DATA STRUCTURE

Chapter 4: Queues

Exam-Oriented Problems & Solutions — Set 4

This is the fourth set of fourteen exam-oriented problems on the Queues chapter, with entirely new scenarios (no repeats from Sets 1–3). It covers dynamic (resizing) array queues and amortized analysis, the two-queues-in-one-array representation, the k-largest-elements and multilevel priority queue applications, three further algorithm-design problems (dequeue using a single stack via recursion, comparing two queues, left-rotation), analytical questions on memory shrinkage, thread-safety and heap-vs-sorted-list trade-offs, and two applications — FCFS disk scheduling and publish–subscribe message ordering.

Section A: Array (Linear) Queue — Resizing / Dynamic Array

Problem 1. A dynamic array queue starts with capacity N = 4 (indices 0–3), FRONT = REAR = -1. Whenever an enqueue is attempted on a full array, the array is resized by doubling its capacity: a new array of size 2N is allocated, the existing logical elements are copied into it starting at index 0 (so FRONT becomes 0 again), and only then is the new item inserted. Trace: enqueue(1), enqueue(2), enqueue(3), enqueue(4), enqueue(5).

Solution:

enqueue(1): FRONT=0, REAR=0. Array (cap 4) = [1, _, _, _]

enqueue(2): REAR=1. Array = [1, 2, _, _]

enqueue(3): REAR=2. Array = [1, 2, 3, _]

enqueue(4): REAR=3 = N-1. Array = [1, 2, 3, 4] — now full (capacity 4).

enqueue(5): array is full → RESIZE: allocate a new array of capacity 8; copy the 4 logical elements (1,2,3,4) into indices 0–3 of the new array; reset FRONT=0, REAR=3. Then perform the pending insertion: REAR=4; new array[4]=5.

Answer: Final array (capacity 8) = [1, 2, 3, 4, 5, _, _, _]; FRONT = 0, REAR = 4. The resize step transparently gave the queue room to keep growing without reporting a false overflow.

Problem 2. For the dynamic array queue of Problem 1, which doubles its capacity every time it becomes full, explain (using the standard "doubling" argument) why the amortized time complexity of enqueue is still O(1), even though an individual resizing enqueue costs O(n) to copy n elements.

Solution:

Consider n total enqueue operations performed from an initially small array. Resizes happen only when the array has previously filled to sizes 1, 2, 4, 8, 16, ..., i.e., at powers of 2, so the total copying work across all resizes is 1 + 2 + 4 + 8 + ... + n/2, which is a geometric series.

This geometric series sums to less than 2n (it is bounded by 2·(n/2) = n plus lower terms, all together strictly less than 2n).

So the total cost of all the resizing copy-work across n enqueue operations is O(n), on top of the O(n) cost of the n individual O(1) insertions themselves — giving a combined total cost of O(n) for n operations.

Answer: Since the total cost of n enqueue operations is O(n), the average (amortized) cost per enqueue is O(n)/n = O(1). A small fraction of enqueues are individually expensive (O(n) during a resize), but they are rare enough that the cost, spread evenly over all operations, works out to constant time per operation on average.

Section B: Circular / Shared-Array Queue Representation

Problem 3. Two separate queues, Queue1 and Queue2, are stored within a single shared array of size N = 10 (indices 0–9): Queue1 grows forward from index 0 (FRONT1 = 0, next free slot tracked by REAR1, initially -1), and Queue2 grows backward from index 9 (next free slot tracked by REAR2, initially 10). Insert A, B, C into Queue1 (in that order) and X, Y into Queue2 (in that order). Show the resulting array.

Solution:

Insert into Queue1 (forward from index 0): A → index 0, REAR1=0; B → index 1, REAR1=1; C → index 2, REAR1=2.

Insert into Queue2 (backward from index 9): X → index 9, REAR2=9; Y → index 8, REAR2=8.

Answer: Array (index 0 to 9) = [A, B, C, _, _, _, _, _, Y, X]. Indices 3 to 7 (5 slots) remain free between the two queues, which can still grow toward each other.

Problem 4. Continuing from Problem 3, five more elements D, E, F, G, H are inserted into Queue1 (in that order), with nothing further inserted into Queue2. Show the final array, and state whether any further insertion is possible into either queue without first removing an element.

Solution:

D → index 3, REAR1=3; E → index 4, REAR1=4; F → index 5, REAR1=5; G → index 6, REAR1=6; H → index 7, REAR1=7.

Now Queue1's next free slot would be index 8 (REAR1+1=8), but index 8 already holds Y (the last element of Queue2) — the two queues have met with no gap remaining.

Answer: Final array = [A, B, C, D, E, F, G, H, Y, X] — all 10 slots are occupied. No further insertion is possible into either Queue1 or Queue2 (overflow) until an element is dequeued from one of them to free a slot.

Section C: Priority Queue Applications

Problem 5. Using a min-heap of size at most k = 3, find the 3 largest elements seen so far in the stream 5, 15, 10, 20, 3, 8 (process the elements in this order): insert into the heap while its size is below k; once the heap holds k elements, for each new value x, if x is greater than the heap's minimum, remove the minimum and insert x — otherwise discard x. Trace the heap after processing every value and state the final 3 largest elements.

Solution:

x=5: heap size 0 < 3 → insert. Heap = [5]

x=15: size 1 < 3 → insert. Heap = [5, 15]

x=10: size 2 < 3 → insert. Heap = [5, 15, 10] (heap-ordered: root 5 ≤ both children).

x=20: size = 3 = k. 20 > heap-min (5) → remove-min (5): move last element (10) to root, heap becomes [10, 15]; heapify-down: child 15 ≥ 10, no swap needed. Insert 20: append → [10, 15, 20], parent 10 ≤ 20, no swap. Heap = [10, 15, 20].

x=3: 3 > heap-min (10)? No (3 < 10) → discard. Heap unchanged = [10, 15, 20]

x=8: 8 > heap-min (10)? No (8 < 10) → discard. Heap unchanged = [10, 15, 20]

Answer: Final heap = [10, 15, 20], so the 3 largest elements of the stream are {20, 15, 10}. (Check: the full stream sorted descending is 20, 15, 10, 8, 5, 3 — the top 3 are indeed 20, 15, 10.)

Problem 6. A multilevel priority scheduling system uses three separate plain FIFO queues — High, Medium, and Low — and always serves every job in the High queue before touching Medium, and every job in

Medium before touching Low. Jobs arrive in this order: J1 (Medium), J2 (High), J3 (Low), J4 (High), J5 (Medium). Determine the order in which the jobs are served.

Solution:

Sort jobs into their respective level queues in arrival order: High = [J2, J4]; Medium = [J1, J5]; Low = [J3].

Service proceeds level by level, highest first: drain High completely (J2, then J4), then drain Medium completely (J1, then J5), then drain Low (J3).

Answer: Service order: J2, J4, J1, J5, J3.

Section D: Algorithm Design Problems

Problem 7. Write a recursive algorithm to dequeue from a queue that is implemented using only ONE stack (no second stack), and trace it on a stack S holding (bottom to top) 1, 2, 3, 4 — representing a queue where 1 was enqueued first.

Solution:

```
DEQUEUE_ONE_STACK(S):
    If S has exactly one element:
        Return POP(S)
    x = POP(S)
    result = DEQUEUE_ONE_STACK(S)     // recurse to reach the bottom
    PUSH(S, x)                        // restore x on the way back up
    Return result
```

S = [1,2,3,4] (bottom to top). Call DEQUEUE_ONE_STACK(S): pop x=4, S=[1,2,3]. Recurse.

Inside recursion: pop x=3, S=[1,2]. Recurse.

Inside recursion: pop x=2, S=[1]. Recurse.

S now has exactly one element → return POP(S) = 1. S = [ ]. This is the result being carried back up.

Unwinding: push back x=2 → S=[2]; return result=1. Push back x=3 → S=[2,3]; return result=1. Push back x=4 → S=[2,3,4]; return result=1.

Answer: DEQUEUE_ONE_STACK returns 1 (correctly the earliest-enqueued/front element), and the stack is left as S=[2,3,4] (bottom to top) — the remaining elements restored in their original relative order, using only a single stack and recursion (the call stack substitutes for the second stack).

Problem 8. Write an algorithm to check whether two queues Q1 and Q2 contain exactly the same sequence of elements, without permanently altering either original queue, and trace it on Q1 = [1, 2, 3] and Q2 = [1, 2, 4] (front to rear).

Solution:

```
ARE_EQUAL(Q1, Q2):
    If SIZE(Q1) != SIZE(Q2): return FALSE
    n = SIZE(Q1); result = TRUE
    Create empty queues T1, T2
    For i = 1 to n:
        a = DEQUEUE(Q1); b = DEQUEUE(Q2)
        If a != b: result = FALSE
        ENQUEUE(T1, a); ENQUEUE(T2, b)
    For i = 1 to n:                   // restore both queues
        ENQUEUE(Q1, DEQUEUE(T1))
        ENQUEUE(Q2, DEQUEUE(T2))
    Return result
```

Sizes both 3, proceed. i=1: a=1,b=1 (equal); T1=[1],T2=[1]

i=2: a=2,b=2 (equal); T1=[1,2],T2=[1,2]

i=3: a=3,b=4 (NOT equal) → result=FALSE; T1=[1,2,3],T2=[1,2,4]

Restore: Q1 = [1,2,3] (from T1), Q2 = [1,2,4] (from T2) — both back to their original contents.

Answer: ARE_EQUAL returns FALSE (they differ at the 3rd element), and both Q1 and Q2 are left unchanged after the check.

Problem 9. Write an algorithm to left-rotate a queue by d positions (move the first d elements to the rear, preserving the relative order of all elements), using only enqueue and dequeue. Trace it on Q = [10, 20, 30, 40, 50, 60] (front to rear) with d = 2.

Solution:

```
ROTATE_LEFT(Q, d):
    For i = 1 to d:
        ENQUEUE(Q, DEQUEUE(Q))
```

i=1: dequeue 10, enqueue it back → Q = [20, 30, 40, 50, 60, 10]

i=2: dequeue 20, enqueue it back → Q = [30, 40, 50, 60, 10, 20]

Answer: Final rotated queue = [30, 40, 50, 60, 10, 20] — the first 2 elements (10, 20) have moved to the rear, in their original order, after 2 left-rotations.

Section E: Analytical / Complexity Problems

Problem 10. The dynamic array queue of Problems 1–2 in this set only ever doubles its capacity when full, and never shrinks when elements are dequeued. Explain what memory inefficiency this can cause in a long-running system, and describe the standard fix.

Solution:

If the array only grows and never shrinks, then after a burst of enqueues drives the array to some large capacity, followed by a long period where most elements are dequeued and few new ones are enqueued, the array still occupies its large peak-allocated size in memory even though it now holds very few elements — the unused capacity is never returned to the system.

In a long-running system with repeated bursts, this can lead to persistent, unnecessary memory usage (memory bloat) even during periods of low actual load.

Answer: The standard fix is to also shrink the array (commonly by halving it) once the element count falls below some threshold (e.g., 1/4 of the current capacity), mirroring the growth strategy in reverse — this keeps memory usage roughly proportional to the current number of elements while still preserving amortized O(1) enqueue and dequeue, provided the shrink threshold is chosen to avoid repeatedly growing and shrinking back and forth for the same few operations.

Problem 11. Explain, conceptually, why a plain (non-synchronized) array-based queue can produce incorrect results if two threads call enqueue at exactly the same time, and name the general technique used to prevent this in a concurrent (multi-threaded) system.

Solution:

In the unsynchronized enqueue algorithm, each thread reads the current value of REAR, computes the new position, writes the new element there, and then updates REAR. If two threads execute this concurrently, both may read the same value of REAR before either has written its update (a race condition).

As a result, both threads could write their respective new elements to the SAME array slot (overwriting one of them, silently losing data), and REAR might end up being incremented only once instead of twice — leaving REAR inconsistent with the queue's actual contents.

Answer: The general technique used to prevent this is mutual exclusion (locking): the enqueue and dequeue operations are wrapped in a lock/mutex so that only one thread can execute the critical section — reading and updating FRONT/REAR and the underlying array or linked list — at any given time, ensuring operations from different threads do not interleave incorrectly.

Problem 12. Compare a priority queue implemented as a fully sorted (descending) linked list against one implemented as a binary heap, specifically for an application that must frequently PEEK at (without removing) the top-k highest-priority elements, for various values of k. Which structure is better suited, and why?

Solution:

A sorted (descending) linked list keeps ALL elements in fully sorted order at all times, so peeking at the top-k elements simply means reading the first k nodes from the head of the list — a straightforward O(k) operation for any k, without disturbing the list.

A binary heap only guarantees that its ROOT is the single overall maximum; elements beyond the root are only partially ordered by the heap property (a heap is not fully sorted), so retrieving the top-k elements without removing them is not a simple direct read — it typically requires k repeated extract-max-and-reinsert operations, costing O(k log n), or an auxiliary traversal procedure.

Answer: The sorted linked list is better suited for an application that frequently peeks at more than just the single top element, since it offers direct O(k) access to any number of top elements at any time. The binary heap remains superior when the application mainly needs fast insertion and repeated extraction of just the single current maximum, since it trades away easy access to lower-ranked elements for O(log n) insert/extract-max instead of the sorted list's O(n) insertion cost.

Section F: Application-Based Problems

Problem 13. A disk scheduler uses First-Come-First-Served (FCFS) scheduling — a pure FIFO queue of pending track requests — serving the disk head strictly in the order requests arrive. The head starts at track 50, and the request queue (in arrival order) is: 82, 170, 43, 140, 24, 16, 190. Compute the total head movement (in number of tracks) needed to service all requests under FCFS.

Solution:

The head moves strictly in queue order: 50 → 82 → 170 → 43 → 140 → 24 → 16 → 190.

|82 − 50| = 32; |170 − 82| = 88; |43 − 170| = 127; |140 − 43| = 97; |24 − 140| = 116; |16 − 24| = 8; |190 − 16| = 174.

Total = 32 + 88 + 127 + 97 + 116 + 8 + 174.

Answer: Total head movement under FCFS = 642 tracks. (This illustrates why FCFS, though simple and fair since it is a pure FIFO queue, is not necessarily efficient — a scheduling algorithm that reorders requests, such as Shortest-Seek-Time-First, can reduce total head movement considerably.)

Problem 14. A publish–subscribe messaging system uses a single shared FIFO queue: multiple producer threads publish (enqueue) messages, and one subscriber thread receives (dequeues) them. Explain, using the FIFO property, why the subscriber is guaranteed to receive messages in exactly the order they were published, even with multiple concurrent producers — and explain what would break this guarantee if a priority queue were used instead.

Solution:

Every publish operation is an enqueue and every delivery is a dequeue on the SAME shared queue. A queue's dequeue operation always removes whichever message was enqueued earliest among those still present, regardless of which producer inserted it.

Provided all producer enqueue calls are properly serialized (using mutual exclusion, as in Problem 11 of this set, so they do not corrupt the shared structure), the relative order in which messages actually enter the queue is well-defined, and the subscriber necessarily receives them in exactly that same order.

Answer: The FIFO discipline guarantees publish-order delivery because dequeue always respects insertion order, not which producer inserted a message. If a priority queue were used instead, delivery order would be determined by each message's priority value rather than its publish time — a message published later but with higher priority could be delivered before an earlier, lower-priority message, breaking the publish-order guarantee.