**Waleed Afzal**

**23p-0566**

**BSCS – 3D  
DSA Assignment 4**

**Ques1:**#include <iostream>

#include <vector>

using namespace std;

class MinHeap {

vector<int> heap;

void swap(int& num1, int& num2) {

int temp = num1;

num1 = num2;

num2 = temp;

}

void percolateUp(int index) {

while (index > 0) {

int parentIndex = (index - 1) / 2;

if (heap[index] < heap[parentIndex]) {

swap(heap[index], heap[parentIndex]);

index = parentIndex;

}

else

break;

}

}

void percolateDown(int index) {

int size = heap.size();

while (true) {

int leftChild = 2 \* index + 1;

int rightChild = 2 \* index + 2;

int smallest = index;

if (leftChild < size && heap[leftChild] < heap[smallest])

smallest = leftChild;

if (rightChild < size && heap[rightChild] < heap[smallest])

smallest = rightChild;

if (smallest == index)

break;

swap(heap[index], heap[smallest]);

index = smallest;

}

}

public:

void insertHeap(int val) {

heap.push\_back(val);

percolateUp(heap.size() - 1);

}

int deleteHeap() {

if (heap.size() == 0) //if size is zero

return -1;

//else

int min = heap.front(); //first value of heeap

if (heap.size() == 1)

heap.pop\_back();

else {

heap.front() = heap.back();

heap.pop\_back();

percolateDown(0);

}

return min;

}

void display() {

if (heap.size() == 0) {

cout << "\n-----Error! Heap is Empty-----\n";

return;

}

cout << "\nMin Heap : ";

for (int val : heap) {

cout << val << " ";

}

cout << endl;

}

};

int main() {

MinHeap heapObj;

char choice;

int val = 0;

do {

cout << "\n<======Min Heap Operations======>"

<< "\n1. Insert Tasks"

<< "\n2. Extract Highest Priority"

<< "\n3. Display Heap"

<< "\n4. Exit"

<< "\nEnter choice (1-4) : ";

cin >> choice;

switch (choice)

{

case '1':

cout << "Enter elements (end input with a non-numeric value) : ";

while (cin >> val) {

heapObj.insertHeap(val);

}

cin.clear();

cin.get();

cout << "\n-----Tasks Inserted Successfully!-----\n";

break;

case '2': {

cout << "\nEnter the number of priorities to be extracted : ";

cin >> val;

int i = 1;

while (i <= val) {

int priority = heapObj.deleteHeap();

if (priority != -1) {

cout << "Highest-Priority Task " << i << ": " << priority << endl;

++i;

}

else {

cout << "\n-----Error! Heap is Empty-----\n";

break;

}

}

break;

}

case '3':

heapObj.display();

break;

case '4':

cout << "\n-----Program Terminated!-----\n";

break;

default:

cout << "\n-----Invalid Input!-----\n";

break;

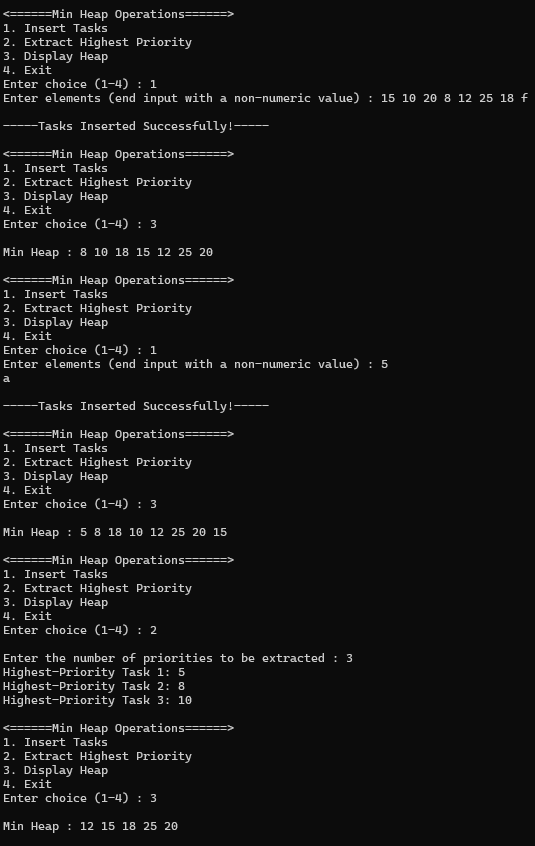
}

} while (choice != '4');

system("pause");

return 0;

}

 **Ques2:**

#include <iostream>

#include <vector> //better to use vector for graphs then arrays

#include <climits> //for int\_min and int\_max

using namespace std;

bool nextPermutation(vector<int>& arr) {

int n = arr.size();

int i = n - 2;

while (i >= 0 && arr[i] >= arr[i + 1])

i--;

if (i < 0)

return false;

int j = n - 1;

while (arr[j] <= arr[i])

j--;

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

reverse(arr.begin() + i + 1, arr.end());

return true;

}

void reverse(vector<int>& arr, int start) {

int end = arr.size() - 1;

while (start < end) {

swap(arr[start], arr[end]);

start++;

end--;

}

}

void travellingSalesmanProblem(int graph[][4], int s, int size) {

vector<int> vertex;

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

if (i != s)

vertex.push\_back(i);

int min\_path = INT\_MAX;

vector<int> min\_route; //array for storing the path

do {

int current\_weight = 0;

int k = s;

vector<int> current\_route = { s };

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

current\_weight += graph[k][vertex[i]];

k = vertex[i];

current\_route.push\_back(k);

}

current\_weight += graph[k][s];

current\_route.push\_back(s);

if (current\_weight < min\_path) {

min\_path = current\_weight;

min\_route = current\_route;

}

} while (nextPermutation(vertex));

// Print the minimum path

cout << "Minimum Path: ";

for (int i = 0; i < min\_route.size(); i++) {

cout << char('A' + min\_route[i]);

if (i != min\_route.size() - 1)

cout << " -> ";

}

cout << endl;

cout << "Minimum Cost: " << min\_path << endl;

}

int main() {

int graph[4][4] = { // A B C D

{0, 10, 15, 20}, // A [0 10 15 20 ]

{10, 0, 35, 25}, // B |10 0 35 25 |

{15, 35, 0, 30}, // C |15 35 0 30 |

{20, 25, 30, 0} // D [20 25 30 0 ]

};

int start = 0; // Starting city

travellingSalesmanProblem(graph, start, 4);

system("pause");

return 0;

}

**A screenshot of a computer

Description automatically generated**

**Ques3:**#include <iostream>

#include <queue>

#include <stack>

#include <vector>

using namespace std;

struct Node {

int vertex;

Node\* next;

Node(int V) : vertex(V), next(nullptr) {}

};

class Graph {

Node\*\* adjList;

int vertices;

public:

Graph(int V) {

vertices = V;

adjList = new Node \* [vertices];

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

adjList[i] = nullptr;

}

}

// Function to add edge for undirected graph

void addEdge(int src, int dest) const {

Node\* newNode = new Node(dest);

if (adjList[src] == nullptr) {

adjList[src] = newNode;

}

else {

Node\* temp = adjList[src];

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

newNode = new Node(src);

if (adjList[dest] == nullptr) {

adjList[dest] = newNode;

}

else {

Node\* temp = adjList[dest];

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

}

void BFS(int start) {

queue<int> q;

bool\* visited = new bool[vertices]();

char user;

q.push(start);

visited[start] = true;

cout << "BFS Traversal : ";

while (!q.empty()) {

int current = q.front();

q.pop();

user = 'A' + current;

cout << user << " ";

Node\* temp = adjList[current];

while (temp != nullptr) {

if (!visited[temp->vertex]) {

visited[temp->vertex] = true;

q.push(temp->vertex);

}

temp = temp->next;

}

}

cout << endl;

delete[] visited;

}

void DFS(int start) {

stack<int> stack;

Node\* temp;

char user;

bool\* visited = new bool[vertices]();

cout << "DFS Traversal : ";

visited[start] = true;

stack.push(start);

user = 'A' + start;

cout << user << " ";

while (!stack.empty()) {

int current = stack.top();

temp = adjList[current];

while (visited[temp->vertex] && temp->next != nullptr) {

temp = temp->next;

}

if (!visited[temp->vertex]) {

visited[temp->vertex] = true;

stack.push(temp->vertex);

user = 'A' + temp->vertex;

cout << user << " ";

}

else {

stack.pop();

}

}

cout << endl;

delete[] visited;

}

void findShortestPath(int start, int end) {

queue<int> q;

vector<int> distance(vertices, -1);

vector<int> parent(vertices, -1);

char user1, user2;

q.push(start);

distance[start] = 0;

while (!q.empty()) {

int current = q.front();

q.pop();

Node\* temp = adjList[current];

while (temp != nullptr) {

if (distance[temp->vertex] == -1) {

distance[temp->vertex] = distance[current] + 1;

parent[temp->vertex] = current;

q.push(temp->vertex);

}

temp = temp->next;

}

}

if (distance[end] == -1) {

user1 = 'A' + start;

user2 = 'A' + end;

cout << "No path exists between " << user1 << " and " << user2 << "." << endl;

}

else {

user1 = 'A' + start;

user2 = 'A' + end;

cout << "Shortest path between " << user1 << " and " << user2 << " is: ";

vector<int> path;

for (int at = end; at != -1; at = parent[at]) {

path.push\_back(at);

}

for (int i = path.size() - 1; i >= 0; i--) {

cout << char('A' + path[i]) << (i > 0 ? " -> " : "");

}

cout << " (Distance: " << distance[end] << ")" << endl;

}

}

~Graph() {

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

Node\* head = adjList[i];

while (head->next != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

delete[] adjList;

}

};

int main() {

Graph adjListObj(6);

adjListObj.addEdge(0, 1); // A-B

adjListObj.addEdge(0, 2); // A-C

adjListObj.addEdge(1, 3); // B-D

adjListObj.addEdge(1, 4); // B-E

adjListObj.addEdge(2, 5); // C-F

// Final Adjacency List (when adding edge at head)

// 0 : 2 1

// 1 : 4 3 0

// 2 : 5 0

// 3 : 1

// 4 : 1

// 5 : 2

adjListObj.BFS(0);

adjListObj.DFS(0);

adjListObj.findShortestPath(0, 5);

adjListObj.findShortestPath(0, 3);

system("pause");

return 0;

}

**A screenshot of a computer

Description automatically generated**

**Ques4:**#include <iostream>

#include <vector>

using namespace std;

class MST {

vector<vector<int>> mat;

int vertices;

public:

MST(int V) {

vertices = V; // (rows, vector<int>(vertices));

mat = vector<vector<int>>(vertices, vector<int>(vertices));

}

void insertMatrix() {

cout << "Enter the adjacency matrix (use O for no edge): " << endl;

for (int row = 0; row < mat.size(); row++) {

for (int col = 0; col < mat[row].size(); col++) {

cin >> mat[row][col];

}

}

cout << endl;

}

void primsMST() {

vector<int> key(vertices, 1000);

vector<bool> inMST(vertices, false);

vector<int> parent(vertices, -1);

key[0] = 0;

int mstWeight = 0;

cout << "\nPrim's MST:\n";

for (int count = 0; count < vertices - 1; count++) {

int minKey = 1000, u = -1;

for (int v = 0; v < vertices; v++) {

if (!inMST[v] && key[v] < minKey) {

minKey = key[v];

u = v;

}

}

inMST[u] = true;

for (int v = 0; v < vertices; v++) {

if (mat[u][v] != 0 && mat[u][v] != 1000 && !inMST[v] && mat[u][v] < key[v]) {

key[v] = mat[u][v];

parent[v] = u;

}

}

}

for (int v = 1; v < vertices; v++) {

if (parent[v] != -1) {

cout << parent[v] << " - " << v << "\tWeight: " << mat[parent[v]][v] << endl;

mstWeight += mat[parent[v]][v];

}

}

cout << "Minimum cost using Prim's: " << mstWeight << endl;

}

};

int main() {

int V;

// Prompt the user to enter the number of vertices

cout << "Enter the number of vertices : ";

cin >> V;

MST graph(V);

graph.insertMatrix(); // Calling function take user input

graph.primsMST(); // Calling function to calculate Prim's MST

system("pause");

return 0;

}

**A screenshot of a computer program

Description automatically generated**