Assignment no. 4

Title: Queue

Part 1 and Part 2 are mandatory.

Part 3 is bonus. You can expect part 3 type questions in the end semester exam.

- Part 1
    1. Queue Implementation:
        a. Define a constant "MAX_SIZE" to represent the maximum size of the queue.
        b. Define an integer array named "queue" with a size of "MAX_SIZE" to represent the queue.
        c. Declare two integer variables named "front" and "rear" and initialize them to -1. These variables will track the front and rear of the queue, respectively.
        d. Create functions for the following queue operations:
            - "enqueue" - adds an element to the rear of the queue.
            - "dequeue" - removes an element from the front of the queue.
            - "isEmpty" - checks if the queue is empty.
            - "isFull" - checks if the queue is full.
            - "display" - prints the elements of the queue.
        e. Implement the functions mentioned above to perform the respective operations.
    2. Queue Operations:
        a. Create an empty queue using the defined queue implementation.
        b. Enqueue some integer values into the queue.
        c. Print the elements of the queue using the "display" function.
        d. Dequeue elements from the queue and print the dequeued elements.
        e. Check if the queue is empty and print the result.
        f. Enqueue additional elements into the queue and print the updated queue.

- Part 2
    1. Circular Queue: Implement a circular queue to optimize memory utilization. Modify the enqueue and dequeue functions to handle the circular behavior correctly.
    2. Queue using Linked List: Implement a queue data structure using linked lists instead of an array. Create functions to enqueue, dequeue, and display elements in the linked list-based queue.
    3. Priority Queue: Implement a priority queue, where each element has an associated priority (or assume it's value as it's priority). Elements with higher priority are dequeued before elements with lower priority.

- Part 3
    1. Print job management in a printer queue: In this scenario, we will simulate the management of print jobs in a printer queue using the queue data structure. A printer queue is a typical real-world application of a queue where multiple print jobs are received from various users and need to be processed in the order they arrive.
        a. Define a structure named "PrintJob" with the following members:
            - "jobID" (integer) to represent the unique ID of the print job.
            - "jobName" (string) to store the name of the print job (e.g., document name).
            - "numPages" (integer) to store the number of pages in the print job.
        b. Implement the Printer Queue:
            - Create a queue data structure using the queue implementation provided in the previous lab assignment.
            - The queue will store "PrintJob" structures.
        c. Simulate Print Job Arrival:
            - Prompt the user to enter the details of a print job, including the job ID, job name, and number of pages.
            - Create a "PrintJob" structure with the entered details.
            - Enqueue the created "PrintJob" structure into the printer queue.
        d. Print Job Processing:
            - Simulate the printing process by dequeueing print jobs from the printer queue.
            - Print the details of the dequeued print job (e.g., job ID, job name, and number of pages).
            - Simulate the printing process by introducing a delay to mimic the time it takes to print a job.
        e. Continuously Receive Print Jobs:
            - Implement a loop that allows users to enter multiple print jobs.
            - Each entered print job should be enqueued into the printer queue and processed in the order they are received.
        f. Optional: Priority Printing:
            - Implement a priority queue for print jobs, where the print jobs with fewer pages have higher priority.
            - Modify the print job processing to dequeue the print jobs with the highest priority (i.e., fewer pages) first.
        g. Note: To simulate the printing process, you can introduce a delay using the "sleep" function (for POSIX systems) or "Sleep" function (for Windows systems). The delay can be a few seconds to mimic the time it takes to print a job. For simplicity, you can use the standard C library's **rand ()** function to generate a random job ID.
        h. Ensure that you thoroughly test your simulation by enqueuing multiple print jobs with different page counts and verifying that they are processed in the correct order.
    2. Task scheduling in an operating system: In this scenario, we will simulate task scheduling in an operating system using the queue data structure. Task scheduling is a fundamental component of any operating system, where multiple processes and threads compete for execution time on the CPU. The operating system's task scheduler ensures that tasks are executed efficiently and fairly, considering factors like priority and time slice.
        a. Define the Process Structure: a) Define a structure named "Process" with the following members:
            - "processID" (integer) to represent the unique ID of the process.
            - "processName" (string) to store the name of the process.
            - "priority" (integer) to indicate the priority of the process (higher value means higher priority).
            - "burstTime" (integer) to store the time required for the process to complete its execution.
        b. Implement the Task Scheduler Queue:
            - Create a queue data structure using the queue implementation provided in the previous lab assignment.

- The queue will store "Process" structures.

c. Simulate Task Arrival:
   - Prompt the user to enter the details of a process, including the process ID, process name, priority, and burst time.
   - Create a "Process" structure with the entered details.
   - Enqueue the created "Process" structure into the task scheduler queue.

d. Task Execution:
   - Simulate the task execution process by dequeueing processes from the task scheduler queue.
   - Print the details of the dequeued process (e.g., process ID, process name, priority, and burst time).
   - Simulate the execution of the process by introducing a delay to mimic the time it takes to execute a task.

e. Prioritizing Tasks:
   - Modify the task scheduling logic to prioritize processes based on their priority values.
   - Processes with higher priority should be dequeued and executed before processes with lower priority.

f. Optional: Round-Robin Scheduling:
   - Implement a round-robin scheduling algorithm, where each process is given a fixed time slice for execution before being preempted.
   - Modify the task scheduler to switch between processes after the time slice expires.

g. Note: To simulate the execution of tasks, you can introduce a delay using the "sleep" function (for POSIX systems) or "Sleep" function (for Windows systems). The delay can be a few seconds to mimic the time it takes for a process to execute.

h. Ensure that you thoroughly test your simulation by enqueuing multiple processes with different priorities and burst times and verifying that they are executed in the correct order based on their priorities.

3. Select a real-world scenario from around your life where a queue data structure is used and implement it. Examples include:
   a. Supermarket checkout lanes
   b. Traffic management at intersections
   c. Call center waiting queues
   d. Bank Customer Service Queue