**Data Structures**

**(CSE22101)**

**Assignment 2:**

**Saken Kenzhegulov**

**(20132027)**

#ifndef MAP\_H

#define MAP\_H

// Array-based stack and queue data structure

// CSE221 Fall 2014

// Won-Ki Jeong (wkjeong@unist.ac.kr)

#include <iostream>

#include <math.h>

using namespace std;

template<class type>

class Stack{

public:

// Default constructor

Stack(){

this->capacity = 10; //assigning stack capacity value

this->array = new type[capacity+1]; //allocating memory for dynamic array

this->top = 0; //indicating empty stack

}

// Destructor

~Stack(){

delete [] array; //deleting array references

}

// Return top element of stack

type& Top(){

return this->array[top]; //returning item on the top of the stack

}

void Push(const type& item){

if (this->top == this->capacity) { //expanding our array if stack is already full

type \*temp\_arr;

this->capacity += 10;

temp\_arr = new type[capacity+1];

for(int i=0; i<=top; i++) temp\_arr[i] = this->array[i];

this->array = temp\_arr;

}

array[++this->top] = item; //adding one more item to our stack

}

void Pop(){

if(this->top) top--; //removing the data from the top

}

bool IsEmpty() const{

if(this->top) return false; //checking for emptiness by looking at the value of top

else return true; //top is equal to zero if stack is empty

}

private:

// Data

type \*array;

int capacity;

int top;

};

template<class type>

class Queue

{

public:

// Default constructor

Queue(){

this->capacity = 10; //assigning capacity value of queue

this->array = new type[capacity]; //allocating memory for query array

this->front = 0; //front value is zero

this->rear = 0; //rear is also zero since no items in the query at the beginning

}

// Destructor

~Queue(){

delete [] array; //removing array references

}

// Return top element of stack

type& Front(){

return this->array[front]; //returning front item of the query

}

void Enque(const type& item){

if (this->rear == this->capacity) { //expanding query capacity, and removing used items which are placed

type \*temp\_arr; //before front, in order to use less memory

capacity = this->rear-this->front + 10;

temp\_arr = new type[capacity];

for(int i=0; i<this->rear-this->front+1; i++) temp\_arr[i] = array[i+this->front];

this->rear -= this->front;

this->front = 0;

this->array = temp\_arr;

}

this->array[rear++] = item; //adding new item to the end of query

}

void Deque(){

this->front++; //removing front item of the query

}

bool IsEmpty() const{

if(front == rear) return true; //when front is equal to rear, our query is empty

else return false; //otherwise not

}

private:

// Data

type \*array;

int capacity;

int front;

int rear;

};

// Implementations

/\*\*

\* Read in map file, find the shortest path,

\* and print the path with the map

\*\*/

void path(const char\* map)

{

freopen("map.txt", "rt", stdin); //preparing Input file

int n,m, X, Y, k; //variables for dimensions of the map, exit point, and additional purposes

cin>>n>>m; //reading map dimensions

int a[n][m]; //creating 2D array for our map

const int inf = 1000000; //constant used to show unused point in the map

string s; //string to get input

Queue<int> x; //Queue class for row

Queue<int> y; //Queue class for column

//reading input and converting to int

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

cin>>s;

for(int j=0; j<m; j++) a[i][j] = s[j]-'0';

}

//preparing map for BFS

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

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

if(a[i][j] == 1) a[i][j] = -1; else //value for walls

if(a[i][j] == 2) a[i][j] = 0, x.Enque(i), y.Enque(j); else //entrance, inserting this point to Queue

if(a[i][j] == 3) a[i][j] = inf, X = i, Y = j; //exit point, saving coordinates

else a[i][j] = inf; //unvisited empty point

int i,j; //temporary coordinates

while(a[X][Y] == inf) {

i = x.Front(); x.Deque(); //getting i coordinate from Queue

j = y.Front(); y.Deque();//getting j coordinate from Queue

k = a[i][j] + 1; //path distance

if(i-1>=0 && a[i-1][j]==inf) { //if we have a path to downside, we add that point to Queue

x.Enque(i-1);

y.Enque(j);

a[i-1][j] = k;

}

if(i+1<n && a[i+1][j]==inf) { //if we have a path to upside, we add that point to Queue

x.Enque(i+1);

y.Enque(j);

a[i+1][j] = k;

}

if(j-1>=0 && a[i][j-1]==inf) { //if we have a path to left, we add that point to Queue

x.Enque(i);

y.Enque(j-1);

a[i][j-1] = k;

}

if(j+1<m && a[i][j+1]==inf) { //if we have a path to right, we add that point to Queue

x.Enque(i);

y.Enque(j+1);

a[i][j+1] = k;

}

}

i = X; j = Y; k = a[i][j]; //going backward to get the path

while(true) { //looking for point where is the distance less for 1, since previous points distance is less for 1

if(i-1>=0 && a[i-1][j]+1==k) i--; else

if(i+1<n && a[i+1][j]+1==k) i++; else

if(j-1>=0 && a[i][j-1]+1==k) j--; else

if(j+1<m && a[i][j+1]+1==k) j++;

k = a[i][j]; //saving current distance to entrance

if(!k) break; //if k is zero, we are on entrance and need to stop here

a[i][j] = -2; //marking our path in a map

}

//printing resulted map with shortest distance

a[X][Y] = 3;

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

for(int j=0; j<m; j++) {

if(a[i][j] == -1) cout<<1; else

if(a[i][j] == -2) cout<<'\*'; else

if(a[i][j] == 0) cout<<2; else

if(a[i][j] == 3) cout<<3; else cout<<0;

}

cout<<endl;}

return;

}

#endif