CC - 213 : DATA STRUCTURES AND ALGORITHMS

BSCS MORNING - FALL 2022

ASSIGNMENT # 02

TOPIC: **STACK AND QUEUE**

Question # 01

Problem Statement:

You are required to write a program in C++ that implements a function to process a string and eliminate consecutive duplicate characters. You must decide which data structure (stack or queue) to use to achieve this. The solution must be implemented in linear time using a single loop (no nested loops).

Requirements:

Implement a function string **eliminateConsecutiveDuplicates(string input)** that takes a string as an input parameter and returns a string with consecutive duplicate characters removed.

The function must operate in linear time, using a single loop (no nested loops).

Your program should handle both uppercase and lowercase letters and preserve their case.

You should choose and implement the most appropriate data structure for this task.

Example Output:

```
Input: aabbccddeeff
Output: abcdef
Input: aaaabbbbccccddddeeee
Output: abcde
Input: abccba
Output: abcba
```

Question # 02

Objective:

Implement a Depth First Search (DFS) algorithm using a stack to determine if a path exists between two points in a 2D grid with randomly generated obstacles. The permitted moves are right, left, up, down, and diagonals.

Problem Statement:

You are required to write a program in C++ that generates a 10x10 matrix filled with zeros and then randomly fills 60% of the matrix with ones (representing obstacles). The program should then take a starting and ending point from the user and use the DFS algorithm to determine if a path exists between these points. The DFS implementation must use a stack data structure. Additionally, you should provide a menu-driven main program to facilitate testing.

Requirements:

- Create a 2D matrix of size 10x10 filled with zeros.
- Randomly generate obstacles that cover 60% of the matrix (i.e., fill 60 cells with ones).
- Implement a function `bool pathExists(int startX, int startY, int endX, int endY)` that takes the starting and ending coordinates and uses the DFS algorithm with a stack to determine if a path exists.
- Ensure the DFS algorithm handles out-of-bounds searching.
- Provide a menu-driven main program to test the functionality.
- Permitted moves are right, left, up, down, and diagonals.

Bonus: Make the board size generalized, allowing the user to input the size of the grid and display the path if it exists.

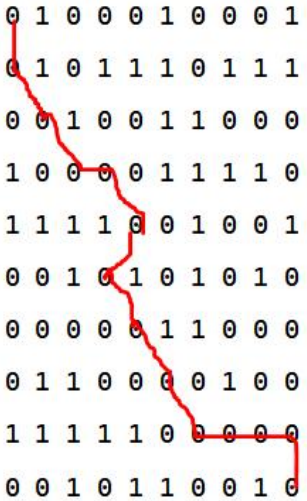
Sample Code:

```cpp
#include <iostream>
#include <vector>
using namespace std;
class Grid
{
public:
    bool ** grid;
    int rows, cols;
    Grid(int rows, int cols) : rows(rows), cols(cols)
    {
        // initialize data member 'grid' as per user requirements
        generateObstacles();
    }
    ~Grid()
    {
        // deallocate memory properly
    }
    void generateObstacles()
```

```
    {
        // Logic to fill 60% of the grid with obstacles
    }
    bool pathExists(int startX, int startY, int endX, int endY)
    {
        // Logic for DFS to check if a path exists
        return false; // Placeholder
    }
    void displayGrid()
    {
        // Logic to display the grid
    }
};
int main()
{
    int rows = 10, cols = 10;
    Grid grid(rows, cols);
    int startX, startY, endX, endY;
    cout << "Enter starting point (x y): ";
    cin >> startX >> startY;
    cout << "Enter ending point (x y): ";
    cin >> endX >> endY;
    grid.displayGrid();
    if (grid.pathExists(startX, startY, endX, endY))
    {
        cout << "Path exists!" << endl;
    }
    else
    {
        cout << "No path exists!" << endl;
    }
    return 0;
}
```

Sample Output:

```
Enter starting point (x y): 0 0
Enter ending point (x y): 9 9
0 1 0 0 0 1 0 0 0 1
0 1 0 1 1 1 0 1 1 1
0 0 1 0 0 1 1 0 0 0
1 0 0 0 0 1 1 1 1 0
1 1 1 1 0 0 1 0 0 1
0 0 1 0 1 0 1 0 1 0
0 0 0 0 0 1 1 0 0 0
0 1 1 0 0 0 0 1 0 0
1 1 1 1 1 0 0 0 0 0
0 0 1 0 1 1 0 0 1 0
```



```
Result : Path exists!
Path : (0,0) -> (1,0) -> (2,1) -> (3,2) -> (3,3) -> (4,4) -> (5,3)
-> (6,4) -> (7,5) -> (8,6) -> (8,7) -> (8,8) -> (8,9) -> (9,9)
```

Question # 03

Problem Description

George is the world's worst tic-tac-toe player. He has never won a game. You, being a good friend, are determined to help him win his first game. You will insist that George makes the first move in the game.

In addition, you have noticed that George has been selecting squares in a methodical way. He has a definite preference order in which he selects squares. He chooses as his next square, the open square that is highest on his preference list. You will devise a list of moves that ensures, even with a really bad strategy, that George will be assured of a win.

Tic-tac-toe is a paper-and-pencil game for two players, **X** and **O**, who take turns marking the spaces in a **3 × 3 grid**. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

Objective:

Making Sure George wins tic-tac-toe in the fewest possible moves.

Input

The input consists of nine lines with the preference list of George's moves. Each line contains a two integers giving the row and the column of the moves. The rows and columns are numbers between 1 and 3, inclusive. **The nine squares in the input are distinct.**

Output

The output will consist of a list of turns that you will make to ensure **that George wins in the fewest possible moves**. If at any point, you could make more than one move that would lead to a win for George in the fewest moves; you should choose the move that is in the lowest numbered row. If there are multiple moves in the lowest numbered row, you should choose the one with the lowest numbered column.

Sample Input	Sample Output
1 2 3 3 3 1 1 1 2 2 2 3 3 2 2 1 1 3	1 1 2 2 2 3

Question # 04

Problem Description

In an operating system, efficient memory management is crucial for performance. One common approach to manage page frames is to use the Least Recently Used (LRU) algorithm. In this assignment, you will implement a fixed-size queue using a linked list to simulate page frame management. The queue should follow the LRU policy, where the least recently used page is the first to be removed when the queue is full.

You are required to:

1. Implement a queue using a linked list.
2. Ensure the queue has a fixed size, defined as the number of available page frames.
3. Follow the LRU priority for adding and removing pages.
4. Track and output the number of page hits and page faults for a given sequence of page requests.

LRU (Least Recently Used) Cache

An LRU cache evicts the least recently used item when the cache reaches its fixed size limit. This ensures that the most recently accessed items are kept in the cache, and the least accessed items are removed.

How LRU Works Using Queue

A queue is a data structure that follows the First-In-First-Out (FIFO) principle.
For an LRU cache, we modify the queue to also consider the most recently accessed items.
When a page is accessed:
If the page is already in the cache (page hit), it is moved to the front of the queue.
If the page is not in the cache (page fault), it is added to the front of the queue. If the queue is full, the page at the back of the queue (the least recently used) is removed.
The Queue is updated every time an element (page) is accessed. The accessed element is moved to the front of top of the queue (assuming the implementation in which element is popped from the end of a Queue).

Input

- The first line contains an integer N representing the number of page frames.
- The second line contains a sequence of integers representing the page requests, where each integer is a page value ranging from 1 to 9.

Output

- The first line should contain the number of page hits.
- The second line should contain the number of page faults.

Definitions

- **Page Hit**: When a requested page is already present in the queue.
- **Page Fault**: When a requested page is not in the queue and a new page needs to be added.

Sample Input	Sample Output
3 1 2 3 4 1 2 5 1 2 3 4 5	3 9

Explanation

- Initially, the queue is empty.
- Requesting page 1 results in a page fault (queue: [1]).
- Requesting page 2 results in a page fault (queue: [1, 2]).
- Requesting page 3 results in a page fault (queue: [1, 2, 3]).
- Requesting page 4 results in a page fault, and page 1 is removed (queue: [2, 3, 4]).
- Requesting page 1 results in a page fault, and page 2 is removed (queue: [3, 4, 1]).
- Requesting page 2 results in a page fault, and page 3 is removed (queue: [4, 1, 2]).
- Requesting page 5 results in a page fault, and page 4 is removed (queue: [1, 2, 5]).
- Requesting page 1 results in a page hit (queue: [2, 5, 1]).
- Requesting page 2 results in a page hit (queue: [5, 1, 2]).
- Requesting page 3 results in a page fault, and page 5 is removed (queue: [1, 2, 3]).
- Requesting page 4 results in a page fault, and page 1 is removed (queue: [2, 3, 4]).
- Requesting page 5 results in a page fault, and page 2 is removed (queue: [3, 4, 5]).

In total, there are 3 page hits and 9 page faults.

Question # 05

Problem Description

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return **the number of islands.**

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically (NOT DIAGONALLY). You may assume all four edges of the grid are all surrounded by water.

Sample Input	Sample Output
• [["1","1","1","1","0"], ["1","1","0","1","0"], ["1","1","0","0","0"], ["0","0","0","0","0"]] • [["1","1","0","0","0"], ["1","1","0","0","0"], ["0","0","1","0","0"], ["0","0","0","1","1"]]	• 1 • 3

Guidelines:

- Thoroughly read the assignment prompt to understand the requirements, constraints, and objectives.
- Choose meaningful names for variables, functions, and classes that clearly describe their purpose.
- Write comments to explain complex sections of your code.
- Follow consistent indentation and spacing to make your code more readable.
- Submitting code that is not your own, whether from online sources, other students, previous assignments or generated by AI tools will result in penalty.

Happy coding!