DATA STRUCTURE

Chapter 4: Queues

Exam-Oriented Problems & Solutions

This booklet contains fourteen exam-oriented problems on the Queues chapter, covering array/linear queue tracing, circular queue tracing, priority queue ordering, heap-based priority queue construction, algorithm design, complexity analysis, and real-world applications (CPU scheduling and BFS). 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..4] (size N = 5) is initially empty, with FRONT = REAR = -1. The following operations are performed in order: enqueue(10), enqueue(20), enqueue(30), dequeue(), enqueue(40), dequeue(), enqueue(50), enqueue(60). Trace the queue after every operation and state what happens on the last operation.

Solution:

Initial state: FRONT = -1, REAR = -1, Queue = [ ]

1) enqueue(10): FRONT becomes 0 (first insertion), REAR = 0. Queue = [10]

2) enqueue(20): REAR = 1. Queue = [10, 20]

3) enqueue(30): REAR = 2. Queue = [10, 20, 30]

4) dequeue(): removes QUEUE[FRONT] = 10, FRONT = 1. Queue (logical) = [20, 30]

5) enqueue(40): REAR = 3. Queue = [20, 30, 40]

6) dequeue(): removes 20, FRONT = 2. Queue (logical) = [30, 40]

7) enqueue(50): REAR = 4. Queue = [30, 40, 50]. Note REAR has now reached N - 1 = 4, so the array is physically full even though only 3 of the 5 slots are logically occupied.

8) enqueue(60): Check REAR = N - 1 → condition is true, so "QUEUE OVERFLOW" is reported and 60 cannot be inserted.

Answer: Overflow occurs at the 8th operation, even though only 3 elements (30, 40, 50) are logically present out of a capacity of 5. This is the classic "false overflow" problem of a simple linear array queue — slots 0 and 1 are free but unusable because REAR never moves backward.

Problem 2. For the same operations as Problem 1, what are the final values of FRONT and REAR just before the overflow is detected, and how many usable free slots remain unused?

Solution:

From the trace in Problem 1, just before the failed 8th operation: FRONT = 2, REAR = 4.

Logical elements present: QUEUE[2] = 30, QUEUE[3] = 40, QUEUE[4] = 50 → 3 elements.

Physically free slots: QUEUE[0] and QUEUE[1] (2 slots), left vacant after the two dequeue operations.

Answer: FRONT = 2, REAR = 4; 2 slots (index 0 and 1) remain unused and unreachable, confirming the wasted space caused by the linear queue design.

Section B: Circular Queue — Trace-Based Problems

Problem 3. A circular queue of size N = 5 (indices 0 to 4) uses a COUNT variable to track the number of elements (FRONT = 0, REAR = -1, COUNT = 0 initially). REAR is updated as REAR = (REAR + 1) MOD N on insertion, and FRONT as FRONT = (FRONT + 1) MOD N on deletion. Perform: enqueue(1), enqueue(2), enqueue(3), enqueue(4), enqueue(5), dequeue(), dequeue(), enqueue(6), enqueue(7). Trace the array, FRONT, REAR and COUNT after every operation.

Solution:

enqueue(1): REAR = (−1+1) MOD 5 = 0. QUEUE[0] = 1. FRONT = 0, REAR = 0, COUNT = 1.

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

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

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

enqueue(5): REAR = 4. QUEUE[4] = 5. COUNT = 5 = N → queue is now FULL. Array = [1, 2, 3, 4, 5].

dequeue(): removes QUEUE[FRONT] = QUEUE[0] = 1. FRONT = (0+1) MOD 5 = 1. COUNT = 4. Returned value: 1.

dequeue(): removes QUEUE[1] = 2. FRONT = 2. COUNT = 3. Returned value: 2. (Logical queue is now 3, 4, 5 at indices 2, 3, 4; indices 0 and 1 hold stale values but are free for reuse.)

enqueue(6): REAR = (4+1) MOD 5 = 0. QUEUE[0] = 6 (overwrites the stale 1). COUNT = 4.

enqueue(7): REAR = (0+1) MOD 5 = 1. QUEUE[1] = 7 (overwrites the stale 2). COUNT = 5 = N → FULL again.

Answer: Final array = [6, 7, 3, 4, 5] (index 0 to 4); FRONT = 2, REAR = 1, COUNT = 5. Logical queue front-to-rear = 3, 4, 5, 6, 7. Unlike Problem 1, all 5 slots were successfully reused — the circular queue completed 7 successful insertions in total using only 5 storage locations, avoiding false overflow.

Problem 4. A circular queue of size N = 8 currently has FRONT = 5 and REAR = 2 (the queue has wrapped around). Using the formula COUNT = (REAR − FRONT + N) MOD N + 1, find the number of elements currently stored in the queue.

Solution:

COUNT = (REAR − FRONT + N) MOD N + 1

COUNT = (2 − 5 + 8) MOD 8 + 1

COUNT = (5) MOD 8 + 1 = 5 + 1 = 6

Answer: The queue currently holds 6 elements.

Section C: Priority Queue Problems

Problem 5. Five tasks arrive with the following priority values (a higher number means higher priority): A = 3, B = 1, C = 4, D = 2, E = 5. State the order in which the tasks would be served by (a) a descending (max-first) priority queue and (b) an ascending (min-first) priority queue.

Solution:

(a) Descending priority queue serves the highest priority value first: E (5) → C (4) → A (3) → D (2) → B (1).

(b) Ascending priority queue serves the lowest priority value first: B (1) → D (2) → A (3) → C (4) → E (5).

Answer: (a) E, C, A, D, B (b) B, D, A, C, E

Problem 6. Construct a binary min-heap (array representation, 1-indexed) by inserting the following elements one at a time in the given order: 5, 3, 8, 1, 9, 2. Show the heap array after every insertion, applying the standard "insert at end, then bubble up" algorithm.

Solution:

Insert 5: [5]

Insert 3: append → [5, 3]; parent(2)=1, heap[1]=5 > 3 → swap → [3, 5]

Insert 8: append → [3, 5, 8]; parent(3)=1, heap[1]=3 < 8 → no swap → [3, 5, 8]

Insert 1: append → [3, 5, 8, 1]; parent(4)=2, heap[2]=5 > 1 → swap → [3, 1, 8, 5]; now at index 2, parent(2)=1, heap[1]=3 > 1 → swap → [1, 3, 8, 5]

Insert 9: append → [1, 3, 8, 5, 9]; parent(5)=2, heap[2]=3 < 9 → no swap → [1, 3, 8, 5, 9]

Insert 2: append → [1, 3, 8, 5, 9, 2]; parent(6)=3, heap[3]=8 > 2 → swap → [1, 3, 2, 5, 9, 8]; now at index 3, parent(3)=1, heap[1]=1 < 2 → no swap → final: [1, 3, 2, 5, 9, 8]

Answer: Final min-heap array = [1, 3, 2, 5, 9, 8]. The root (index 1) = 1 is the minimum element, confirming the min-heap property is satisfied at every node.

Section D: Algorithm Design Problems

Problem 7. Write an algorithm to implement a queue using two stacks, and trace it for the operations: enqueue(1), enqueue(2), enqueue(3), dequeue(), enqueue(4), dequeue(), dequeue().

Solution:

```
Let STACK1 = input stack, STACK2 = output stack.

ENQUEUE(x):
    PUSH(STACK1, x)

DEQUEUE():
    If STACK2 is empty, then
        While STACK1 is not empty:
            PUSH(STACK2, POP(STACK1))
    If STACK2 is still empty, report UNDERFLOW.
    Else return POP(STACK2)
```

enqueue(1): STACK1 = [1]

enqueue(2): STACK1 = [1, 2]

enqueue(3): STACK1 = [1, 2, 3]

dequeue(): STACK2 is empty, so move all of STACK1 to STACK2: pop 3→push, pop 2→push, pop 1→push, giving STACK2 = [3, 2, 1] (top = 1); STACK1 = [ ]. Pop STACK2 → returns 1. STACK2 = [3, 2]

enqueue(4): STACK1 = [4]

dequeue(): STACK2 is not empty → pop top → returns 2. STACK2 = [3]

dequeue(): STACK2 is not empty → pop top → returns 3. STACK2 = [ ]

Answer: Dequeue results in order: 1, 2, 3 — exactly the FIFO order in which they were enqueued, confirming correctness. Element 4 remains stored in STACK1, awaiting the next dequeue.

Problem 8. Write an algorithm to reverse the elements of a queue Q using only one auxiliary stack, and trace it on Q = [10, 20, 30, 40] (front to rear).

Solution:

```
REVERSE_QUEUE(Q):
Step 1: While Q is not empty:
            PUSH(S, DEQUEUE(Q))
Step 2: While S is not empty:
            ENQUEUE(Q, POP(S))
```

Step 1 — empty Q into stack S: dequeue 10 → push (S=[10]); dequeue 20 → push (S=[10,20]); dequeue 30 → push (S=[10,20,30]); dequeue 40 → push (S=[10,20,30,40]). Now Q is empty.

Step 2 — empty S back into Q: pop 40 → enqueue (Q=[40]); pop 30 → enqueue (Q=[40,30]); pop 20 → enqueue (Q=[40,30,20]); pop 10 → enqueue (Q=[40,30,20,10]).

Answer: Reversed queue (front to rear) = 40, 30, 20, 10. The LIFO nature of the stack reverses the FIFO order of the queue.

Problem 9. Write an algorithm that simulates a sequence of ENQUEUE/DEQUEUE operations on an array queue of capacity C and reports the position of the first operation that causes overflow or underflow (if any). Apply it to C = 3 and the operation sequence: E, E, D, E, E, E, D (E = enqueue, D = dequeue).

Solution:

```
VALIDATE(ops[1..k], C):
    count = 0
    For i = 1 to k:
        If ops[i] = ENQUEUE, then
            If count = C, then
                Report "Overflow at operation i" and stop.
            Else count = count + 1
        If ops[i] = DEQUEUE, then
            If count = 0, then
                Report "Underflow at operation i" and stop.
            Else count = count - 1
    Report "All operations valid"
```

i=1 (E): count 0 < 3 → count = 1

i=2 (E): count 1 < 3 → count = 2

i=3 (D): count 2 > 0 → count = 1

i=4 (E): count 1 < 3 → count = 2

i=5 (E): count 2 < 3 → count = 3

i=6 (E): count = 3 = C → OVERFLOW detected

Answer: Overflow occurs at operation 6 (the third enqueue after the dequeue), since the queue's capacity of 3 is already full at that point.

Section E: Analytical / Complexity Problems

Problem 10. Tabulate the time complexity of enqueue, dequeue, peek and search operations for: (a) a simple array queue, (b) a circular array queue, (c) a linked-list queue, and (d) a priority queue implemented with a binary heap.

Solution:

Implementation	Enqueue	Dequeue	Peek	Search
Simple array queue	O(1)*	O(1)	O(1)	O(n)
Circular array queue	O(1)	O(1)	O(1)	O(n)
Linked list queue	O(1)	O(1)	O(1)	O(n)
Priority queue (binary heap)	O(log n)	O(log n)	O(1)	O(n)

* For a simple (non-circular) array queue, enqueue is O(1) per operation, but the structure suffers from false overflow (Problems 1–2), so its effective usable capacity is lower than its actual size.

Problem 11. Using the results of Problems 1 and 3 (both using arrays of size N = 5), compare the effective slot utilisation of a linear queue versus a circular queue, and justify why the circular queue is preferred in practice.

Solution:

Linear queue (Problem 1): capacity N = 5, but overflow was reported after only 3 net elements remained logically stored (30, 40, 50), with REAR permanently stuck at index 4. Effective utilisation before failure = 3/5 = 60%, and 2 slots became permanently unusable for the remainder of the queue's life.

Circular queue (Problem 3): the same capacity N = 5 supported a total of 7 successful enqueue operations across the whole sequence (5 initial + 2 more after wrap-around), because freed slots (index 0 and 1) were reused via the MOD N arithmetic. Effective utilisation = 100% of the array at all times.

Answer: The circular queue is preferred because it reuses freed memory locations at the front of the array once elements are dequeued, avoiding the false overflow problem and achieving full (100%) utilisation of the allocated array, whereas a simple linear queue wastes space permanently as REAR advances.

Problem 12. A circular queue is to be built for N = 6. Compare the two standard techniques for distinguishing the full and empty conditions — (i) sacrificing one array slot, and (ii) maintaining a separate COUNT variable — in terms of maximum usable capacity and extra memory needed.

Solution:

(i) Sacrificed-slot method: the queue is considered full when (REAR + 1) MOD N = FRONT, and empty when FRONT = REAR. No extra variable is required, but one array slot can never be used for data, so the maximum usable capacity is N − 1 = 5 elements out of the 6 allocated.

(ii) COUNT-variable method: an integer COUNT tracks the number of elements directly; the queue is full when COUNT = N and empty when COUNT = 0. All N = 6 slots can be used for data (maximum usable capacity = 6), but this requires one extra integer variable that must be updated on every enqueue and dequeue.

Answer: The sacrificed-slot method loses 1 out of 6 slots (about 16.7% wasted for N = 6) but needs no extra variable; the COUNT method uses 100% of the allocated array but needs O(1) extra space for the counter. For small N, the sacrificed slot is a proportionally larger loss, so the COUNT-variable method is generally preferred when memory is tight.

Section F: Application-Based Problems

Problem 13. Three processes P1 (burst time 5), P2 (burst time 3) and P3 (burst time 8) arrive at time 0 and are scheduled using Round Robin with time quantum = 3, using a FIFO ready queue. Trace the ready queue and draw the Gantt chart, then find the completion time of each process.

Solution:

Ready queue initially: [P1, P2, P3]

t = 0 to 3: run P1 for 3 units (quantum). P1 remaining = 5 − 3 = 2. P1 is re-enqueued. Queue: [P2, P3, P1]

t = 3 to 6: run P2 for 3 units. P2 remaining = 3 − 3 = 0 → P2 completes at t = 6. Queue: [P3, P1]

t = 6 to 9: run P3 for 3 units. P3 remaining = 8 − 3 = 5. P3 is re-enqueued. Queue: [P1, P3]

t = 9 to 11: run P1 for its remaining 2 units (less than the quantum) → P1 completes at t = 11. Queue: [P3]

t = 11 to 14: run P3 for 3 units. P3 remaining = 5 − 3 = 2. P3 is re-enqueued. Queue: [P3]

t = 14 to 16: run P3 for its remaining 2 units → P3 completes at t = 16.

```
Gantt Chart:
| P1 | P2 | P3 | P1 | P3 | P3 |
0    3    6    9    11   14   16
```

Answer: Completion times: P2 = 6, P1 = 11, P3 = 16. This illustrates how the operating system's ready queue (a pure FIFO queue) underlies the Round Robin CPU scheduling algorithm.

Problem 14. An undirected graph has the adjacency lists: A → {B, C}, B → {A, D}, C → {A, D}, D → {B, C, E}, E → {D}. Perform a Breadth-First Search (BFS) starting from vertex A, showing the queue content at each step, and state the final visiting order.

Solution:

Initialise: visited = {A}, queue = [A]

Dequeue A; visit unvisited neighbours B, C (in list order) and enqueue them. visited = {A,B,C}, queue = [B, C]

Dequeue B; neighbours A (visited), D (unvisited) → enqueue D. visited = {A,B,C,D}, queue = [C, D]

Dequeue C; neighbours A (visited), D (visited) → nothing new enqueued. queue = [D]

Dequeue D; neighbours B (visited), C (visited), E (unvisited) → enqueue E. visited = {A,B,C,D,E}, queue = [E]

Dequeue E; neighbour D (visited) → nothing new. queue = [ ] → BFS terminates.

Answer: BFS visiting order: A, B, C, D, E. This demonstrates how a queue enforces level-by-level exploration: all neighbours of a node are enqueued before any of their own neighbours are processed.