Data Structures: Queue Applications

Practice Questions

Question 1: Task Scheduler Using Queue

Problem Statement

Design and implement a **Task Scheduler** system using a Queue data structure. The task scheduler should manage tasks that arrive in a queue and process them in **First-In-First-Out (FIFO)** order.

Requirements

Your implementation should include:

1. **Task Class**: Create a Task class with the following properties:
   - `id` (unique identifier)
   - `name` (task description)
   - `priority` (integer value)
   - `executionTime` (time required to complete the task in milliseconds)
   - `createdTime` (timestamp when task was added)
2. **TaskScheduler Class**: Implement a TaskScheduler with the following operations:
   - `enqueue(task)`: Add a new task to the queue
   - `dequeue()`: Remove and return the next task from the queue
   - `peek()`: View the next task without removing it
   - `isEmpty()`: Check if the queue has any tasks
   - `getQueueLength()`: Return the number of tasks in the queue
   - `processAllTasks()`: Execute all tasks in the queue sequentially
3. **Processing Logic**:
   - Each task should execute for its specified `executionTime`
   - Display when each task starts and completes
   - Track total execution time for all tasks
   - Handle empty queue scenarios gracefully

Example Usage

Input:
Task 1: Send email notification (1000ms)
Task 2: Generate report (2000ms)
Task 3: Backup database (1500ms)

Expected Output:
[00:00] Task 1 started
[01:00] Task 1 completed
[01:00] Task 2 started
[03:00] Task 2 completed

[03:00] Task 3 started
[04:30] Task 3 completed
Total execution time: 4500ms

Follow-up Questions

a) What is the **time complexity** of enqueue and dequeue operations in your implementation?

b) How would you modify the scheduler to support **priority-based task execution** instead of FIFO?

c) What are the **advantages and disadvantages** of using a Queue vs. a Stack for this scheduler?

Question 2: Circular Queue Implementation and Analysis

Problem Statement

Implement a **Circular Queue** data structure and analyze its performance characteristics compared to a standard linear queue. Provide a complete implementation in your preferred language and solve the following scenario.

Part A: Implementation Requirements

Implement a Circular Queue with the following operations:

1. **Constructor**: Create a circular queue with a specified maximum size
2. `enqueue(element)`: Add an element to the rear of the queue
3. `dequeue()`: Remove and return the element from the front
4. `isFull()`: Check if the queue is at maximum capacity
5. `isEmpty()`: Check if the queue is empty
6. `peek()`: View the front element without removing it
7. `displayQueue()`: Print all elements in the queue

Part B: Scenario Analysis

You have a **Printer Job Queue** system with the following specifications:

- Maximum of 10 print jobs can be queued simultaneously
- Jobs arrive at: T=0, T=2, T=5, T=8, T=12, T=15, T=18, T=20, T=22, T=25 seconds
- Each job takes 3 seconds to complete
- After the 7th job is dequeued, two new jobs arrive at T=17 and T=19
- A job can be removed before execution if it was submitted in error

Tasks

a) **Trace through the queue operations** showing the state of the circular queue after each operation (enqueue/dequeue).

b) **Calculate the maximum queue length** that occurs during execution.

c) **Identify any queue overflow scenarios** and explain how you would handle them.

d) **Compare memory usage** between a linear queue and circular queue for this scenario.

Implementation Constraints

- Use array-based implementation (not linked list)
- Include boundary condition handling
- Provide clear comments in your code
- Time complexity should be O(1) for all operations

Follow-up Questions

a) Under what circumstances is a circular queue **more efficient** than a linear queue?

b) How would you implement a **dynamic circular queue** that resizes when full?

c) What modifications would be needed to support **priority-based dequeue** operations?

Answer Guidelines

For both questions, your solution should include:

- Clean, well-commented source code
- Time and space complexity analysis
- Test cases demonstrating correctness
- Discussion of real-world applications
- Explanation of design decisions

Programming languages accepted: C++, Java, Python, C#, or JavaScript

---

Document Created: December 23, 2025
Level: Intermediate (2nd Year CSE Students)
Estimated Time: 3-4 hours total