

UNIT 3

Queue

(3 Hours) — Data Structures and Algorithms (CACS201)

3.1 Introduction

A **Queue** is a linear data structure in which elements are inserted from one end, called the **REAR** (or back), and removed from the other end, called the **FRONT**. This restriction makes a queue follow the **FIFO (First-In-First-Out)** principle — the element inserted first is the first one to be removed.

Everyday examples of queue behaviour include people standing in a line at a ticket counter, vehicles waiting at a toll booth, or print jobs waiting for a printer. In computing, queues are used to manage tasks waiting for a CPU, requests waiting at a server, and to implement breadth-first search in graphs.

Feature	Stack	Queue
Principle	LIFO (Last-In-First-Out)	FIFO (First-In-First-Out)
Insertion point	TOP	REAR
Deletion point	TOP	FRONT
Pointers needed	One (TOP)	Two (FRONT, REAR)

3.2 Queue as an Abstract Data Type

As an ADT, the Queue specifies a set of operations and the FIFO rule that governs them, without prescribing how the queue is implemented internally (array or linked list).

Operation	Description	Typical Return
enqueue(item)	Inserts an element at the REAR of the queue	None / success flag
dequeue()	Removes and returns the element at the FRONT	Removed element
peek() / front()	Returns the FRONT element without removing it	Front element
isEmpty()	Checks whether the queue has no elements	true / false
isFull()	Checks whether the queue has reached capacity	true / false (array only)
size()	Returns the number of elements in the queue	Integer count

3.3 Primitive Operations in Queue

Two pointers are maintained to manage a queue implemented with an array of size N :

- **FRONT** — index of the element to be removed next (initially -1, meaning empty).
- **REAR** — index of the last inserted element (initially -1, meaning empty).

All queue behaviour is built from four primitive operations, each of which must guard against invalid access:

Primitive Operation	Purpose	Failure Condition
Insertion (enqueue)	Adds a new element at REAR	Queue Overflow
Deletion (dequeue)	Removes the element at FRONT	Queue Underflow
Traversal	Visits each element from FRONT to REAR	—
Peek / Front access	Reads the FRONT element without removing it	Underflow if empty

Note: Unlike a Stack, a Queue needs two pointers (FRONT and REAR) instead of one, because insertion and deletion happen at opposite ends of the structure.

3.4 Linear Queue and Its Application

A **Linear Queue** is the simplest array-based implementation: REAR increases by 1 on every enqueue() and FRONT increases by 1 on every dequeue(). Elements are never shifted, so the array positions before FRONT are simply abandoned.

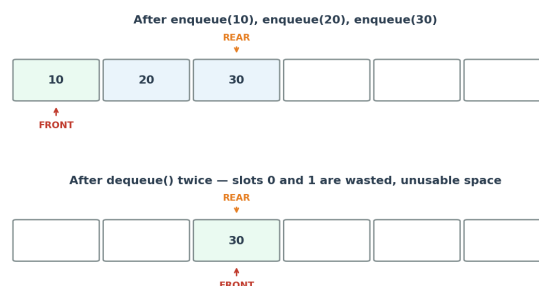


Figure 3.1 — Enqueue and dequeue on a linear queue of size 6

This design has a serious drawback: once REAR reaches the last index of the array ($N-1$), no further insertion is possible — **even if slots at the front are empty** — because REAR cannot move backward. This wasted space is known as **false overflow**.

- **Applications of a simple linear queue:** single-server waiting lines (e.g., a one-time batch print queue), simple task buffers where the queue is drained completely before being reused, and data buffering between two processes of different speeds.

Note: Because of false overflow, a linear queue is rarely used in practice for continuously running systems — a Circular Queue solves this limitation.

3.5 Circular Queue and Its Application

A **Circular Queue** treats the underlying array as circular — the last index is logically connected back to index 0. When REAR or FRONT reaches the end of the array, it wraps around to the beginning using the modulo operator, reusing the empty slots freed by earlier dequeue() calls.

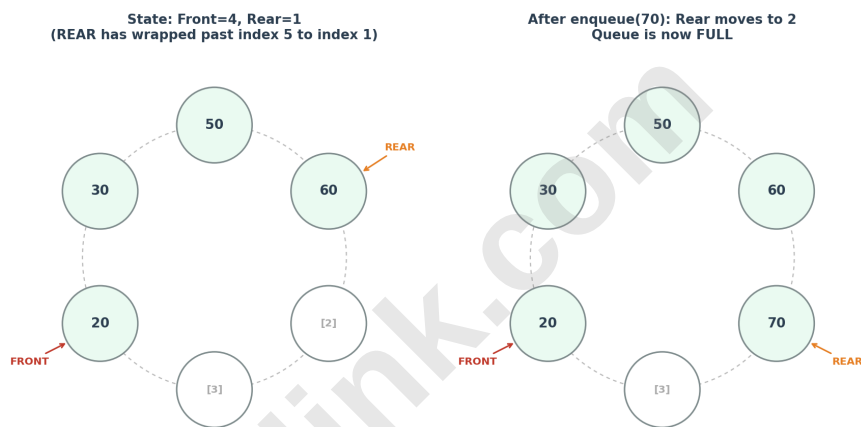


Figure 3.2 — REAR wrapping around from the end of the array back to index 0

In a circular queue of size N, the position for the next insertion or removal is computed as:

$$\text{REAR} = (\text{REAR} + 1) \bmod N \quad \text{FRONT} = (\text{FRONT} + 1) \bmod N$$

- **Applications of a circular queue:** CPU scheduling (round-robin), traffic light control systems that cycle through phases, buffering of streaming data (audio/video ring buffers), and keyboard/network input buffers where data arrives and is consumed continuously.

3.6 Enqueue and Dequeue Operations

3.6.1 Enqueue (Insertion) on a Circular Queue

Algorithm — enqueue(item)

1. If $(\text{REAR} + 1) \bmod N = \text{FRONT}$, print "Queue Overflow" and exit.
2. If $\text{FRONT} = -1$, set $\text{FRONT} = 0$. (first element being inserted)
3. Set $\text{REAR} = (\text{REAR} + 1) \bmod N$.
4. Set $\text{QUEUE}[\text{REAR}] = \text{item}$.

3.6.2 Dequeue (Deletion) on a Circular Queue

Algorithm — dequeue()

1. If $\text{FRONT} = -1$, print "Queue Underflow" and exit.
2. Set $\text{item} = \text{QUEUE}[\text{FRONT}]$.
3. If $\text{FRONT} = \text{REAR}$, set $\text{FRONT} = -1$ and $\text{REAR} = -1$. (queue becomes empty)
4. Else, set $\text{FRONT} = (\text{FRONT} + 1) \bmod N$.
5. Return item.

Worked Example — Circular Queue of size 5 (indices 0–4)

Start: $\text{FRONT} = -1$, $\text{REAR} = -1$ (empty)

enqueue(10) → $\text{FRONT} = 0$, $\text{REAR} = 0$, Queue = [10]

enqueue(20) → $\text{REAR} = 1$, Queue = [10, 20]

enqueue(30) → $\text{REAR} = 2$, Queue = [10, 20, 30]

dequeue() → returns 10, $\text{FRONT} = 1$, Queue = [20, 30]

enqueue(40) → $\text{REAR} = 3$, Queue = [20, 30, 40]

enqueue(50) → $\text{REAR} = 4$, Queue = [20, 30, 40, 50]

enqueue(60) → $(\text{REAR}+1) \bmod 5 = 0 \neq \text{FRONT}(1)$, so REAR wraps to 0, Queue = [60, 20, 30, 40, 50] (60 stored at index 0)

3.7 Priority Queue

A **Priority Queue** is a special type of queue in which every element carries a **priority**, and elements are removed based on priority rather than strictly on arrival order. The element with the highest priority is served first; among elements of equal priority, FIFO order is normally preserved.

Type	Removal Rule
Ascending Priority Queue	Element with the smallest priority value is removed first
Descending Priority Queue	Element with the largest priority value is removed first

Priority queues can be implemented using a sorted/unsorted array, a linked list, or — most efficiently — a **heap** data structure, which supports insertion and removal of the highest-priority element in $O(\log n)$ time.

Worked Example — Descending Priority Queue (job scheduler)

Insert Job A (priority 2), Job B (priority 5), Job C (priority 1), Job D (priority 4).

Queue (sorted by priority, highest first): B(5), D(4), A(2), C(1)

dequeue() → removes B (priority 5) first, regardless of insertion order.

dequeue() → removes D (priority 4) next.

Remaining queue: A(2), C(1) — C will be served last even though it was inserted third.

- **Applications of a priority queue:** CPU task scheduling by process priority, Dijkstra's shortest-path algorithm, Huffman encoding for data compression, hospital emergency-room patient triage, and event-driven simulations.

Key Takeaways

- ✓ A Queue is a linear structure that follows the FIFO principle; insertion happens at REAR, deletion happens at FRONT.
- ✓ As an ADT, a Queue exposes enqueue, dequeue, peek, isEmpty, and isFull without exposing its internal implementation.
- ✓ A Linear Queue suffers from false overflow once REAR reaches the last index, even if front slots are free.
- ✓ A Circular Queue solves this by wrapping REAR and FRONT using the modulo operator, reusing freed slots.
- ✓ A Priority Queue serves elements by priority rather than arrival order, and is typically implemented efficiently using a heap.

— End of Unit 3 —