# 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:
   * **CopyConstructor**: Implement a copy constructor that take an object of queue and copies it to the queue.
   * **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:**

**Multi-Level Feedback Queue Scheduler (Node-Based Queue**)

Objective:

Design and implement a multi-level feedback queue scheduler in C++ using node-based queues. This scheduler will manage multiple queues with different priorities and use the First-Come, First-Served (FCFS) scheduling algorithm within each queue. The task will involve creating classes and functions to simulate the scheduling process.

Task Description:

**Process Class:**

* + Create a Process class to represent individual processes. Each process should have the following attributes:
    - Process ID
    - Priority (0 = high, 1 = medium, 2 = low)
    - Burst Time (the time it takes to execute the process)
    - Next pointer (for linking processes in the queue)

**CustomQueue Class:**

* + Implement a CustomQueue class for node-based queues. Each queue node should contain a Process object and a next pointer to the next node.
  + Include the following member functions in the CustomQueue class:
    - isEmpty(): Check if the queue is empty.
    - enqueue(Process\* process): Add a process to the queue.
    - dequeue(): Remove and return the process at the front of the queue.

**MultiLevelFeedbackQueueScheduler Class:**

* + Create a MultiLevelFeedbackQueueScheduler class to manage the scheduling process.
  + Include the following member functions in the MultiLevelFeedbackQueueScheduler class:
    - addProcess(Process process): Add a process to the appropriate priority queue based on its priority.
    - executeProcesses(): Implement the scheduling logic using FCFS within each priority queue and promote processes after a certain number of executions.

**Task 3:**

Design a C++ class called ZigzagIterator that takes two queues of integers as input and allows for zigzag iteration through these queues. Implement this iterator using the provided queues. Your class should have the following methods:

* ZigzagIterator(queue<int>& q1, queue<int>& q2): A constructor that initializes the iterator with two input queues, q1 and q2.
* bool hasNext(): A method that returns true if there are more elements to iterate, and false otherwise.
* int next(): A method that returns the next element in the zigzag order. If there are no more elements to iterate, return -1.

The ZigzagIterator should iterate through the two queues in the following manner:

* Initially, start with the front element of q1.
* Then, alternate between q1 and q2, one element at a time.
* If one of the queues is empty, continue iterating through the non-empty queue until all elements are exhausted.

For example, if the input queues are q1 = {1, 2} and q2 = {3, 4, 5, 6}, the zigzag iteration should yield the sequence: 1, 3, 2, 4, 5, 6.

**Bonus Task:**

Implementing a Message Buffering System in C++

Background: In this lab task, you will design and implement a basic message buffering system in C++. This system will simulate message production and consumption with minimal synchronization mechanisms.

Requirements:

* Implement a message buffering system that includes the following components:
  + Message struct with fields for id (an integer) and content (a string).
  + MessageQueue class for message storage with methods to Enqueue and Dequeue messages.
  + MessageProducer function to generate and enqueue messages with random delays.
  + MessageConsumer function to dequeue and process messages with random delays.
* Simulate message generation in the MessageProducer function. The producer should generate and enqueue messages with unique IDs (starting from 1) and content like "Message 1", "Message 2", and so on. Introduce random delays for each message generation, emulating message arrival at variable rates.
* Simulate message processing in the MessageConsumer function. The consumers should dequeue and process messages from the MessageQueue. Introduce random delays for each message processing, emulating variable processing times.
* Implement a basic form of message prioritization based on the order of enqueuing. Older messages should be processed before newer ones.

**Instructions:**

* Implement the Message struct with appropriate fields.
* Implement the MessageQueue class, ensuring that it can safely enqueue and dequeue messages without data corruption or race conditions.
* Implement the MessageProducer function to generate and enqueue messages with random delays.
* Implement the MessageConsumer function to dequeue and process messages with random delays.
* Ensure that messages are processed in the order they were enqueued.

**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

1. 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.

