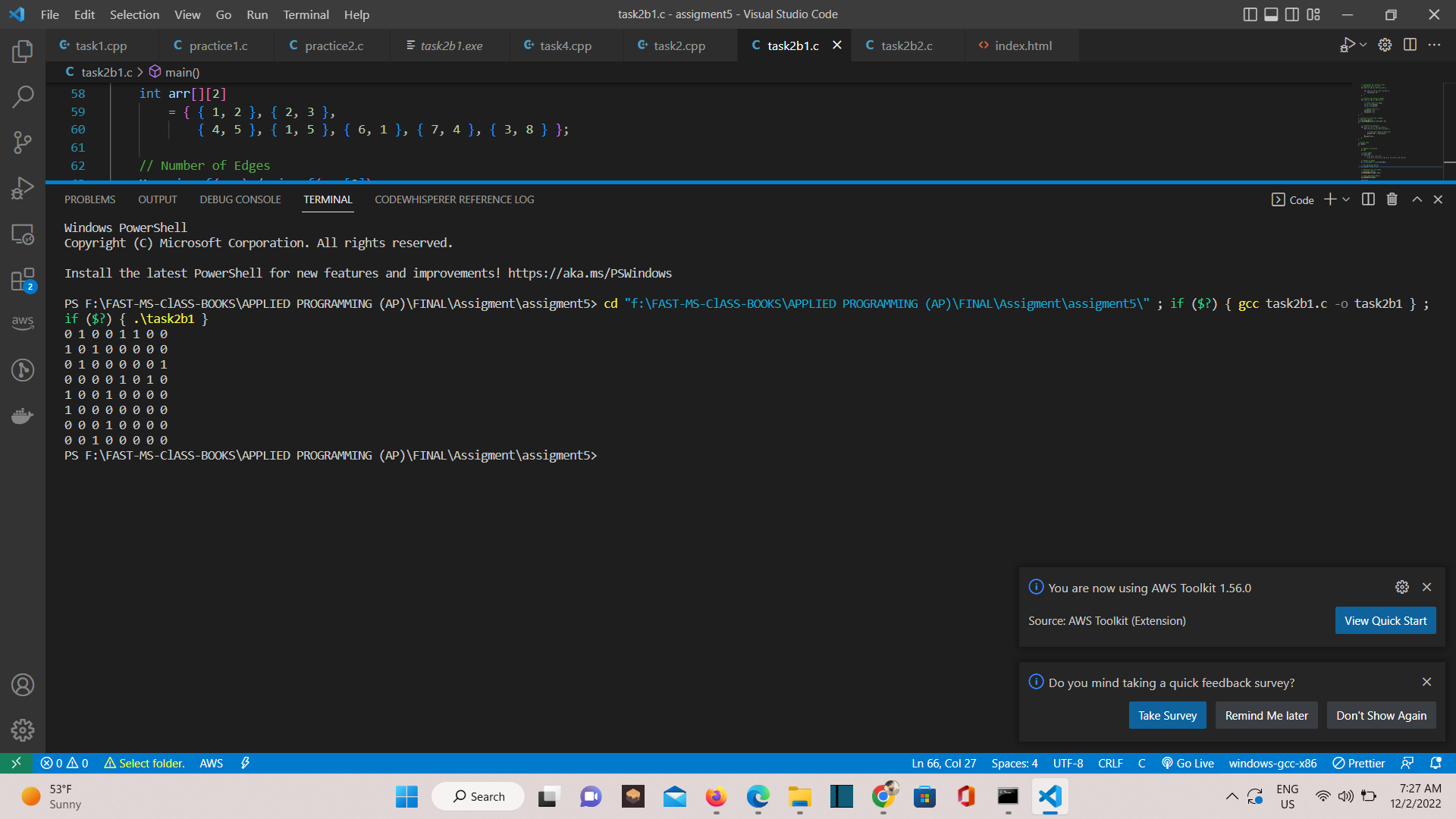
E = sizeof(arr) / sizeof(arr[0]);

int Adj[V + 1][V + 1];

createAdjMatrix(Adj, arr);

printAdjMatrix(Adj);

return 0;

}

**(b - 2)**

#include <stdio.h>

#include <stdlib.h>

struct node

{

int vertex;

struct node\* next;

};

struct node\* createNode(int);

struct Graph

{

int numVertices;

struct node\*\* adjLists;

};

// Create a node

struct node\* createNode(int v)

{

struct node\* newNode = malloc(sizeof(struct node));

newNode->vertex = v;

newNode->next = NULL;

return newNode;

}

// Create a graph

struct Graph\* createAGraph(int vertices)

{

struct Graph\* graph = malloc(sizeof(struct Graph));

graph->numVertices = vertices;

graph->adjLists = malloc(vertices \* sizeof(struct node\*));

int i;

for (i = 0; i < vertices; i++)

graph->adjLists[i] = NULL;

return graph;

}

void addEdge(struct Graph\* graph, int s, int d)

{

struct node\* newNode = createNode(d);

newNode->next = graph->adjLists[s];

graph->adjLists[s] = newNode;

newNode = createNode(s);

newNode->next = graph->adjLists[d];

graph->adjLists[d] = newNode;

}

// Print the graph

void printGraph(struct Graph\* graph)

{

int v;

for (v = 0; v < graph->numVertices; v++)

{

struct node\* temp = graph->adjLists[v];

printf("n Vertex %dn: ", v);

while (temp)

{

printf("%d -> ", temp->vertex);

temp = temp->next;

}

printf("n");

}

}

int main()

{

struct Graph\* graph = createAGraph(8);

addEdge(graph, 0, 1);

addEdge(graph, 1, 2);

addEdge(graph, 2, 3);

addEdge(graph, 4, 5);

addEdge(graph, 1, 5);

addEdge(graph, 6, 1);

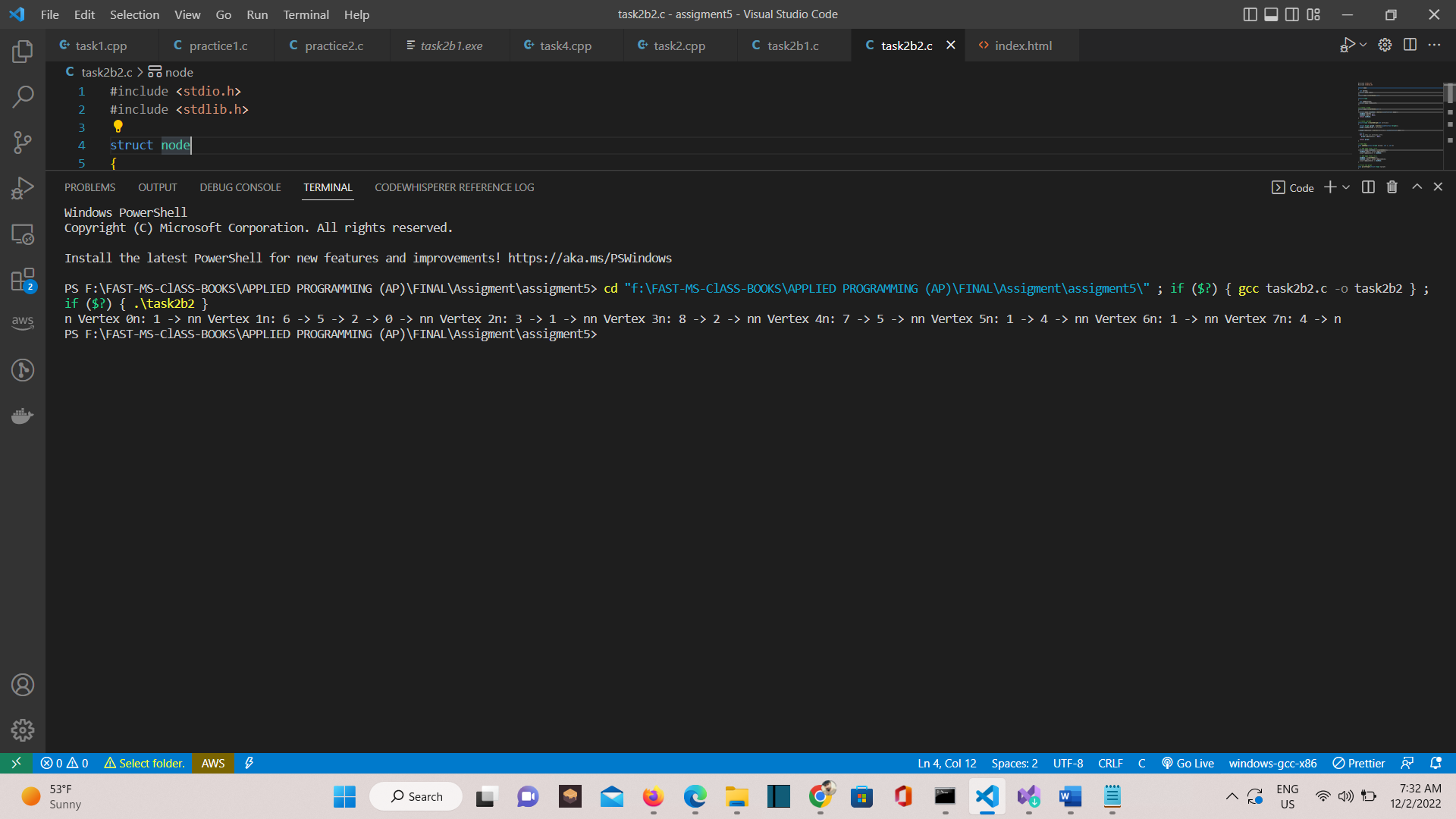
addEdge(graph, 7, 4);

addEdge(graph, 3, 8);

printGraph(graph);

return 0;

}



**Question No. 3: Graphs: BFS and DFS**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

const int v = 20;

string names[v] =

{

"Arad", "Bucharest", "Craiova", "Dobreta", "Eforie",

"Fagaras", "Hirsova", "Iasi", "Lugoj", "Oradea",

"Pitesti", "Rimnicu Vilcea", "Urziceni", "Giurgiu", "Mehadia",

"Sibiu", "Neamt", "Vaslui", "Timisoara", "Zerind" };

struct Vertex

{

string point;

int cost;

};

struct Point

{

string point;

vector<Vertex> vec\_list;

};

int getIndex(Point\*& point, string name)

{

int index = -1;

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

{

if (point[i].point == name)

{

index = i;

break;

}

}

return index;

}

void addEdge(Point\*& point, string src, string dest, int cost)

{

int i = getIndex(point, src);

if (i == -1)

{

cout << "error" << endl;

return;

}

point[i].vec\_list.push\_back({ dest, cost });

i = getIndex(point, dest);

if (i == -1)

{

cout << "error" << endl;

return;

}

point[i].vec\_list.push\_back({ src, cost });

}

void displayGraph(Point\* point)

{

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

{

cout << point[i].point << " ---> ";

for (auto j : point[i].vec\_list)

cout << j.point << " ";

cout << endl;

}

}

void applyBFS(Point\* point, string src, string dest)

{

queue<Vertex> q;

bool\* visited = new bool[v];

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

visited[i] = false;

int startindex = getIndex(point, src);

if (startindex == -1)

{

cout << "error" << endl;

return;

}

q.push({ point[startindex].point, 0 });

visited[startindex] = true;

bool goal = false;

while (!q.empty())

{

Vertex vert = q.front();

q.pop();

if (vert.point == dest)

{

goal = true;

break;

}

cout << vert.point << " ---> ";

for (auto i : point[getIndex(point, vert.point)].vec\_list)

{

startindex = getIndex(point, i.point);

if (visited[startindex] == false)

{

q.push({ i.point, i.cost });

visited[startindex] = true;

}

}

}

if (goal == true)

cout << "goal found...\n";

else

cout << "goal not found...\n";

}

void applyDFSUtil(Point\* point, string dest, int v, bool\*& visited, bool& goal)

{

visited[v] = true;

int index;

for (auto i : point[v].vec\_list)

{

index = getIndex(point, i.point);

if (visited[index] == false)

{

if (i.point == dest)

{

goal = true;

return;

}

cout << i.point << " ---> ";

applyDFSUtil(point, dest, getIndex(point, i.point), visited, goal);

}

if (goal)

{

return;

}

}

}

void applyDFS(Point\* point, string src, string dest)

{

bool\* visited = new bool[v];

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

visited[i] = false;

int startindex = getIndex(point, src);

if (startindex == -1)

{

cout << "error" << endl;

return;

}

bool goal = false;

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

{

if (visited[i] == false && !goal)

{

cout << point[i].point << " ---> ";

applyDFSUtil(point, dest, i, visited, goal);

}

}

if (goal == true)

cout << "goal found...\n";

else

cout << "goal not found...\n";

}

int main()

{

Point\* point = new Point[v];

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

point[i].point = names[i];

addEdge(point, "Arad", "Sibiu", 140);

addEdge(point, "Arad", "Timisoara", 118);

addEdge(point, "Arad", "Zerind", 75);

addEdge(point, "Bucharest", "Fagaras", 211);

addEdge(point, "Bucharest", "Giurgiu", 90);

addEdge(point, "Bucharest", "Pitesti", 101);

addEdge(point, "Bucharest", "Urziceni", 85);

addEdge(point, "Craiova", "Dobreta", 120);

addEdge(point, "Craiova", "Pitesti", 138);

addEdge(point, "Craiova", "Rimnicu Vilcea", 146);

addEdge(point, "Dobreta", "Mehadia", 75);

addEdge(point, "Eforie", "Hirsova", 86);

addEdge(point, "Fagaras", "Sibiu", 99);

addEdge(point, "Hirsova", "Urziceni", 98);

addEdge(point, "Iasi", "Neamt", 87);

addEdge(point, "Iasi", "Vaslui", 92);

addEdge(point, "Lugoj", "Mehadia", 70);

addEdge(point, "Lugoj", "Timisoara", 111);

addEdge(point, "Oradea", "Zerind", 71);

addEdge(point, "Oradea", "Sibiu", 151);

addEdge(point, "Pitesti", "Rimnicu Vilcea", 97);

addEdge(point, "Rimnicu Vilcea", "Sibiu", 80);

addEdge(point, "Urziceni", "Vaslui", 142);

// displayGraph(point);

cout << "BFS \n";

applyBFS(point, "Arad", "Bucharest");

cout << endl;

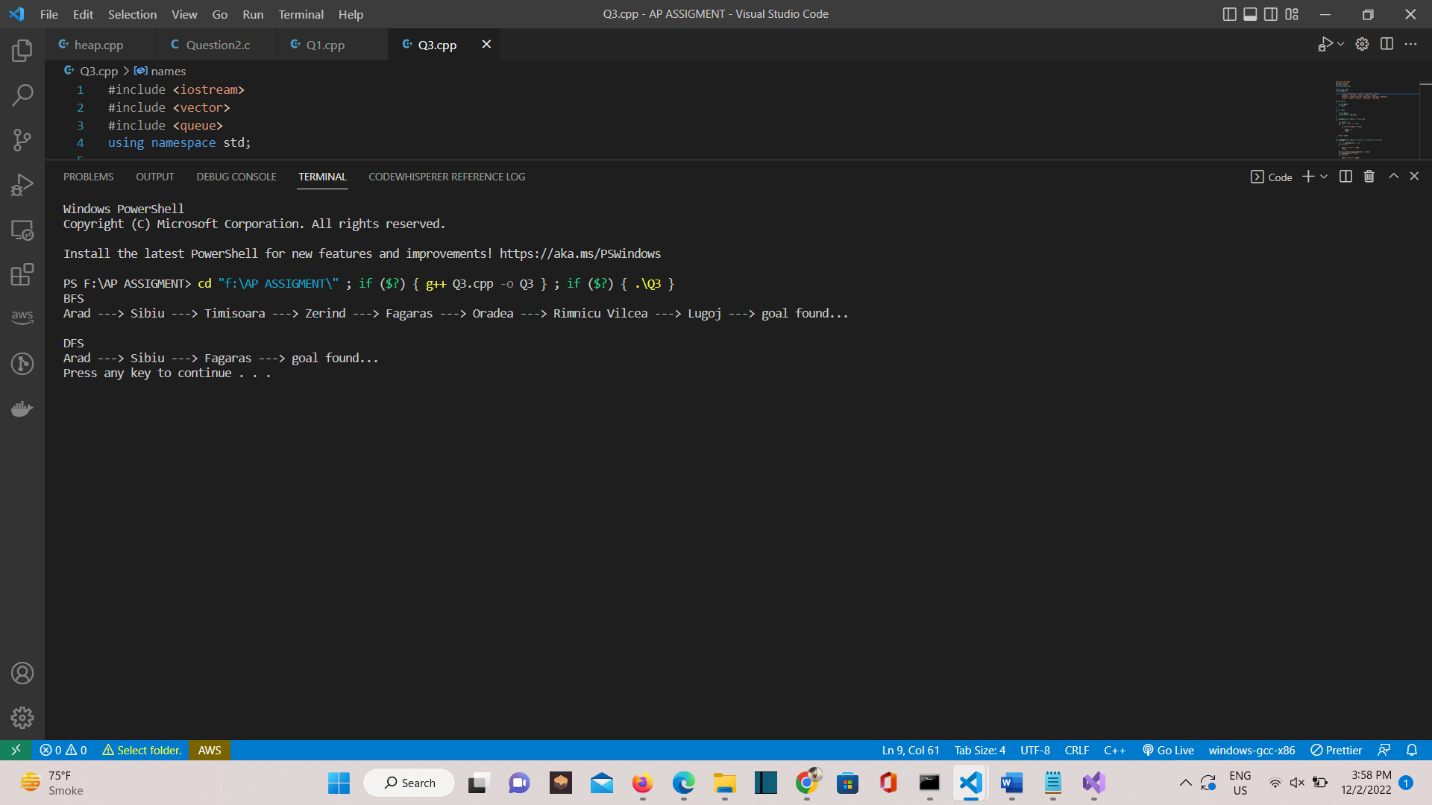
cout << "DFS \n";

applyDFS(point, "Arad", "Bucharest");

system("PAUSE");

return 0;

}

****

**Question No. 4: Heap**

#include <stdio.h>

#include <stdlib.h>

typedef struct MinHeap MinHeap;

struct MinHeap

{

int\* arr;

int size;

int capacity;

};

int parent(int i)

{

return (i - 1) / 2;

}

int left\_child(int i)

{

return (2 \* i + 1);

}

int right\_child(int i)

{

return (2 \* i + 2);

}

int get\_min(MinHeap\* heap)

{

return heap->arr[0];

}

MinHeap\* init\_minheap(int capacity)

{

MinHeap\* minheap = (MinHeap\*)calloc(1, sizeof(MinHeap));

minheap->arr = (int\*)calloc(capacity, sizeof(int));

minheap->capacity = capacity;

minheap->size = 0;

return minheap;

}

MinHeap\* insert\_minheap(MinHeap\* heap, int element)

{

if (heap->size == heap->capacity)

{

fprintf(stderr, "Cannot insert %d. Heap is already full!\n", element);

return heap;

}

heap->size++;

heap->arr[heap->size - 1] = element;

int curr = heap->size - 1;

while (curr > 0 && heap->arr[parent(curr)] > heap->arr[curr])

{

// Swap

int temp = heap->arr[parent(curr)];

heap->arr[parent(curr)] = heap->arr[curr];

heap->arr[curr] = temp;

curr = parent(curr);

}

return heap;

}

MinHeap\* heapify(MinHeap\* heap, int index)

{

if (heap->size <= 1)

return heap;

int left = left\_child(index);

int right = right\_child(index);

int smallest = index;

if (left < heap->size && heap->arr[left] < heap->arr[index])

smallest = left;

if (right < heap->size && heap->arr[right] < heap->arr[smallest])

smallest = right;

if (smallest != index)

{

int temp = heap->arr[index];

heap->arr[index] = heap->arr[smallest];

heap->arr[smallest] = temp;

heap = heapify(heap, smallest);

}

return heap;

}

MinHeap\* delete\_minimum(MinHeap\* heap)

{

// Deletes the minimum element, at the root

if (!heap || heap->size == 0)

return heap;

int size = heap->size;

int last\_element = heap->arr[size - 1];

heap->arr[0] = last\_element;

heap->size--;

size--;

heap = heapify(heap, 0);

return heap;

}

MinHeap\* delete\_element(MinHeap\* heap, int index)

{

heap->arr[index] = get\_min(heap) - 1;

int curr = index;

while (curr > 0 && heap->arr[parent(curr)] > heap->arr[curr])

{

int temp = heap->arr[parent(curr)];

heap->arr[parent(curr)] = heap->arr[curr];

heap->arr[curr] = temp;

curr = parent(curr);

}

heap = delete\_minimum(heap);

return heap;

}

void print\_heap(MinHeap\* heap)

{

printf("Min Heap:\n");

for (int i = 0; i < heap->size; i++)

{

printf("%d -> ", heap->arr[i]);

}

printf("\n");

}

void free\_minheap(MinHeap\* heap)

{

if (!heap)

return;

free(heap->arr);

free(heap);

}

int main()

{

MinHeap\* heap = init\_minheap(10);

insert\_minheap(heap, 40);

insert\_minheap(heap, 50);

insert\_minheap(heap, 5);

print\_heap(heap);

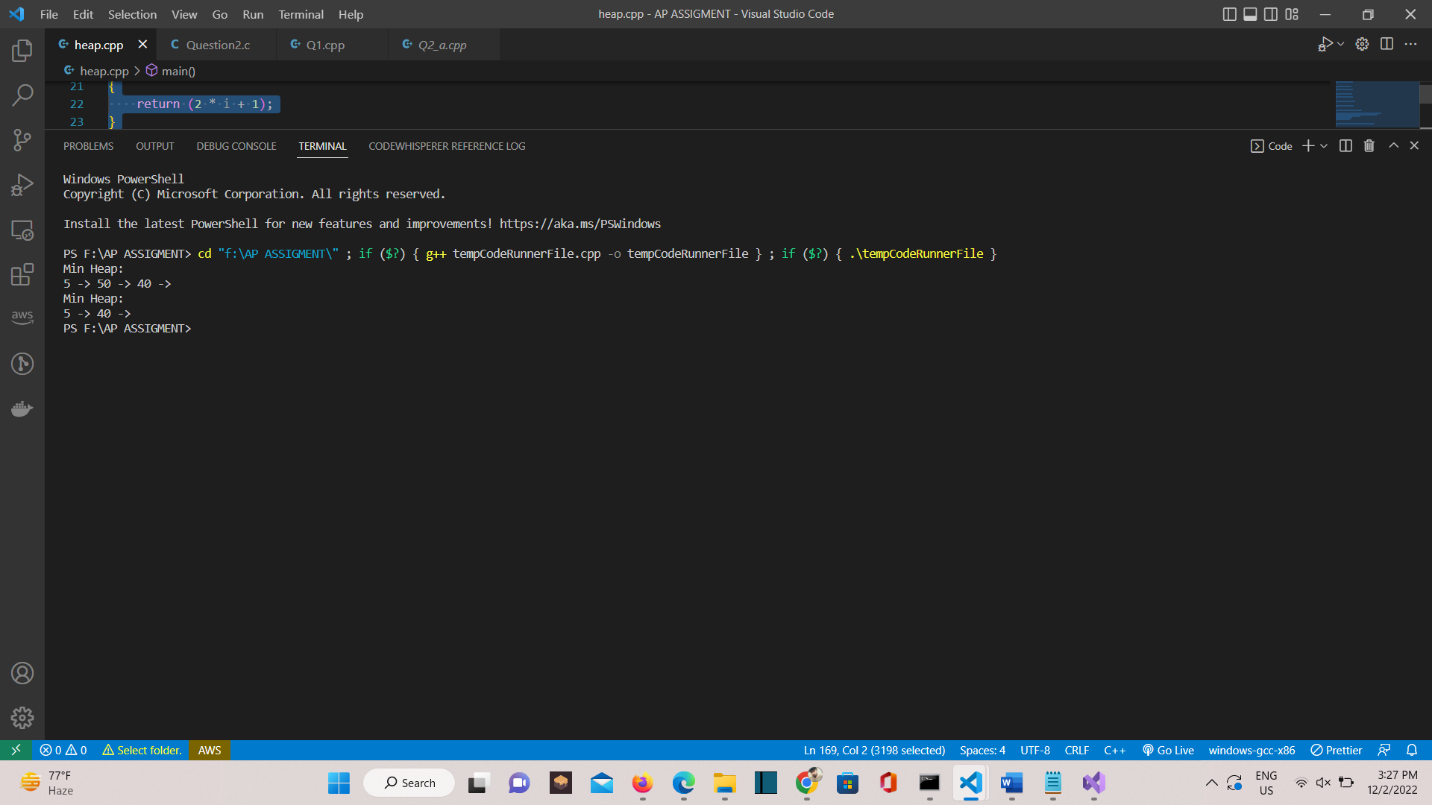
delete\_element(heap, 1);

print\_heap(heap);

free\_minheap(heap);

return 0;

}

****