Data Structures Fall 2023

Lab Task 08 Queues and Unit Testing

Introduction to Queues:

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. In a queue, the first element added to the queue is the first one to be removed. Queues are often used to manage tasks or data in a way that ensures that the oldest item is processed or removed before newer items.

A common way to implement a queue is by using a linked list. In a linked list-based queue, each element of the queue is represented as a node in the linked list.

Usage of Queues in daily life:

Queues are used in various aspects of daily life and can be found in many scenarios. Here are some examples of queue usage in everyday life:

1. **Supermarket Checkout**: When you're at the supermarket, you typically join a queue (line) to check out your items. The first person in the queue is the first to be served, following the FIFO principle.
2. **ATM Machines**: At an ATM machine, people form a queue to withdraw money. The person who arrives first gets to use the ATM first.
3. **Traffic**: In traffic, vehicles follow a queue-like order. Cars move forward in a line, and the car at the front of the line is the first to proceed when the traffic light turns green.
4. **Airport Security**: At airport security checkpoints, travelers wait in a queue to pass through security screening. The first person in line goes through the screening process before the next person.
5. **Customer Support Hotlines**: When you call a customer support hotline, you are often placed in a queue and served by support agents in the order you called. The person who has been waiting the longest is typically the next to be assisted.

Lab Work

Task 1: Queue Implementation

Objective: Implement a queue data structure using templates and perform basic operations on it.

Requirements:

1. Create a class called **Node using templates** to represent a node for a queue. Each node should have an integer data field and a pointer to the next node.
2. Create a class called **Queue using templates** to represent the queue data structure.
3. Queue will have a **front** pointer that will point towards the node in the front of the queue and a **rear** pointer that will point towards the end of the queue.
4. Implement the following methods in the **Queue** class:
   - **enqueue(item)**: Add an item to the back of the queue.
   - **dequeue()**: Remove and return the item from the front of the queue.
   - **peek()**: Return the item at the front of the queue without removing it.
   - **is_empty()**: Return **True** if the queue is empty, **False** otherwise.
   - **size()**: Return the number of elements in the queue.
   - **clear()**: Clear all elements from a node-based queue, effectively making it an empty queue.

Task 2: Process Scheduling using Round Robin

In this exercise, you will simulate a simplified process scheduling system based on the OS 5-state model using C++ and queues. The goal is to implement a basic process scheduling algorithm where processes are classified into five different queues (New, Ready, Running, Blocked, and Terminated) on their states. The scheduling algorithm we will be using is Round Robin.

Instructions:

1. Implement a `**Process**` class that represents a process with the following attributes:

   - int processID: A unique identifier for the process.
   - int state: An integer representing the process state (New, Ready, Running, Blocked, or Terminated).
   - int executionTime: The amount of time the process needs to execute.

- bool onceBlocked: Boolean which will be true if a process ever enters a blockedqueue.

2. Create five queues (e.g., newQueue, readyQueue, runningQueue, blockedQueue, and terminatedQueue) to manage processes in different states. You have to use the **Queue** class you implemented in the previous question for this purpose. Each Node of the queue will be of Process type.

3. Write a function `**simulateProcessScheduling**` that performs the following tasks:

- Generate a random number between 1 and 5 to determine the execution time of a newly created process.
- Newly created processes will be enqueued into the new Queue.
- The processes will initially move from the new queue to the ready queue.
- The process at the beginning of the ready queue will move into its running state i.e. running queue.
- Initialize the Time quantum = 2, then process in the running process will process for 2 seconds.
- After every two seconds of execution, a random number will be generated and if it's a multiple of 10 only then will a process from the running state will move into the blocked state i.e. blocked queue. This happens when a process needs an I/O device.
- A random number will be generated and if it's a multiple of 100 only then will a process from the blocked state will move into the ready state i.e. ready queue
- Once the execution time is over the process will go in its terminated state i.e. terminated queue.
- Continue the simulation until all processes are in the Terminated state.

4. Display the state of each process (New, Ready, Running, Blocked, Terminated) at each step of the simulation.

Additional Guidelines:

- You may use the C++ `rand()` function for random number generation.
- Ensure that processes are dequeued and enqueued correctly as they move between states.

Task 3:

Palindrome means in any order you read ‘from start to end’ or ‘from end to start’ it must be the same.
Implement a queue using a linked list and write a function isPalindrome() to determine if a given sequence of characters is a palindrome. **For example:**

- r->a->d->a->r is palindrome.
- a->p->p->l->e is not a palindrome.

Note: You must use the Queue class for this function.

Bonus Task:

Title: C++ Flood Fill Algorithm Using a Queue

Objective:

Design and implement a C++ program to perform the Flood Fill algorithm using a queue-based approach, without using `pair` or vectors. This task aims to evaluate your ability to design classes and functions for a well-structured solution.

Problem Description:

You are required to create a C++ program that implements the Flood Fill algorithm. The program should read a 2D matrix representing an image, and apply the Flood Fill operation to change the color of connected cells. This task involves designing classes and functions to maintain a structured codebase.

Requirements:

1. Class Design:
   - Design a `FloodFiller` class that encapsulates the Flood Fill logic.
   - The `FloodFiller` class should have appropriate member variables and functions to carry out the task.
   - Use the object-oriented approach to structure your code.

2. Functionality:
   - Implement a function within the `FloodFiller` class to read the matrix from the user.
   - Implement the Flood Fill algorithm within the class based on the provided guidelines.
   - Provide methods to set the start node, target color, and replacement color.
   - Implement a function to perform the Flood Fill operation.
   - Display the updated matrix using a class method.

3. Input and Error Handling:
   - Prompt the user for input regarding matrix dimensions, elements, start node, target color, and replacement color.
   - Perform input validation to handle cases where the input is out of bounds or invalid.

4. Queue-Based Approach:
   - Implement the Flood Fill algorithm using a queue-based approach as described in the previous solution.

- Ensure that the queue does not use the `pair` data structure.

5. Structured Code:
   - Organize your code into appropriate classes and functions to promote readability and reusability.
   - Comment your code to explain the logic and any non-trivial parts of your implementation.

Example Usage:

```
FloodFiller filler;
filler.readMatrix();
filler.setStartNode(2, 3);
filler.setColors(2, 4);
filler.performFloodFill();
filler.displayUpdatedMatrix();
```

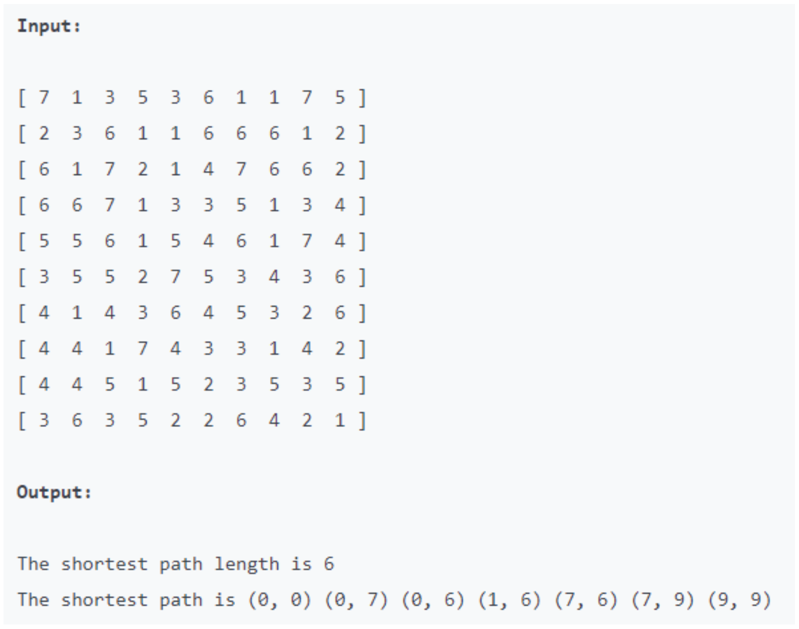
Guidelines:
1. Start by designing the `FloodFiller` class with member variables and methods.
2. Implement the Flood Fill algorithm within the class using a queue.
3. Create functions for input, error handling, and displaying the matrix.
4. Test your program with various matrices and scenarios, including boundary cases.

Bonus (Optional):
1. Implement additional error handling for invalid user input.
2. Provide an option to save the modified matrix to a file for further analysis.
3. Optimize your algorithm to minimize stack usage in recursion (e.g., using tail recursion or an iterative approach).

Home Task (Non-graded):

1. Write a C++ program to reverse the elements of a queue using recursion.
2. Write a C++ program to sort the elements of a queue.
3. Write a C++ program to find the median of all elements of a queue.
4. Write a C++ program to remove all duplicate elements from a queue.
5. Write a C++ program to concatenate two queues
6. Write a C++ program to copy one queue to another.
7. Write a C++ program to find the top and bottom elements of a queue.
8. Given a positive number n, efficiently generate binary numbers between 1 and n using the queue data structure in linear time.
   For example, for n = 16, the binary numbers are:
   1 10 11 100 101 110 111 1000 1001 1010 1011 1100 1101 1110 1111 10000

9. Given an N × N matrix of positive integers, find the shortest path from the first cell of the matrix to its last cell that satisfies given constraints.
   We are allowed to move exactly k steps from any cell in the matrix where k is the cell's value, i.e., from a cell (i, j) having value k in a matrix M, we can move to (i+k, j), (i-k, j), (i, j+k), or (i, j-k). The diagonal moves are not allowed.



```
Input:

[ 7  1  3  5  3  6  1  1  7  5 ]
[ 2  3  6  1  1  6  6  6  1  2 ]
[ 6  1  7  2  1  4  7  6  6  2 ]
[ 6  6  7  1  3  3  5  1  3  4 ]
[ 5  5  6  1  5  4  6  1  7  4 ]
[ 3  5  5  2  7  5  3  4  3  6 ]
[ 4  1  4  3  6  4  5  3  2  6 ]
[ 4  4  1  7  4  3  3  1  4  2 ]
[ 4  4  5  1  5  2  3  5  3  5 ]
[ 3  6  3  5  2  2  6  4  2  1 ]

Output:

The shortest path length is 6
The shortest path is (0, 0) (0, 7) (0, 6) (1, 6) (7, 6) (7, 9) (9, 9)
```

10.