DATA STRUCTURE

Chapter 4: Queues

Exam-Oriented Problems & Solutions — Set 5

This is the fifth set of fourteen exam-oriented problems on the Queues chapter, with entirely new scenarios (no repeats from Sets 1–4). It covers a naive shifting-array queue and its complexity trade-offs, a circular singly linked-list queue using only a single REAR pointer, Shortest-Job-First scheduling and priority-queue stability, three further algorithm-design problems (stack using two queues, merging two sorted queues, summing a queue without altering it), analytical questions on cache locality, real-time predictability and space trade-offs, and two applications — a keyboard input buffer and an asynchronous task scheduler.

Section A: Array (Linear) Queue — A Shifting Implementation

Problem 1. A naive array-based queue keeps FRONT permanently fixed at index 0 by shifting all remaining elements one position to the left on every dequeue (rather than just incrementing a FRONT pointer). Given QUEUE = [10, 20, 30, 40, 50] (capacity N = 6, one free slot at the end), trace: dequeue(), enqueue(60), dequeue(), showing the shifting steps.

Solution:

dequeue(): remove QUEUE[0] = 10. Shift the remaining 4 elements one position left: 20→index0, 30→index1, 40→index2, 50→index3. Queue = [20, 30, 40, 50] (length 4). Returned value: 10.

enqueue(60): append at the next free position (index 4, since FRONT is always 0). Queue = [20, 30, 40, 50, 60] (length 5).

dequeue(): remove QUEUE[0] = 20. Shift the remaining 4 elements left: 30→index0, 40→index1, 50→index2, 60→index3. Queue = [30, 40, 50, 60] (length 4). Returned value: 20.

Answer: Final queue = [30, 40, 50, 60]. Every dequeue in this scheme costs time proportional to the number of remaining elements, since all of them must be shifted left by one position.

Problem 2. Compare the shifting-array queue of Problem 1 with the FRONT-pointer method (used throughout the main chapter notes, where FRONT is simply incremented on dequeue instead of shifting elements). State the time complexity of enqueue and dequeue for each, and explain the trade-off each method makes.

Solution:

Shifting method (Problem 1): enqueue is O(1) (a new element is simply appended at the next free position), but dequeue is O(n), where n is the number of elements remaining, because every one of them must be shifted left by one position to keep FRONT fixed at index 0.

FRONT-pointer method (main notes): both enqueue and dequeue are O(1), since dequeue is done purely by incrementing FRONT with no data movement at all.

Answer: The FRONT-pointer method is strictly faster for dequeue (O(1) vs O(n)), which is why it is the standard approach — but it pays for this by leaving used slots before FRONT permanently unusable (the false-overflow problem), which the shifting method avoids since it always keeps all live elements packed starting at index 0. In practice, the false-overflow problem is solved separately with a circular queue, making the FRONT-pointer approach superior overall.

Section B: Circular Singly Linked-List Queue

Problem 3. A queue is implemented as a circular singly linked list using only ONE external pointer, REAR (REAR.next always gives the FRONT node). The algorithms are:

```
ENQUEUE(x):
    Create new node with data = x
    If REAR is NULL (empty queue):
        newNode.next = newNode; REAR = newNode
    Else:
        newNode.next = REAR.next     // link to old FRONT
        REAR.next = newNode          // old REAR points to new node
        REAR = newNode                // update REAR

DEQUEUE():
    If REAR is NULL: report UNDERFLOW
    front = REAR.next
    If front = REAR (only one node): REAR = NULL
    Else: REAR.next = front.next     // unlink the front node
    Return front.data
```

Solution:

enqueue(10): REAR is NULL → new node(10).next = itself; REAR = node(10). The list is a single self-pointing circular node; FRONT = REAR.next = node(10).

enqueue(20): new node(20).next = REAR.next = node(10) [old FRONT]; REAR.next (node 10's link) = node(20); REAR = node(20). List (circularly): 10 → 20 → 10 ... ; FRONT = REAR.next = node(10).

enqueue(30): new node(30).next = REAR.next = node(10) [FRONT]; REAR.next (node 20's link) = node(30); REAR = node(30). List: 10 → 20 → 30 → 10 ... ; FRONT = node(10).

dequeue(): front = REAR.next = node(10). front ≠ REAR(30), so REAR.next = front.next = node(20) — node(10) is unlinked. List is now: 20 → 30 → 20 ... ; REAR = node(30), FRONT = REAR.next = node(20). Returned value: 10.

Answer: After these operations, the circular list holds 20 and 30 (front to rear), with REAR pointing at node(30) and FRONT obtained as REAR.next = node(20) — demonstrating that a single pointer is enough to manage both ends.

Problem 4. Explain why maintaining only the REAR pointer (as in Problem 3) is sufficient to access both ends of the queue in O(1) time, without needing a separate FRONT pointer.

Solution:

Because the linked list is circular, the node immediately after REAR in the chain (REAR.next) is always the very first node that was inserted and has not yet been removed — that is, precisely the FRONT of the queue, by construction of the enqueue algorithm, which always inserts new nodes just before the current FRONT and then moves REAR to the new node.

Answer: Since REAR.next always equals FRONT in this circular structure, FRONT is obtainable in O(1) time directly from REAR without needing a dedicated FRONT pointer — saving the memory and bookkeeping of a second pointer while still supporting O(1) access to both ends of the queue.

Section C: Priority Queue Applications

Problem 5. Shortest Job First (SJF, non-preemptive) CPU scheduling can be implemented with a priority queue where priority = burst time (a shorter burst time means higher priority). Given processes, all arriving

at t = 0: P1 (burst 6), P2 (burst 2), P3 (burst 8), P4 (burst 3), find the execution order and the average waiting time.

Solution:

Order by ascending burst time (shortest first): P2 (2), P4 (3), P1 (6), P3 (8).

Waiting times: P2 starts immediately → wait = 0. P4 waits for P2's burst (2) → wait = 2. P1 waits for P2 + P4 (2+3=5) → wait = 5. P3 waits for P2 + P4 + P1 (2+3+6=11) → wait = 11.

Average waiting time = (0 + 2 + 5 + 11) / 4 = 18 / 4 = 4.5.

Answer: Execution order: P2, P4, P1, P3. Average waiting time = 4.5 time units.

Problem 6. Explain the difference between a "stable" priority queue (ties broken by arrival order, like FIFO among equal-priority elements) and an "unstable" one. State which of the two standard implementations — an unsorted array/list scanned for the minimum, or a binary heap — is naturally stable, and briefly justify your answer.

Solution:

A stable priority queue always serves two equal-priority elements in the same relative order in which they were inserted — the earlier-arriving one first, exactly as a plain FIFO queue would treat them. An unstable priority queue makes no such guarantee; among equal-priority elements, the service order may not match arrival order.

An unsorted array/list, if its linear scan for the minimum is implemented to keep the FIRST element found on a tie (a natural left-to-right scan order), is naturally stable, since it always encounters earlier-inserted equal elements before later ones.

A binary heap is generally NOT naturally stable: its bubble-up and bubble-down operations can freely reorder equal-priority elements relative to one another as insertions and deletions reshuffle the heap's internal array, so the relative order of equal-priority elements is not preserved by the standard heap algorithms.

Answer: The unsorted array/list scan can be made naturally stable; the binary heap is generally not — achieving stability in a heap typically requires attaching an extra insertion-sequence number to each element and using it as a tiebreaker in comparisons.

Section D: Algorithm Design Problems

Problem 7. Write an algorithm to implement a STACK using two queues, Q1 and Q2 (the reverse of implementing a queue with two stacks), and trace it for the operations push(1), push(2), push(3), pop().

Solution:

```
PUSH(x):
    ENQUEUE(Q2, x)
    While Q1 is not empty:
        ENQUEUE(Q2, DEQUEUE(Q1))
    Swap the names/roles of Q1 and Q2

POP():
    Return DEQUEUE(Q1)
```

push(1): enqueue 1 into Q2 → Q2=[1]. Q1 is empty, nothing to move. Swap → Q1=[1], Q2=[ ].

push(2): enqueue 2 into Q2 → Q2=[2]. Move Q1's contents into Q2: dequeue 1 from Q1, enqueue into Q2 → Q2=[2,1]; Q1=[ ]. Swap → Q1=[2,1], Q2=[ ].

push(3): enqueue 3 into Q2 → Q2=[3]. Move Q1(=[2,1]) into Q2: dequeue 2→enqueue → Q2=[3,2]; dequeue 1→enqueue → Q2=[3,2,1]. Q1=[ ]. Swap → Q1=[3,2,1], Q2=[ ].

pop(): dequeue the front of Q1 → returns 3, correctly the most recently pushed element (LIFO behaviour). Q1=[2,1].

Answer: pop() correctly returns 3, confirming the two-queue technique reproduces stack (LIFO) behaviour: every push re-inserts the new element ahead of all existing ones by rebuilding the queue, so the front of Q1 always holds the most recently pushed item.

Problem 8. Write an algorithm to merge two already ascending-sorted queues, Q1 and Q2, into a single ascending-sorted queue, using only enqueue, dequeue and peek (no random access). Trace it on Q1 = [2, 5, 8] and Q2 = [1, 3, 9] (front to rear, both ascending).

Solution:

```
MERGE_SORTED_QUEUES(Q1, Q2):
    Create empty queue RESULT
    While Q1 is not empty AND Q2 is not empty:
        If PEEK(Q1) <= PEEK(Q2):
            ENQUEUE(RESULT, DEQUEUE(Q1))
        Else:
            ENQUEUE(RESULT, DEQUEUE(Q2))
    While Q1 is not empty: ENQUEUE(RESULT, DEQUEUE(Q1))
    While Q2 is not empty: ENQUEUE(RESULT, DEQUEUE(Q2))
    Return RESULT
```

peek(Q1)=2, peek(Q2)=1: 2 ≤ 1 is false → dequeue Q2 (1) → RESULT=[1]. Q2=[3,9]

peek(Q1)=2, peek(Q2)=3: 2 ≤ 3 → dequeue Q1 (2) → RESULT=[1,2]. Q1=[5,8]

peek(Q1)=5, peek(Q2)=3: 5 ≤ 3 is false → dequeue Q2 (3) → RESULT=[1,2,3]. Q2=[9]

peek(Q1)=5, peek(Q2)=9: 5 ≤ 9 → dequeue Q1 (5) → RESULT=[1,2,3,5]. Q1=[8]

peek(Q1)=8, peek(Q2)=9: 8 ≤ 9 → dequeue Q1 (8) → RESULT=[1,2,3,5,8]. Q1=[ ]

Q1 is now empty, main loop ends. Q2 still holds [9] → append it: RESULT=[1,2,3,5,8,9].

Answer: Final merged queue = [1, 2, 3, 5, 8, 9] — fully sorted in ascending order, obtained purely through FIFO operations. This is the same core step used inside merge sort.

Problem 9. Write an algorithm to compute the sum of all elements currently in a queue WITHOUT permanently altering it (the queue must be restored to its original state afterward), and trace it on Q = [4, 7, 2, 9] (front to rear).

Solution:

```
SUM_QUEUE(Q, n):
    sum = 0
    Create empty queue TEMP
    For i = 1 to n:
        x = DEQUEUE(Q)
        sum = sum + x
        ENQUEUE(TEMP, x)
    For i = 1 to n: ENQUEUE(Q, DEQUEUE(TEMP))   // restore original queue
    Return sum
```

dequeue 4, sum=4, TEMP=[4]. dequeue 7, sum=11, TEMP=[4,7]. dequeue 2, sum=13, TEMP=[4,7,2]. dequeue 9, sum=22, TEMP=[4,7,2,9].

Restore: enqueue back from TEMP into Q in the same order → Q = [4, 7, 2, 9], unchanged from the original.

Answer: Sum = 22, and the original queue Q = [4, 7, 2, 9] is left exactly as it was before the operation.

Section E: Analytical / Complexity Problems

Problem 10. Even though both an array-based queue and a linked-list queue offer O(1) enqueue and dequeue, explain — in terms of memory locality / cache behaviour — why the array-based version generally performs faster in practice.

Solution:

Array elements are stored in contiguous memory locations, so accessing consecutive elements (as enqueue and dequeue effectively do) benefits from spatial locality: the CPU cache can pre-fetch nearby memory efficiently, resulting in fewer cache misses.

A linked list's nodes are typically scattered across heap memory wherever they happened to be individually allocated, so following a pointer from one node to the next often causes a cache miss, since the next node's memory address bears no particular relation to the current one.

Answer: Although both structures share the same O(1) Big-O time complexity for enqueue and dequeue, the array-based queue tends to have a smaller constant factor in practice due to better cache locality, making it generally faster for typical workloads despite the linked list's advantage of not needing a pre-declared fixed capacity.

Problem 11. Explain why a hard real-time embedded system (e.g., a flight controller) almost always prefers a fixed-size circular buffer over a dynamically resizing queue for buffering sensor data, focusing on predictable, bounded-time behaviour rather than average-case speed.

Solution:

A hard real-time system must guarantee that every operation completes within a strict, known time bound (a deadline); missing this bound even occasionally can cause system failure.

A fixed-size circular queue's enqueue and dequeue are always O(1), doing the same small, constant amount of work on every single call, regardless of history — this makes worst-case timing easy to analyse and guarantee.

A dynamically resizing queue occasionally performs an expensive O(n) resize-and-copy operation once it becomes full. This is efficient on AVERAGE (amortized O(1)), but that occasional O(n) call could, on an unlucky occurrence, take far longer than the system's deadline allows — an unpredictable worst-case spike that is unacceptable even if rare.

Answer: A fixed-capacity circular buffer is preferred because every one of its operations has the same small, bounded worst-case cost, giving predictable timing — at the cost of needing to size it carefully in advance and handle overflow explicitly (e.g., by dropping data) rather than silently growing, unlike a resizing queue whose rare but large worst-case spikes are unacceptable in a hard real-time context.

Problem 12. For a queue holding n elements, state the space complexity of (a) an array-based queue with fixed capacity $N \geq n$, and (b) a linked-list queue. Explain a circumstance where (a) is more space-efficient than (b), and a circumstance where the reverse is true.

Solution:

(a) Array-based queue: space is O(N), the fixed declared capacity, regardless of how many of those N slots ($n \leq N$) are actually occupied at any given moment.

(b) Linked-list queue: space is O(n), growing and shrinking exactly with the number of elements currently stored, but each of the n nodes carries an additional pointer-field overhead beyond its data.

(a) is more space-efficient when the queue's size stays consistently close to its declared capacity N (little wasted pre-allocation) and the per-element data is small, so the linked list's per-node pointer overhead would be proportionally large by comparison.

(b) is more space-efficient when the queue's actual size fluctuates greatly and is often much smaller than any fixed capacity that would need to be pre-declared for the array to avoid overflow — the array would then waste a large amount of reserved-but-unused space, whereas the linked list only ever uses space proportional to what is currently stored.

Answer: Array: O(N) fixed; Linked list: O(n) exact. The array wins when usage stays near full capacity with small elements; the linked list wins when the size varies widely and unpredictably.

Section F: Application-Based Problems

Problem 13. A keyboard input buffer is a fixed-size circular queue with capacity 8 characters. The system only processes (removes and displays) characters after all typing has stopped — nothing is removed while typing is in progress — and any new character typed once the buffer is full is simply dropped. A user types "PROGRAMMING" (11 characters) without pausing. Determine which characters, if any, are lost.

Solution:

"PROGRAMMING" has 11 characters, in order: P, R, O, G, R, A, M, M, I, N, G.

The first 8 typed characters — P, R, O, G, R, A, M, M — fill the buffer completely (8/8), since nothing is removed while typing continues.

The 9th, 10th and 11th characters — I, N, G — arrive to find the buffer already full, with no processing having occurred to free a slot, so each is dropped.

Answer: The characters 'I', 'N', and 'G' (the last 3 typed) are lost. The buffer retains only the first 8 characters typed: "PROGRAMM".

Problem 14. An asynchronous task scheduler (such as a JavaScript-style event loop) processes tasks strictly in FIFO order, running each task fully to completion before starting the next ("run-to-completion"). Three tasks are already enqueued when the scheduler starts: TaskA (5 ms), TaskB (2 ms), TaskC (4 ms), enqueued in that order. Find the completion time of each task, and explain why TaskB — despite needing only 2 ms and being ready from the very start — still finishes relatively late.

Solution:

TaskA runs first, from t=0 to t=5 (5 ms), completing at t=5.

TaskB runs next, from t=5 to t=7 (2 ms), completing at t=7.

TaskC runs last, from t=7 to t=11 (4 ms), completing at t=11.

Answer: Completion times: TaskA = 5 ms, TaskB = 7 ms, TaskC = 11 ms. Even though TaskB was ready to run from t=0 and only needed 2 ms of work, the strict FIFO ordering combined with run-to-completion execution forces it to wait for the entirety of TaskA's 5 ms runtime first — a single long-running task at the front of the queue delays every task behind it, regardless of how short those later tasks are. This is exactly why a long-running synchronous task can cause noticeable delays ("jank") in a real event-loop-based system.