DATA STRUCTURE

Chapter 4: Queues

Exam-Oriented Problems & Solutions — Set 3

This is the third set of fourteen exam-oriented problems on the Queues chapter, with entirely new scenarios (no repeats from Sets 1 or 2). It covers underflow tracing, a circular queue with REAR wrap-around counting, priority-queue-based CPU scheduling with waiting-time computation, min-heap deletion (extract-min), three new algorithm-design problems (first non-repeating character, reverse the first k elements, and a queue-ordering proof), further analytical/complexity questions, and two applications — the classic Josephus elimination problem and network packet buffering.

Section A: Array (Linear) Queue — Trace-Based Problems

Problem 1. A linear array-based queue QUEUE[0..3] (size N = 4) is initially empty, with FRONT = REAR = -1. The dequeue algorithm reports UNDERFLOW whenever FRONT = -1 OR FRONT > REAR; after removing the last element it resets FRONT = REAR = -1. Trace the following operations: dequeue(), enqueue(5), enqueue(15), dequeue(), dequeue(), dequeue().

Solution:

dequeue(): FRONT = -1 → "QUEUE UNDERFLOW" reported; nothing to remove (the queue was never used).

enqueue(5): FRONT = 0, REAR = 0. Queue = [5]

enqueue(15): REAR = 1. Queue = [5, 15]

dequeue(): removes 5. FRONT = 1. Queue (logical) = [15]

dequeue(): removes 15. FRONT becomes 2, which is now > REAR (1), so the algorithm resets FRONT = REAR = -1 (queue empty). Queue = [ ]

dequeue(): FRONT = -1 → "QUEUE UNDERFLOW" reported again; the queue has been drained and is empty.

Answer: Underflow is reported twice: once at the very first operation (queue never used) and once at the final operation (queue drained empty after two dequeues). Between these, 5 and 15 were successfully enqueued and later dequeued in that same order.

Problem 2. Explain why the dequeue algorithm's underflow check must test "FRONT = -1 OR FRONT > REAR" rather than just "FRONT = -1". Use Problem 1 to illustrate a case where checking only FRONT = -1 would fail to detect an empty queue.

Solution:

FRONT = -1 alone only detects the case where the queue has never had anything enqueued into it.

A queue can also become empty later, after being used and fully drained by dequeue operations — in that situation FRONT keeps advancing but is not automatically -1 unless the algorithm explicitly resets it.

In Problem 1, after the second dequeue() removes the last element (15), FRONT would become 2 while REAR remains 1 — that is, FRONT > REAR — signalling the queue is empty even though FRONT is not -1. If the reset step were omitted and the underflow check only tested "FRONT = -1", this drained-empty state would go undetected, and the next dequeue() would incorrectly try to read QUEUE[2], an invalid/stale position.

Poly Notes Hub

Answer: The condition "FRONT = -1 OR FRONT > REAR" is needed because a queue can be empty either from having never been used (FRONT = -1) or from being fully drained after use (FRONT overtakes REAR) — both cases must be caught to avoid reading invalid data.

Section B: Circular Queue — Trace-Based Problems

Problem 3. A circular queue of size N = 5 uses a COUNT variable (FRONT = 0, REAR = -1, COUNT = 0 initially; REAR = (REAR+1) MOD N on enqueue, FRONT = (FRONT+1) MOD N on dequeue). Trace: enqueue(A), enqueue(B), enqueue(C), dequeue(), dequeue(), enqueue(D), enqueue(E), enqueue(F), enqueue(G), dequeue(), enqueue(H). State how many times REAR wraps around from index N-1 back to index 0 during this sequence.

Solution:

enqueue(A): REAR = 0. Q[0]=A. COUNT=1

enqueue(B): REAR = 1. Q[1]=B. COUNT=2

enqueue(C): REAR = 2. Q[2]=C. COUNT=3

dequeue(): removes Q[0]=A. FRONT=1. COUNT=2

dequeue(): removes Q[1]=B. FRONT=2. COUNT=1 (only C remains, at index 2)

enqueue(D): REAR = 3. Q[3]=D. COUNT=2

enqueue(E): REAR = 4. Q[4]=E. COUNT=3

enqueue(F): REAR = (4+1) MOD 5 = 0 — REAR wraps from 4 back to 0. Q[0]=F (overwrites the stale A). COUNT=4

enqueue(G): REAR = 1. Q[1]=G (overwrites stale B). COUNT=5 = N → queue FULL.

dequeue(): removes Q[FRONT=2]=C. FRONT=3. COUNT=4

enqueue(H): REAR = (1+1) MOD 5 = 2. Q[2]=H (overwrites old C). COUNT=5 → FULL again.

Answer: Final array = [F, G, H, D, E] (index 0 to 4); FRONT = 3, REAR = 2, COUNT = 5. Logical queue (front to rear) = D, E, F, G, H. REAR wrapped around from index 4 to index 0 exactly once, during the enqueue of F.

Problem 4. In a circular queue of size N = 10 that uses the "REAR points to the next free slot" convention (as in the enqueue algorithm of Problem 3 in this set, but with the sacrificed-slot method for detecting full instead of a COUNT variable), FRONT = 6 and REAR = 2 are observed. Using the formula: elements present = (REAR − FRONT + N) MOD N, find how many elements are currently stored, and how many more can be enqueued before the queue reports full (maximum usable capacity = N − 1).

Solution:

Elements present = (REAR − FRONT + N) MOD N = (2 − 6 + 10) MOD 10 = 6 MOD 10 = 6.

Maximum usable capacity under the sacrificed-slot method = N − 1 = 10 − 1 = 9.

Free slots remaining before the queue reports full = 9 − 6 = 3.

Answer: 6 elements are currently stored, and 3 more can be enqueued before the queue becomes full.

Section C: Priority Queue Problems

Problem 5. Four processes arrive together with the following (priority, burst time) pairs, where a LOWER priority number means higher priority and ties are broken by the order listed: P1 (2, 4), P2 (1, 3), P3 (3, 2), P4 (1, 5). Using non-preemptive priority scheduling, find the execution order and the waiting time of each process, and compute the average waiting time.

Poly Notes Hub

Solution:

Group by priority: priority 1 → P2, P4 (P2 listed first); priority 2 → P1; priority 3 → P3.

Execution order (highest priority, i.e. lowest number, first): P2, P4, P1, P3.

Waiting time = sum of burst times of all processes executed before it. P2 starts immediately: waiting time = 0.

P4 waits for P2's burst (3): waiting time = 3.

P1 waits for P2 + P4 (3 + 5 = 8): waiting time = 8.

P3 waits for P2 + P4 + P1 (3 + 5 + 4 = 12): waiting time = 12.

Average waiting time = (0 + 3 + 8 + 12) / 4 = 23 / 4 = 5.75.

Answer: Execution order: P2, P4, P1, P3. Waiting times: P2=0, P4=3, P1=8, P3=12. Average waiting time = 5.75 time units.

Problem 6. A min-heap (array representation, 1-indexed) currently holds H = [2, 4, 3, 8, 9, 7, 10]. Perform one EXTRACT-MIN operation: remove the minimum, move the last element to the root, and heapify-down (sift-down) to restore the heap property. Show the array after each step.

Solution:

Step 1 — remove the minimum: the minimum is H[1] = 2 (this is the value returned by extract-min).

Step 2 — move the last element to the root and shrink the heap size to 6: H[1] = H[7] = 10, giving H = [10, 4, 3, 8, 9, 7].

Step 3 — heapify-down from index 1: children of index 1 are H[2]=4 and H[3]=3; the smaller child is H[3]=3. Since H[1]=10 > 3, swap → H = [3, 4, 10, 8, 9, 7].

Continue from index 3: its only child (in a 6-element heap) is H[6]=7. Since H[3]=10 > 7, swap → H = [3, 4, 7, 8, 9, 10].

Continue from index 6: it is a leaf (no children) → heapify-down stops.

Answer: Extract-min returns 2. Final heap array after the operation = [3, 4, 7, 8, 9, 10], which is a valid min-heap (every parent ≤ its children).

Poly Notes Hub

Section D: Algorithm Design Problems

Problem 7. Write an algorithm to find the first non-repeating character seen so far in a stream of characters, using a queue and a frequency count, and trace it for the character stream: a, a, b, c, b, c, d.

Solution:

```
FIRST_NON_REPEATING(stream):
    Create empty queue Q; frequency map freq[] = 0
    For each character c in stream:
        freq[c] = freq[c] + 1
        ENQUEUE(Q, c)
        While Q is not empty AND freq[FRONT(Q)] > 1:
            DEQUEUE(Q)
        If Q is not empty:
            Print FRONT(Q)
        Else:
            Print "no non-repeating character"
```

'a': freq[a]=1; Q=[a]; front a has freq 1 → print 'a'

'a': freq[a]=2; Q=[a,a]; front a has freq 2>1 → dequeue, dequeue again → Q=[ ]; empty → print "none"

'b': freq[b]=1; Q=[b]; front b freq 1 → print 'b'

'c': freq[c]=1; Q=[b,c]; front b freq 1 → print 'b'

'b': freq[b]=2; Q=[b,c,b]; front b freq 2>1 → dequeue → Q=[c,b]; front c freq 1 → print 'c'

'c': freq[c]=2; Q=[c,b,c]; front c freq 2>1 → dequeue → Q=[b,c]; front b freq 2>1 → dequeue → Q=[c]; front c freq 2>1 → dequeue → Q=[ ]; empty → print "none"

'd': freq[d]=1; Q=[d]; front d freq 1 → print 'd'

Answer: Output sequence: a, none, b, b, c, none, d. The queue always holds only currently non-repeating characters in order of first appearance, so its front (if any) is the answer at each step.

Problem 8. Write an algorithm to reverse only the first k elements of a queue of size n (k ≤ n), keeping the remaining n − k elements in their original relative order, using one auxiliary stack. Trace it on Q = [1,2,3,4,5,6,7,8,9,10] (front to rear) with n = 10, k = 5.

Solution:

```
REVERSE_FIRST_K(Q, n, k):
Step 1: For i = 1 to k: PUSH(S, DEQUEUE(Q))
Step 2: While S is not empty: ENQUEUE(Q, POP(S))
Step 3: For i = 1 to n-k: ENQUEUE(Q, DEQUEUE(Q))
```

Step 1: dequeue 1,2,3,4,5 and push each onto S → S = [1,2,3,4,5] (top = 5). Q = [6,7,8,9,10]

Step 2: pop 5,4,3,2,1 and enqueue each → Q = [6,7,8,9,10,5,4,3,2,1]. S is now empty.

Step 3: rotate the remaining n−k = 5 original elements from front to rear (dequeue then immediately enqueue), 5 times: 6→ Q=[7,8,9,10,5,4,3,2,1,6]; 7→ Q=[8,9,10,5,4,3,2,1,6,7]; 8→ Q=[9,10,5,4,3,2,1,6,7,8]; 9→ Q=[10,5,4,3,2,1,6,7,8,9]; 10→ Q=[5,4,3,2,1,6,7,8,9,10]

Answer: Final queue = [5, 4, 3, 2, 1, 6, 7, 8, 9, 10]. The first 5 elements are reversed (5,4,3,2,1) while the last 5 (6,7,8,9,10) remain in their original order.

Problem 9. Prove that if the elements 1, 2, ..., n are enqueued into an empty queue in that order, the only possible dequeue sequence is 1, 2, ..., n — regardless of how the enqueue and dequeue operations are

Poly Notes Hub

interleaved. Contrast this with a stack, where interleaving enqueue/push and dequeue/pop operations can produce many different valid output orders.

Solution:

A queue's dequeue operation always removes whichever remaining element was enqueued earliest among those still present (FIFO discipline) — it can never skip ahead to remove a more recently enqueued element while an earlier one still waits.

Since 1, 2, ..., n are enqueued strictly in that order, at any moment the elements still inside the queue are always some contiguous suffix of positions already enqueued, and among them the earliest-enqueued (smallest-numbered) one is always at the front.

Therefore, no matter when dequeue operations are interleaved with the remaining enqueues, each dequeue call is forced to return the smallest-numbered element not yet removed — so the overall sequence of values returned must be exactly 1, 2, 3, ..., n, in that order.

Contrast with a stack: pop always removes the most recently pushed (LIFO) element, so pushing 1, 2, ..., n while interleaving pops can, depending on when the pops occur, yield many different valid output permutations (e.g. push 1, push 2, pop → 2, push 3, pop → 3, pop → 1, giving the order 2, 3, 1).

Answer: A queue can never reorder its elements — enqueuing 1..n in order forces the dequeue sequence to also be exactly 1..n, unlike a stack, which can produce many different output orders from the same push sequence because of its LIFO discipline.

Section E: Analytical / Complexity Problems

Problem 10. Distinguish between a Queue and a Deque (Double-Ended Queue) in terms of the operations each supports, and give one application where a Deque is required but a plain Queue is insufficient.

Solution:

A Queue permits insertion only at the REAR and deletion only at the FRONT — strict FIFO access at exactly one end each.

A Deque (double-ended queue) permits both insertion and deletion at BOTH ends (FRONT and REAR), making it a strict generalisation of both the queue and the stack.

Application requiring a Deque: the "sliding window maximum" algorithm, which scans an array with a window of fixed size and must, at each step, discard elements from the front of the structure that have fallen outside the window and also discard elements from the rear that are smaller than the newly arrived element (since they can never become the maximum while the new, larger element remains in the window). A plain queue cannot remove elements from its rear, so it cannot support this efficiently — a deque is required.

Answer: A queue allows access at one end each for insertion and deletion; a deque allows both operations at both ends. The sliding-window-maximum algorithm is a standard example that requires a deque because elements must be removed from both ends during the scan.

Problem 11. A priority queue is implemented with an unsorted array (insertion O(1), extract-min O(n) via a linear scan) versus a binary heap (insertion O(log n), extract-min O(log n)). For a workload that performs n insertions followed by exactly ONE extract-min at the end, compute the total time in Big-O for both implementations and state which is more efficient for this particular workload.

Solution:

Unsorted array: n insertions at O(1) each = O(n) total; then a single extract-min requires one O(n) linear scan to find the minimum = O(n). Combined total = O(n) + O(n) = O(n).

Binary heap: n insertions at O(log n) each = O(n log n) total; then a single extract-min = O(log n). Combined total = O(n log n) + O(log n) = O(n log n).

Since O(n) grows strictly slower than O(n log n) for large n, the unsorted-array implementation performs less total work for this specific workload.

Answer: The unsorted-array priority queue (total O(n)) is more efficient than the heap-based one (total O(n log n)) for this workload of many insertions followed by only a single extraction. A binary heap becomes worthwhile only when insertions and extractions are both frequent and interleaved, not for a single bulk extraction at the end.

Problem 12. A circular queue of size N is used as a bounded buffer between a fast producer and a slow consumer. Explain, in terms of the queue's full and empty conditions, what happens to the producer when the buffer becomes full, and to the consumer when the buffer becomes empty, and why a bounded (fixed-size) queue is essential for this producer–consumer scenario.

Solution:

When the buffer (circular queue) becomes full — i.e., the isFull() condition becomes true — any further enqueue attempt by the producer must be blocked or made to wait, since there is no free slot to place the new item without overwriting unread data.

When the buffer becomes empty — i.e., isEmpty() is true — any dequeue attempt by the consumer must be blocked or made to wait, since there is nothing available to remove.

A bounded queue is essential here because it caps the amount of memory used regardless of how much faster the producer is than the consumer; without a bound, an unbounded queue would grow indefinitely while the fast producer outpaces the slow consumer, eventually exhausting available memory.

Answer: The producer must pause (or the enqueue must fail/wait) when the buffer is full, and the consumer must pause (or the dequeue must fail/wait) when it is empty. A bounded circular queue is essential because it enforces a fixed memory ceiling and provides the well-defined full/empty signals needed to coordinate producer and consumer speeds safely.

Poly Notes Hub

Section F: Application-Based Problems

Problem 13. (Josephus Problem) Seven people numbered 1 to 7 stand in a circle. Starting the count at person 1, every 3rd person is eliminated, using a queue to simulate the process: repeatedly dequeue and immediately re-enqueue (k − 1) surviving people, then dequeue and eliminate the k-th person (do not re-enqueue them). Repeat until only one person remains. Find the order of elimination and the final survivor, for k = 3.

Solution:

Q = [1,2,3,4,5,6,7]. Pass 1,2 (requeue): dequeue 1→enqueue; dequeue 2→enqueue → Q=[3,4,5,6,7,1,2]. Eliminate: dequeue 3 → ELIMINATED #1. Q=[4,5,6,7,1,2]

Q=[4,5,6,7,1,2]. Requeue 4, 5 → Q=[6,7,1,2,4,5]. Eliminate: dequeue 6 → ELIMINATED #2. Q=[7,1,2,4,5]

Q=[7,1,2,4,5]. Requeue 7, 1 → Q=[2,4,5,7,1]. Eliminate: dequeue 2 → ELIMINATED #3. Q=[4,5,7,1]

Q=[4,5,7,1]. Requeue 4, 5 → Q=[7,1,4,5]. Eliminate: dequeue 7 → ELIMINATED #4. Q=[1,4,5]

Q=[1,4,5]. Requeue 1, 4 → Q=[5,1,4]. Eliminate: dequeue 5 → ELIMINATED #5. Q=[1,4]

Q=[1,4]. Requeue 1, 4 (k−1=2 passes wraps fully around this 2-person queue) → Q=[1,4]. Eliminate: dequeue 1 → ELIMINATED #6. Q=[4]

Answer: Elimination order: 3, 6, 2, 7, 5, 1. Final survivor: person 4. This demonstrates how a plain queue's rotate-and-remove pattern (dequeue-then-enqueue to "pass", plain dequeue to "eliminate") elegantly simulates circular counting-out problems.

Problem 14. A network router buffers incoming packets in a FIFO queue of capacity 4 before forwarding them one at a time, forwarding (removing the oldest buffered packet, if any) at every 2 ms tick starting at t = 0 (i.e., at t = 0, 2, 4, 6, ...). Seven packets P1–P7 arrive at times (ms) 0, 1, 2, 2, 3, 4, 5 respectively (P3 and P4 both arrive at t = 2, P3 first). If a packet arrives when the buffer already holds 4 packets and no forwarding tick occurs at that exact instant, the packet is dropped. Simulate the buffer and identify any dropped packet(s).

Solution:

t=0: forwarding tick (buffer empty, nothing to forward). P1 arrives → buffer=[P1] (1/4).

t=1: no tick. P2 arrives → buffer=[P1,P2] (2/4).

t=2: tick — forward P1 (oldest) → buffer=[P2] (1/4). Then P3 arrives → buffer=[P2,P3] (2/4); P4 arrives → buffer=[P2,P3,P4] (3/4).

t=3: no tick. P5 arrives → buffer=[P2,P3,P4,P5] (4/4, now full).

t=4: tick — forward P2 → buffer=[P3,P4,P5] (3/4). Then P6 arrives → buffer=[P3,P4,P5,P6] (4/4, full).

t=5: no tick, and the buffer is already full (4/4) → P7 arrives and finds no free slot → P7 is DROPPED.

t=6: tick — forward P3 → buffer=[P4,P5,P6] (3/4). No further arrivals remain.

t=8,10,12: ticks forward P4, then P5, then P6, emptying the buffer.

Answer: Packet P7 is dropped — it arrives at t=5 when the buffer is already at full capacity (4/4) and no forwarding tick occurs at that same instant to free a slot. All other packets (P1–P6) are successfully buffered and forwarded.

Poly Notes Hub