Data Structures

Week 8 – The queue: FIFO principle, circular queue, double-ended queue, priority queue

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Stack:** LIFO – only the top is accessible
- Push, Pop, Top – all **O(1)**, with an array at the end of the array
- Applications: postfix evaluation, brackets, undo, call stack

Questions for today

- What is the difference between a stack and a queue?
- Why is the “naive” array queue bad, and how does the **circular** queue fix it?
- What are the double-ended queue and the priority queue?

The concept of a queue

A **queue** is a data structure into which we insert at the **rear** and from which we remove at the **front**.

FIFO – First In, First Out: what went in first comes out first.

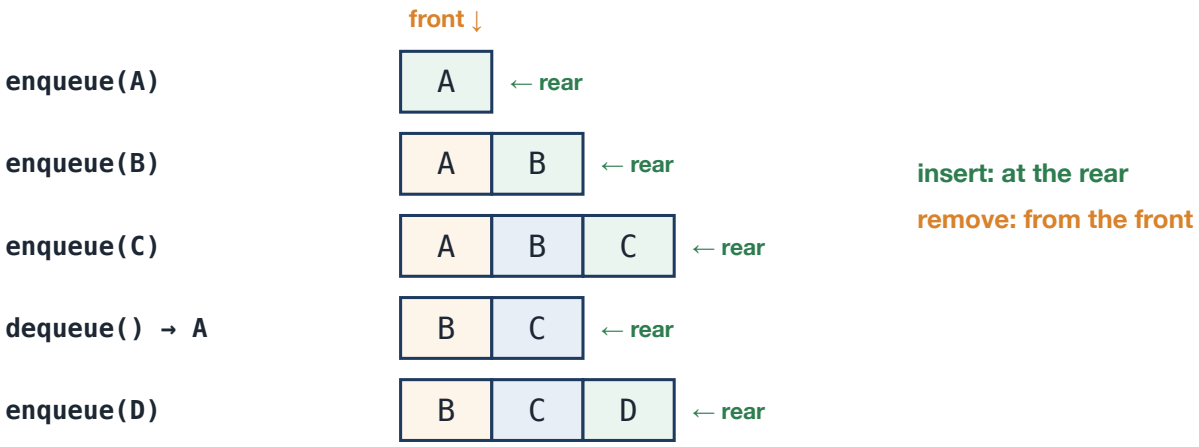
Everyday examples

- Queuing in the canteen, at the checkout
- Print queue
- Customer service phone line
- Keyboard buffer

Properties

Elements	homogeneous
Relationship	linear
Access	**restricted** – front and rear
Size	static or dynamic

Queue operations



insert: at the rear

remove: from the front

Operation	Meaning	Steps
Enqueue	element to the **rear** of the queue	O(1)
Dequeue	removing the **front** element	O(1)
Peek / Front	reading the front element	O(1)
IsEmpty? · IsFull?	state	O(1)

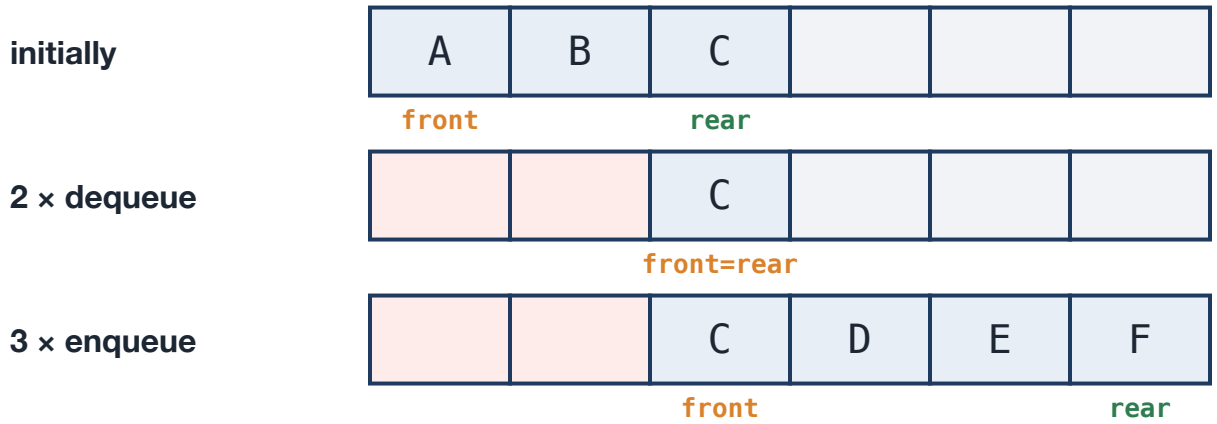
Stack and queue compared

	Stack	Queue
Principle	**LIFO**	**FIFO**
Insertion	onto the top (push)	at the rear (enqueue)
Removal	from the top (pop)	from the front (dequeue)
Access	one end	two ends
Pointers in the array implementation	`top`	`front`, `rear`
Typical application	undo, recursion, expressions	scheduling, buffering, breadth-first traversal

Queue with an array

First attempt: the naive array queue

`front` is the index of the first, `rear` of the last element; enqueue: `rear ← rear + 1`, dequeue: `front ← front + 1`



The queue 'drifts' to the right:

rear at the end → 'full',

although 2 cells are free in front!

Ways to fix the naive queue

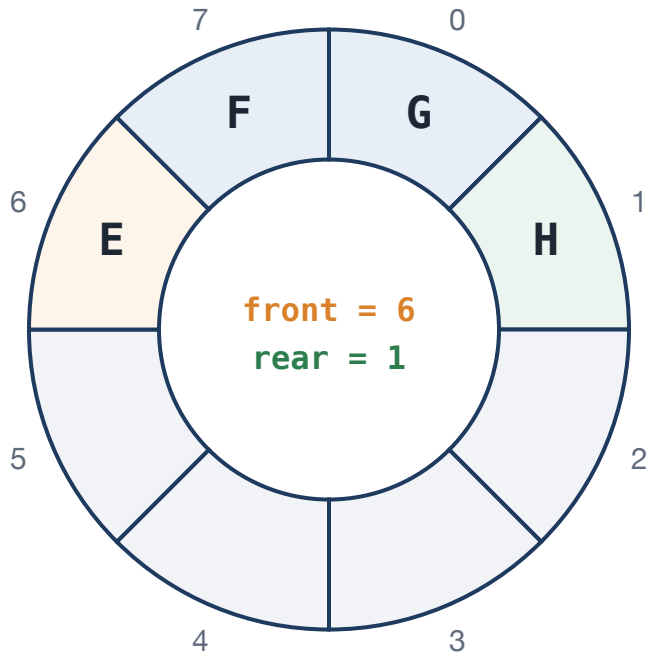
1. Shifting: after every dequeue the elements are shifted forward by one

- The front always stays 0, no space is lost
- But dequeue becomes **O(n) ✗**

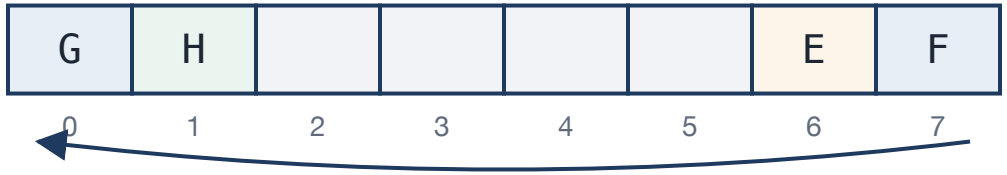
2. Circular queue: the array is treated as a **ring** – when `rear` reaches the end of the array, it continues at the **beginning**

- No element moves, every operation is **O(1) ✓**
- “Wrapping around” the index: **(i + 1) mod n**

Circular queue



The same as an array:




```
enqueue: rear ← (rear + 1) mod n
dequeue: front ← (front + 1) mod n
empty: count = 0
full: count = n
```

every operation O(1), no shifting

Empty or full? The pitfall of the circular queue

With only `front` and `rear`, an **empty** and a **full** queue look the same: `front = (rear + 1) mod n`!

Solutions:

1. Keep track of the **count** (`count`) – empty: `count = 0`, full: `count = n` ← **we use this**
2. **One slot always stays empty** – full: `(rear + 1) mod n = front`, but only **n** – 1 elements fit
3. A boolean flag (`empty`) set by the operations

> The counter-based solution is the simplest and the least error-prone.

Circular queue – pseudocode

```
// array T[0..n-1] (here exceptionally from 0,
// so that mod n is simple)

Procedure Create
  front ← 0 ; rear ← n − 1 ; count ← 0
End procedure

Function IsEmpty
  Return count = 0
End function

Function IsFull
  Return count = n
End function
```

```
Procedure Enqueue(x)
  If IsFull then ERROR("the queue is full")
  rear ← (rear + 1) mod n
  T[rear] ← x
  count ← count + 1
End procedure

Function Dequeue
  If IsEmpty then ERROR("the queue is empty")
  x ← T[front]
  front ← (front + 1) mod n
  count ← count − 1
  Return x
End function
```

Circular queue – Java and Go

Java

```java
class CircularQueue {
    private final int[] data;
    private int front = 0, rear = -1, count = 0;

    void enqueue(int x) {
        if (count == data.length)
            throw new IllegalStateException("full");
        rear = (rear + 1) % data.length;
        data[rear] = x;
        count++;
    }

    int dequeue() {
        if (count == 0)
            throw new IllegalStateException("empty");
        int x = data[front];
        front = (front + 1) % data.length;
        count--;
        return x;
    }
}
```

Go

```go
type CircularQueue struct {
    data               []int
    front, rear, count int
}

func (q *CircularQueue) Enqueue(x int) error {
    if q.count == len(q.data) {
        return errors.New("full")
    }
    q.rear = (q.rear + 1) % len(q.data)
    q.data[q.rear] = x
    q.count++
    return nil
}

func (q *CircularQueue) Dequeue() (int, error) {
    if q.count == 0 {
        return 0, errors.New("empty")
    }
    x := q.data[q.front]
    q.front = (q.front + 1) % len(q.data)
    q.count--
    return x, nil
}
```

Built-in queues

Java

```java
Queue<String> q = new ArrayDeque<>();
q.offer("Li");      // enqueue
q.offer("Wang");
q.peek();           // "Li"
q.poll();           // "Li" — dequeue
q.isEmpty();
```

Internally `ArrayDeque` is a **circular, dynamically growing** array.

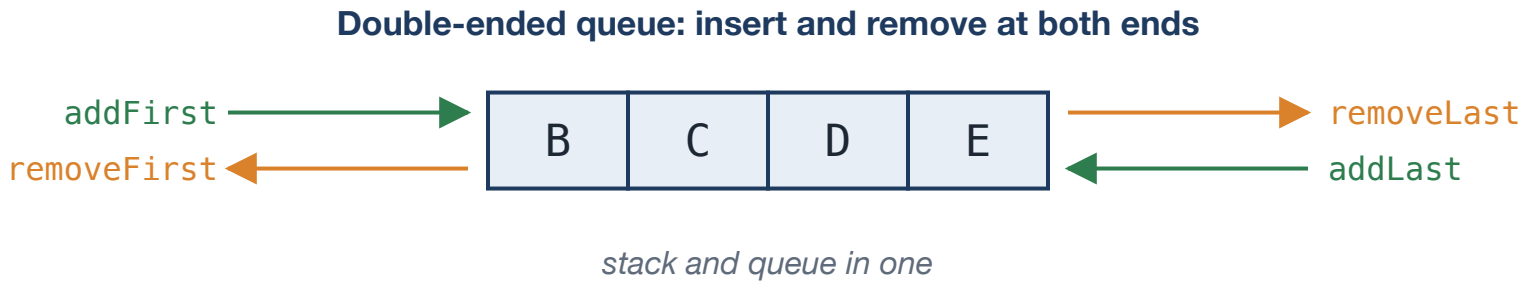
Go

```go
q := []string{}
q = append(q, "Li")   // enqueue
q = append(q, "Wang")
first := q[0]         // peek
q = q[1:]             // dequeue
```

Simple, but `q[1:]` does not free the beginning – for many operations an own circular queue or `container/list` is better.

Variants of the queue

Double-ended queue (deque)



stack and queue in one

- **Deque** (**double-ended queue**, pronounced “deck”): we can **insert** and **remove** at both ends
- Using only one end it is a **stack**, using opposite ends it is a **queue**
- With a circular array: for `addFirst`, `front ← (front − 1 + n) mod n`

> In Java and Go the value of `−1 % n` is **−1**, not **n** − 1! That is why `+ n` is needed.

Priority queue

Every element has a **priority**; we always remove the **most important** element (e.g. the one with the smallest priority number) – not the oldest one.

Examples: hospital emergency care, process scheduling in operating systems, route planning (Dijkstra)

Implementation	Insertion	Removing the min
unsorted array	**O(1)** – at the end	O(n) – minimum search
sorted array	O(n) – shifting into place	**O(1)** – from the end
heap (week 15)	**O(log n)**	**O(log n)**

Built-in: Java `PriorityQueue` · Go `container/heap`

Applications

- **Buffering:** keyboard, network packets, video streaming – balancing the different speeds of producer and consumer
- **Scheduling:** print queue, the **round-robin** scheduler of operating systems
- **Simulation:** checkouts, traffic, service systems
- **Breadth-first traversal** (BFS): level-by-level traversal of trees, shortest paths in graphs
- **Message queues** in distributed systems (e.g. processing the orders of an online shop)

> **Round-robin:** every process runs for at most one time slice and then goes to the rear of the queue – so everybody gets a turn.

Summary

- **Queue:** FIFO – insert at the rear, remove at the front
- Operations: **Enqueue, Dequeue, Peek, IsEmpty?, IsFull?** – all **O(1)**
- **Naive array queue:** the queue “drifts”, space is lost at the front
- **Circular queue:** `(i + 1) mod n` – wraps around, no shifting; empty/full: **count**
- **Deque:** insertion and removal at both ends
- **Priority queue:** the most important element comes out first – O(log n) with a heap

Next week: dynamic data structures – a model of dynamic memory management and the singly linked list.

Review questions

1. What does it print? `enqueue(1); enqueue(2); dequeue(); enqueue(3); enqueue(4); dequeue(); dequeue(); dequeue();`
2. Circular queue, n = 5, initially empty (front = 0, rear = 4). We perform: 4 × enqueue, 3 × dequeue, 3 × enqueue. What are `front`, `rear` and `count` now?
3. What does the following code print for n = 10? What does it do in general?
   `q.enqueue(0); q.enqueue(1); for (i = 0; i < n; i++) { a = q.dequeue(); print(a); b = q.peek(); q.enqueue(a + b); }`
4. True or false: in a circular queue dequeue is O(n), because the elements must be shifted forward.
5. Which data structure would you use for: a) the “Back” button of a browser, b) print jobs, c) hospital emergency admission?

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Naive queue	`NaiveQueue.java`	–	the problem of the “drifting” queue, fix by shifting
Circular queue	`CircularQueue.java`	`w08/circularqueue`	wrapping around, empty/full with a counter
Deque	`CircularDeque.java`	`w08/deque`	both ends, palindrome check
Priority queue	`PriorityQueueDemo.java`	`w08/priorityqueue`	with an unsorted array and with a heap

✎ Exercises

1. **Code tracing:** run the code of review question 3, and explain the output!
2. **Josephus problem:** **n** people stand in a circle, every **k**-th person is eliminated. Who remains last? (With a queue: send k – 1 people to the rear, the **k**-th is eliminated.)
3. **Round-robin:** simulate the scheduling! Processes: P1 = 5, P2 = 2, P3 = 4 time units, time slice = 2. When does each process finish?
4. **Reversing a queue** with the help of a stack.
5. **Code tracing:** letter = enqueue, `*` = dequeue and print: `D A T * A * S * T R * U C * T` (compare with the stack!)
6. **Moving average:** after every new data item, print the average of the last **k** items in O(1) steps!

Solutions: `Exercises08.java`, `w08/exercises`

✎ Homework

1. Implement the circular queue **without a counter**, keeping one slot empty! How many elements fit into an array of size **n**?
2. Extend the circular queue so that, when full, it copies the elements into an array **twice as large** (dynamic circular queue)! Pay attention to the order of the wrapped-around elements!
3. Implement a **queue with two stacks** (see the homework of week 7)! Compare it with the circular queue!
4. **Checkout simulation:** customers arrive randomly, a cashier serves one customer per minute. What is the average waiting time for different arrival rates?