DATA STRUCTURE

Chapter 4: Queues

Exam-Oriented Problems & Solutions — Set 2

This is the second set of fourteen exam-oriented problems on the Queues chapter, with entirely new scenarios and numbers (no repeats from Set 1). It covers array/circular queue tracing with a different full-detection convention, priority queue and max-heap construction, three further algorithm-design problems (palindrome check, binary number generation, interleaving), deeper analytical/complexity questions, and a full waiting-line simulation. Each problem is followed by a complete, step-by-step solution.

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

Problem 1. A linear array-based queue QUEUE[0..5] (size N = 6) is initially empty, with FRONT = REAR = -1. The following operations are performed in order: enqueue(A), enqueue(B), enqueue(C), enqueue(D), dequeue(), dequeue(), enqueue(E), enqueue(F), enqueue(G). Trace the queue after every operation and state what happens on the last operation.

Solution:

enqueue(A): FRONT = 0 (first insertion), REAR = 0. Queue = [A]

enqueue(B): REAR = 1. Queue = [A, B]

enqueue(C): REAR = 2. Queue = [A, B, C]

enqueue(D): REAR = 3. Queue = [A, B, C, D]

dequeue(): removes A, FRONT = 1. Queue (logical) = [B, C, D]

dequeue(): removes B, FRONT = 2. Queue (logical) = [C, D]

enqueue(E): REAR = 4. Queue = [C, D, E]

enqueue(F): REAR = 5 = N - 1. Queue = [C, D, E, F]. The array is now physically full even though only 4 of the 6 slots are logically occupied.

enqueue(G): REAR = N - 1 already → "QUEUE OVERFLOW" reported; G cannot be inserted.

Answer: Overflow occurs at the 9th operation. Only 4 elements (C, D, E, F) are logically present out of capacity 6; slots 0 and 1 are wasted — the familiar false-overflow limitation of a simple linear array queue.

Problem 2. Prove that, for a simple (non-circular) linear array queue of capacity N, at most N enqueue operations can ever succeed over the entire lifetime of the array, no matter how the enqueue and dequeue operations are interleaved.

Solution:

In the standard linear-queue algorithm, REAR is only ever incremented by enqueue — it is never decremented by dequeue (only FRONT moves forward on dequeue).

REAR starts at -1 and can range only over the valid index values 0, 1, 2, ..., N-1 before the overflow condition (REAR = N - 1 on the next enqueue attempt) blocks all further insertion.

Since REAR increases by exactly 1 on every successful enqueue and can take at most N distinct values (0 through N-1), the number of successful enqueue operations over the array's lifetime cannot exceed N — regardless of how many dequeue operations occur in between, because dequeues only move FRONT, never REAR.

Answer: A simple linear array queue of capacity N can perform at most N successful enqueue operations in total, ever — this is precisely why the false-overflow problem occurs and why circular queues (Section B) are used in practice.

Section B: Circular Queue — Trace-Based Problems (Sacrificed-Slot Method)

Problem 3. A circular queue of size N = 4 uses the "sacrificed slot" convention: initially FRONT = REAR = 0 (this represents an empty queue). Before every enqueue, the algorithm checks if (REAR + 1) MOD N = FRONT (queue full); on enqueue it stores the item at QUEUE[REAR] and then sets REAR = (REAR + 1) MOD N. On dequeue, it checks FRONT = REAR (queue empty); otherwise it reads QUEUE[FRONT] and sets FRONT = (FRONT + 1) MOD N. Trace: enqueue(10), enqueue(20), enqueue(30), enqueue(40), dequeue(), enqueue(40).

Solution:

Initial: FRONT = 0, REAR = 0 (empty).

enqueue(10): full check (0+1) MOD 4 = 1 ≠ FRONT(0) → not full. QUEUE[0] = 10; REAR = 1.

enqueue(20): (1+1) MOD 4 = 2 ≠ FRONT(0) → not full. QUEUE[1] = 20; REAR = 2.

enqueue(30): (2+1) MOD 4 = 3 ≠ FRONT(0) → not full. QUEUE[2] = 30; REAR = 3.

enqueue(40): (3+1) MOD 4 = 0 = FRONT(0) → QUEUE FULL. 40 is rejected — only 3 of the 4 slots could ever be used at once, since index 3 (REAR) is always kept as the sacrificed empty slot.

dequeue(): FRONT(0) ≠ REAR(3) → not empty. Returns QUEUE[0] = 10; FRONT = 1.

enqueue(40): (3+1) MOD 4 = 0 ≠ FRONT(1) → not full now. QUEUE[3] = 40; REAR = (3+1) MOD 4 = 0.

Answer: 40 is rejected on the 4th operation (queue full with only 3 elements stored — the sacrificed-slot method has a maximum usable capacity of N − 1 = 3), but succeeds after the dequeue frees a slot. Final state: FRONT = 1, REAR = 0; logical queue (front to rear) = 20, 30, 40.

Problem 4. In a circular queue of size N = 7 using the sacrificed-slot convention of Problem 3, FRONT = 4 and REAR = 4 are observed. Is the queue empty or full? Justify your answer, and explain why this convention can never directly signal "full" through FRONT = REAR.

Solution:

Under the sacrificed-slot convention, the empty condition is defined as FRONT = REAR, and the full condition is defined separately as (REAR + 1) MOD N = FRONT.

Here FRONT = 4 and REAR = 4, so FRONT = REAR holds.

By definition, this always means the queue is EMPTY under this convention — the sacrificed-slot method is designed precisely so FRONT = REAR is reserved exclusively for "empty" and can never simultaneously mean "full", because one array slot is always kept unused, guaranteeing REAR can never catch up to equal FRONT while data is present.

Answer: The queue is EMPTY. Under the sacrificed-slot convention, FRONT = REAR always means empty, never full — that ambiguity is exactly what this convention (at the cost of one wasted slot) is designed to avoid.

Section C: Priority Queue Problems

Problem 5. A hospital emergency triage system uses a priority queue in which a LOWER number means a MORE urgent case (1 = critical, 5 = minor). Patients arrive in this order: P1 (priority 3), P2 (priority 1), P3 (priority 1), P4 (priority 4), P5 (priority 2). Ties are broken by arrival order (FIFO among equal priority). Determine the order in which patients are treated.

Solution:

Group patients by priority value: priority 1 → P2, P3 (in arrival order); priority 2 → P5; priority 3 → P1; priority 4 → P4.

Since a lower number is more urgent, treatment proceeds from priority 1 upward: priority 1 first (P2 then P3, since P2 arrived before P3), then priority 2 (P5), then priority 3 (P1), then priority 4 (P4).

Answer: Treatment order: P2, P3, P5, P1, P4.

Problem 6. Construct a binary max-heap (array representation, 1-indexed) by inserting the following elements one at a time in the given order: 4, 10, 3, 5, 1. Show the heap array after every insertion using the "insert at end, then bubble up" algorithm (swap with parent while parent < child).

Solution:

Insert 4: [4]

Insert 10: append → [4, 10]; parent(2)=1, heap[1]=4 < 10 → swap → [10, 4]

Insert 3: append → [10, 4, 3]; parent(3)=1, heap[1]=10 > 3 → no swap → [10, 4, 3]

Insert 5: append → [10, 4, 3, 5]; parent(4)=2, heap[2]=4 < 5 → swap → [10, 5, 3, 4]; now at index 2, parent(2)=1, heap[1]=10 > 5 → no swap → [10, 5, 3, 4]

Insert 1: append → [10, 5, 3, 4, 1]; parent(5)=2, heap[2]=5 > 1 → no swap → final: [10, 5, 3, 4, 1]

Answer: Final max-heap array = [10, 5, 3, 4, 1]. The root (index 1) = 10 is the maximum element; every parent is ≥ its children, confirming the max-heap property.

Section D: Algorithm Design Problems

Problem 7. Write an algorithm to check whether the elements of a queue Q of size n form a palindrome, without permanently altering the original queue. Trace it on Q = [1, 2, 3, 2, 1] (front to rear).

Solution:

```
IS_PALINDROME(Q, n):
    Create array A[0..n-1]
    For i = 0 to n-1:
        x = DEQUEUE(Q)
        A[i] = x
        ENQUEUE(Q, x)      // restore element to the rear
    i = 0, j = n-1
    While i < j:
        If A[i] != A[j], return FALSE
        i = i + 1; j = j - 1
    Return TRUE
```

Loop copies each element into array A while immediately re-enqueueing it, so after n full rotations Q returns to its original order: A = [1, 2, 3, 2, 1], and Q is unchanged.

Two-pointer comparison: i=0,j=4 → A[0]=1, A[4]=1, equal. i=1,j=3 → A[1]=2, A[3]=2, equal. i=2,j=2 → loop stops (i < j is false).

Answer: All compared pairs matched, so IS_PALINDROME returns TRUE — Q = [1, 2, 3, 2, 1] is a palindrome, and the original queue is left unchanged by the algorithm.

Problem 8. Write an algorithm to generate the binary representations of the first n natural numbers using a queue of strings, and trace it for n = 5.

Solution:

```
GENERATE_BINARY(n):
    Create empty queue Q
    ENQUEUE(Q, "1")
    For i = 1 to n:
        s = DEQUEUE(Q)
        Print s
        ENQUEUE(Q, s + "0")
        ENQUEUE(Q, s + "1")
```

Start: Q = ["1"]

i=1: dequeue "1", print "1"; enqueue "10", "11". Q = ["10", "11"]

i=2: dequeue "10", print "10"; enqueue "100", "101". Q = ["11", "100", "101"]

i=3: dequeue "11", print "11"; enqueue "110", "111". Q = ["100", "101", "110", "111"]

i=4: dequeue "100", print "100"; enqueue "1000", "1001".

i=5: dequeue "101", print "101".

Answer: Printed sequence: 1, 10, 11, 100, 101 — the binary representations of 1 through 5. The queue's FIFO order guarantees the numbers are generated in strictly increasing order without needing any arithmetic conversion.

Problem 9. Write an algorithm to interleave the first half of a queue with its second half using one auxiliary stack (for even n), and trace it on Q = [1, 2, 3, 4, 5, 6] (front to rear, n = 6).

Solution:

```
INTERLEAVE(Q, n):
Step 1: Push the first n/2 elements of Q onto stack S.
Step 2: Pop all of S and enqueue each back into Q (empties S).
Step 3: Dequeue the next n/2 elements of Q and enqueue them back
         immediately (rotates the old second half to the rear).
Step 4: Push the (new) first n/2 elements of Q onto stack S again.
Step 5: Repeat n/2 times: pop S and enqueue into Q, then dequeue
         Q's front and enqueue it back into Q.
```

Stage	Queue Q (front → rear)	Stack S (top → bottom)
Start	1, 2, 3, 4, 5, 6	(empty)
After Step 1	4, 5, 6	3, 2, 1
After Step 2	4, 5, 6, 3, 2, 1	(empty)
After Step 3	3, 2, 1, 4, 5, 6	(empty)
After Step 4	4, 5, 6	1, 2, 3
After Step 5	1, 4, 2, 5, 3, 6	(empty)

Answer: Final interleaved queue = 1, 4, 2, 5, 3, 6 — the first half (1,2,3) and second half (4,5,6) are alternated as required.

Section E: Analytical / Complexity Problems

Problem 10. Even though the array underlying an array-based queue supports direct indexed access (QUEUE[i]), the Queue ADT does not expose such an operation. Explain why, referring to the principle of abstraction.

Solution:

An Abstract Data Type is defined by the operations it exposes to the user, not by how it happens to be implemented internally.

The Queue ADT deliberately restricts access to only the FRONT and REAR ends (through enqueue, dequeue, and peek) in order to guarantee FIFO ordering.

If arbitrary indexed access (QUEUE[i]) were exposed, a program could read or modify an element in the middle of the queue out of turn, silently breaking the FIFO guarantee that the whole ADT exists to enforce.

Answer: Direct indexed access is intentionally excluded from the Queue ADT's interface because exposing it would let users bypass the FIFO ordering contract, even though the underlying array implementation happens to support it — this is exactly what data abstraction/encapsulation is meant to prevent.

Problem 11. Compare the space usage of an array-based circular queue and a singly linked-list queue, each storing n elements, in terms of space complexity and per-element overhead.

Solution:

Array-based circular queue: total space is fixed at allocation time as O(N), where N is the declared array capacity — this space is reserved whether or not it is fully used. There is no extra per-element overhead, since elements sit in contiguous memory with no pointers, but the queue cannot exceed N elements without resizing, and slots may sit unused if n < N.

Linked-list queue: total space is O(n), growing and shrinking dynamically to match exactly the number of elements currently stored, so there is no pre-allocation waste. However, each node carries an extra

"next" pointer field alongside its data field, adding a fixed per-element memory overhead that the array representation does not have.

Answer: The array-based circular queue trades flexibility for zero per-element overhead within a fixed, pre-allocated block of O(N) space; the linked-list queue trades a small per-element pointer overhead for exact, dynamically-sized O(n) space with no capacity limit.

Problem 12. In a linked-list implementation of a queue, explain why a separate REAR pointer must be maintained for O(1) enqueue, and state the time complexity of enqueue if only a FRONT pointer were kept.

Solution:

Enqueue always inserts a new node after the current last node of the list. If the last node's location is already known (via REAR), the new node can be linked in and REAR updated to point to it directly, in a fixed number of steps regardless of list length.

If only FRONT were maintained, the last node's location would be unknown, so the algorithm would have to traverse the entire list starting from FRONT, node by node, until it reached the last node, before it could attach the new node.

Answer: Maintaining REAR keeps enqueue at O(1). Without it, enqueue would degrade to O(n), where n is the current number of elements in the queue, because of the full traversal needed to locate the last node on every insertion.

Section F: Application-Based Problems

Problem 13. A single-lane toll booth serves one car at a time in arrival order (a FIFO queue), and each car takes exactly 2 minutes to process. Cars arrive at times (minutes) 0, 1, 1, 3, 6, labelled Car1 to Car5 respectively in arrival order. Simulate the system to find each car's departure time and the maximum number of cars ever waiting in the queue at once (not counting the car currently being served).

Solution:

t=0: Car1 arrives, server idle → Car1 enters service immediately, serviced during [0, 2). Waiting queue = [ ].

t=1: Car2 arrives; server busy (Car1 until t=2) → Car2 joins the waiting queue = [Car2].

t=1: Car3 arrives; server still busy → joins queue = [Car2, Car3] (waiting count = 2).

t=2: Car1 departs. Server free → dequeue Car2, service during [2, 4). Waiting queue = [Car3].

t=3: Car4 arrives; server busy (Car2 until t=4) → joins queue = [Car3, Car4] (waiting count = 2).

t=4: Car2 departs. Server free → dequeue Car3, service during [4, 6). Waiting queue = [Car4].

t=6: Car3 departs, and Car5 arrives at the same instant. Server free → dequeue Car4 first (already waiting), service during [6, 8). Waiting queue = [ ]. Car5 then joins the (now empty) queue = [Car5].

t=8: Car4 departs. Server free → dequeue Car5, service during [8, 10). Waiting queue = [ ].

t=10: Car5 departs.

Answer: Departure times: Car1 = 2, Car2 = 4, Car3 = 6, Car4 = 8, Car5 = 10. Maximum waiting-queue length observed = 2 (occurred twice: at t=1 and at t=3).

Problem 14. Using the toll-booth scenario of Problem 13, explain why service order always matches arrival order under a FIFO queue, and describe one real scenario where replacing the FIFO queue with a priority queue at such a server would be more appropriate — and what would change.

Solution:

A FIFO queue's dequeue operation always removes the element that has been waiting the longest (the earliest arrival), so as long as jobs join only at the rear and leave only from the front, service order is guaranteed to exactly match arrival order — no other attribute of a job can affect who is served next.

If the FIFO queue were replaced with a priority queue, the server would instead always serve whichever waiting job currently has the highest priority, regardless of how long other jobs have been waiting; a job that arrives later but carries higher priority could overtake one that arrived earlier.

Answer: A real example is a hospital emergency room (or an operating system's interrupt handler): a critical patient (or a hardware interrupt) that arrives later must still be treated before a less urgent case that has been waiting longer — here a priority queue is more appropriate than a pure FIFO queue, because urgency, not arrival time, should determine service order.