Chapter 8: Queues and Deques

Data Structures

FIFO | enqueue / dequeue | Array-based queue | Circular queue | Linked queue | Deque | Sliding window | Round robin

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/data_structures/slides/Chapter_08_Queues_and_Deques.html

INTRODUCTION

Learning Outcomes

The FIFO model

Explain the first-in, first-out behavior of a queue.

Implementation

Implement a queue with an array and with a linked list.

Kinds of queues

Tell a circular queue from a double-ended queue (deque) and use each where it fits.

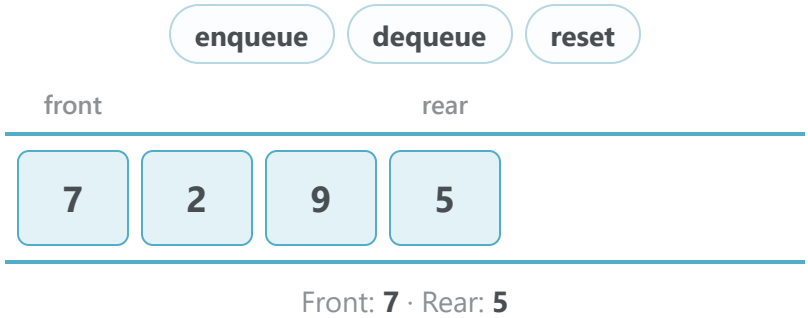
Applications

Choose the right kind of queue for real-world problems.

THE QUEUE ADT

The Queue

- A **linear** data structure that is open at both ends.
- Items leave in the order they arrived: **first in, first out** (FIFO).
- Items join at one end (the **rear**) and leave from the other (the **front**).
- Think of the line at a checkout or a ticket window: whoever arrives first is served first.



Front: **7** · Rear: **5**

Basic Queue Operations

Enqueue

Adds a new item at the rear of the queue.

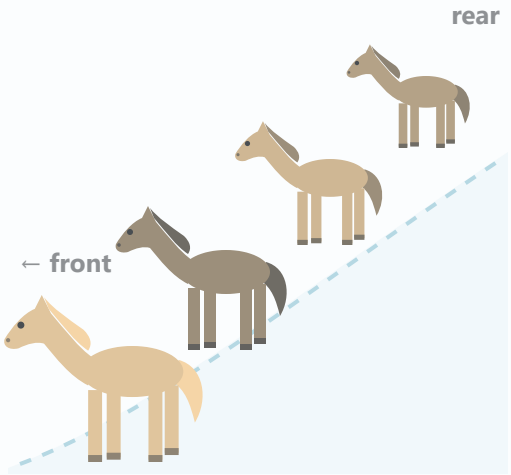
`Time: O(1)`

Dequeue

Removes and returns the item at the front. The first item in is the first out (FIFO).

`Time: O(1)`

Items in the middle are never accessed directly; only the **front** and **rear** are used.



Horses walking single file: the one in front passes first, and newcomers join at the back.

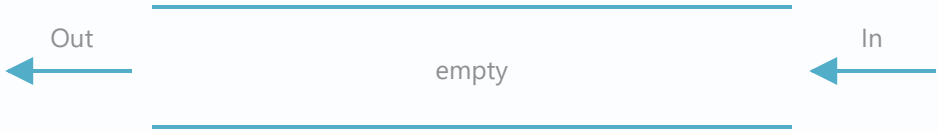
The Queue Abstract Data Type

Operation	Description
`enqueue(e)`	adds e at the rear
`dequeue()`	removes and returns the front item
`first()` / `peek()`	returns the front item without removing it
`rear()`	returns the rear item without removing it
`size()`	number of items
`isEmpty()`	is the queue empty?
`isFull()`	is it full? (fixed capacity only)

```
public interface Queue<E> {
    int size();
    boolean isEmpty();
    void enqueue(E e);
    E first();    // null if empty
    E dequeue();  // null if empty
}
```

Dequeuing from an empty queue is an **underflow**; enqueuing into a full one, an **overflow**. Goodrich's interface returns `null` when the queue is empty; some implementations throw an exception.

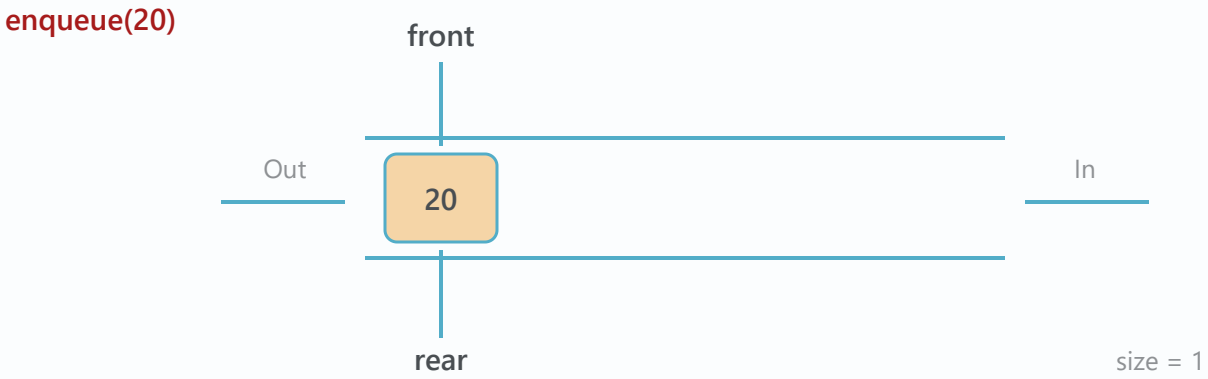
Queue Operations, Step by Step



front = rear = null

Initially the queue is empty: `front` and `rear` point to nothing (null).

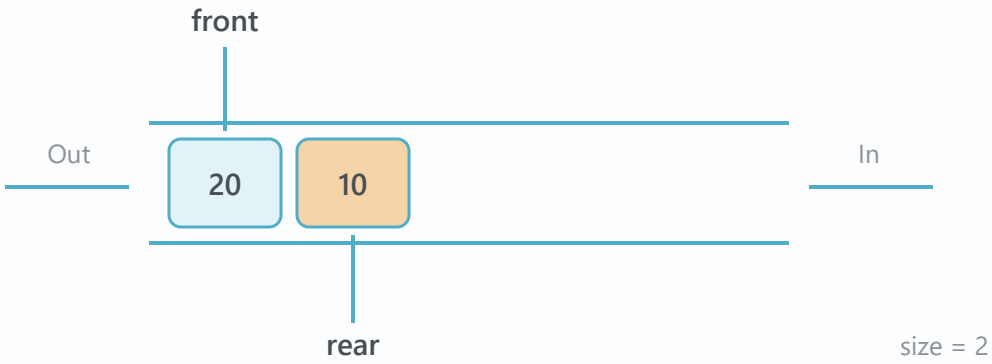
Queue Operations, Step by Step



The first item goes in: a lone item is both the **front** and the **rear**.

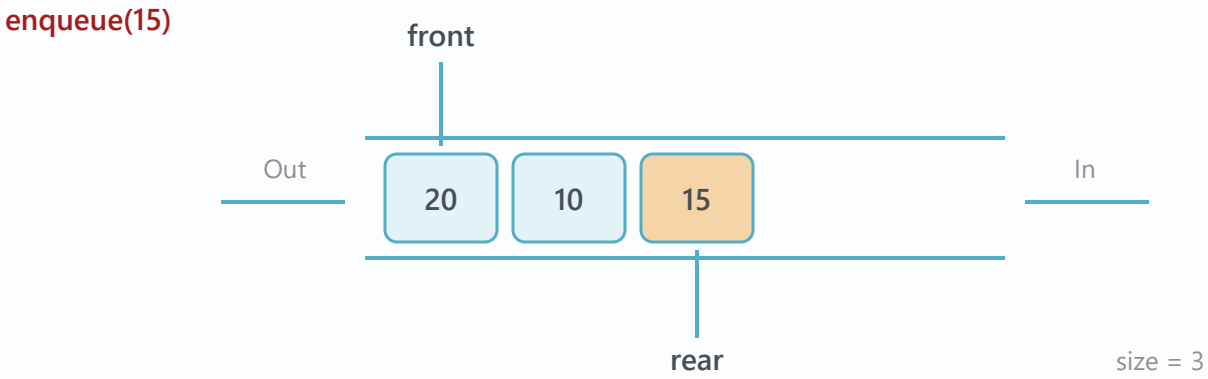
Queue Operations, Step by Step

enqueue(10)



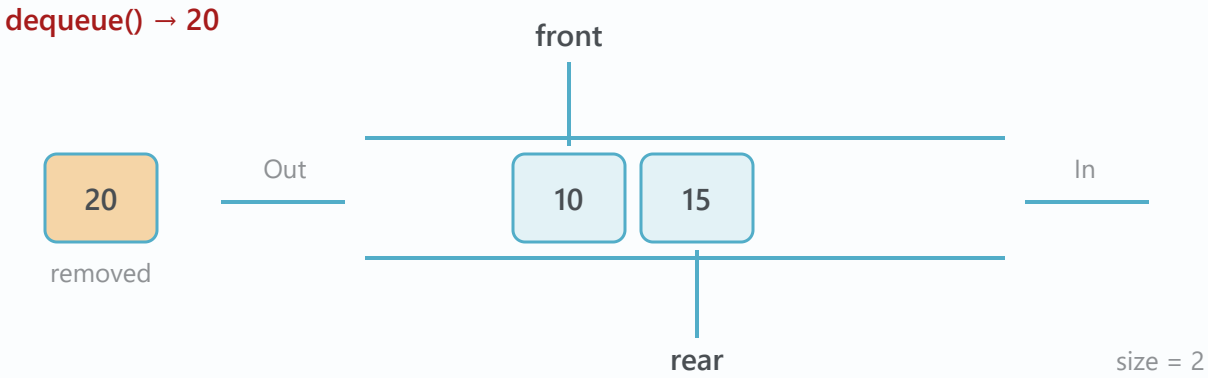
10 joins at the rear: `rear` advances, `front` stays put.

Queue Operations, Step by Step



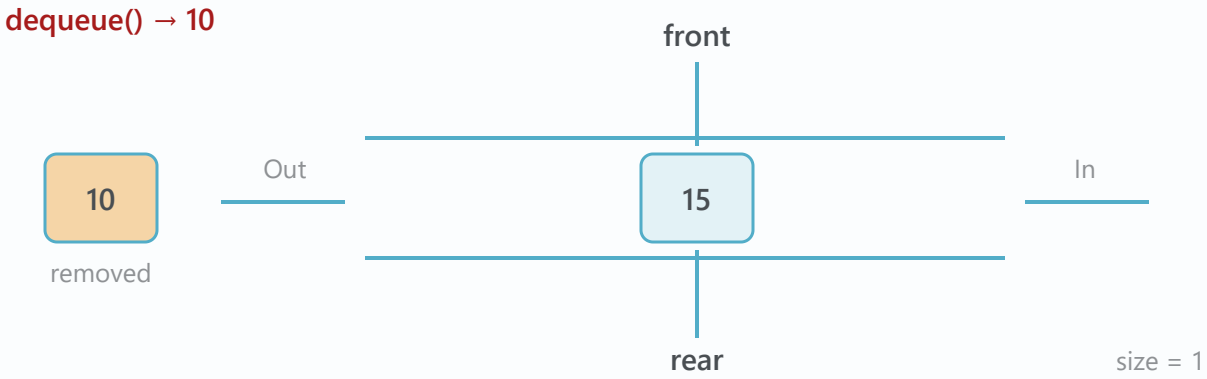
15 joins at the rear. Queue: 20, 10, 15 (front to rear).

Queue Operations, Step by Step



20 leaves from the front; `front` advances to the next item (10).

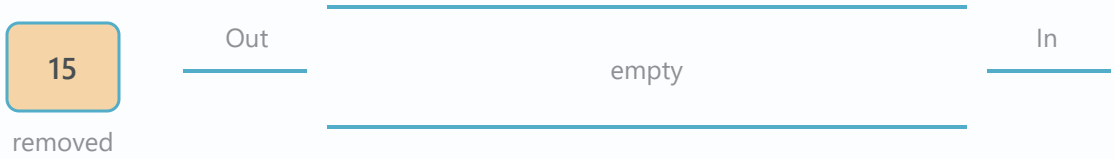
Queue Operations, Step by Step



10 leaves. Now `front` and `rear` point to the same item (15).

Queue Operations, Step by Step

dequeue() → 15



front = rear = null

When the last item leaves, both become null. Items left in the order 20, 10, 15, the order they arrived (FIFO).

Queues in Java: `java.util.Queue`

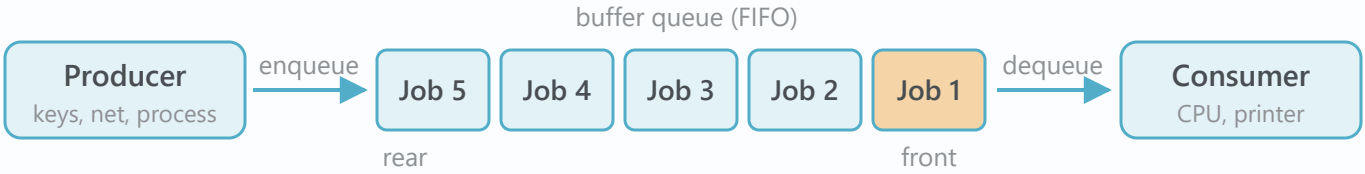
Operation	Throws an exception	Returns a special value
Insert	`add(e)`	`offer(e)` → false
Remove	`remove()`	`poll()` → null
Examine	`element()`	`peek()` → null

```
Queue<Integer> q = new ArrayDeque<>();
q.offer(20);
q.offer(10);
q.offer(15);        // [20, 10, 15]
q.peek();           // 20, not removed
q.poll();           // 20 → [10, 15]
q.poll();           // 10 → [15]
q.size();           // 1
q.poll();           // 15 → []
q.poll();           // null (empty)
```

- `ArrayDeque` : backed by a circular array; usually the fastest choice.
- `LinkedList` : backed by a linked list.

Our own classes use the names `enqueue` / `dequeue` ; the Java library uses `offer` / `poll` / `peek` .

Where Queues Are Used



Operating System Scheduler

Ready processes wait in the ready queue and get the CPU in turn.

Print Queue

Documents sent to a printer are printed in the order they arrive.

Buffers

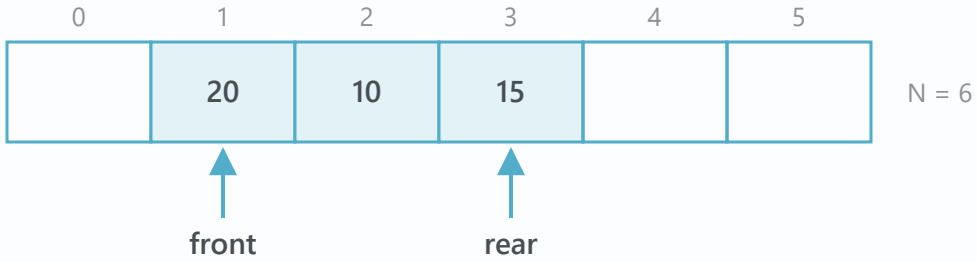
Keyboard input, network packets, video streams: a buffer evens out the speed difference between producer and consumer.

Breadth-First Search (BFS)

Discovered nodes wait their turn in a queue, so the graph is explored level by level.

A Queue in an Array

- The items live in an array, and two indices mark the ends of the queue:
  - **front**: index of the first item.
  - **rear**: index of the last item.
- Enqueue increments `rear`; dequeue increments `front`. Items never move.
- Simple and fast, but the array has a **fixed capacity**.



The item at index 0 was dequeued earlier. Its slot is free now, but a simple array queue cannot reuse it.

The Trouble with a Simple Array Queue



empty: front = 0, rear = −1

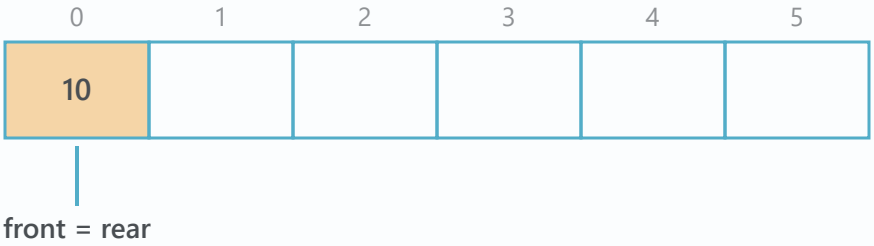
Empty queue, capacity N = 6. `rear = -1` : no items yet.

Step 1 / 11

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



enqueue(10) → rear = 0

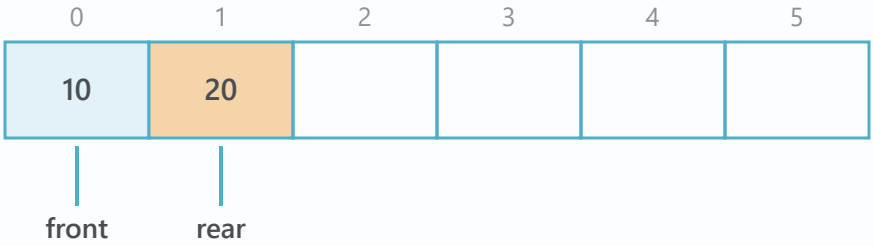
`enqueue(10)` : `rear++` → 0 → data[0].

Step 2 / 11

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



enqueue(20) → rear = 1

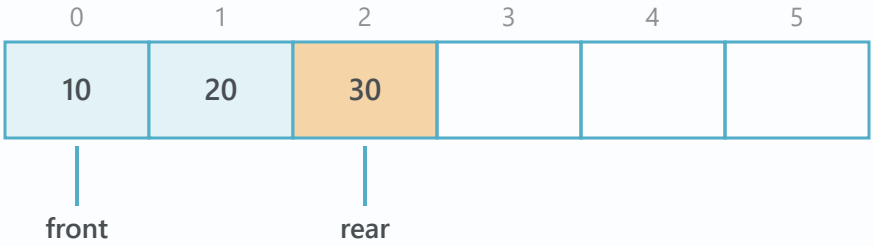
`enqueue(20)` : `rear++` → 1 → data[1].

Step 3 / 11

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



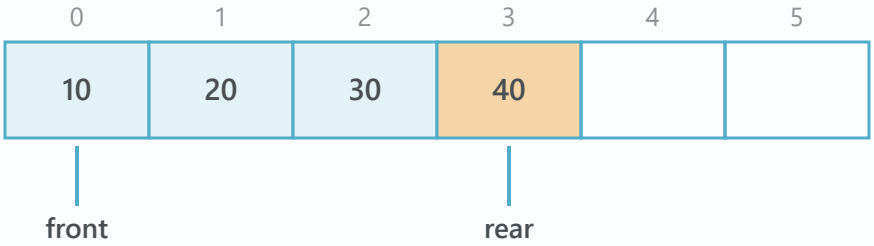
enqueue(30) → rear = 2

`enqueue(30)` : `rear++` → 2 → data[2].

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



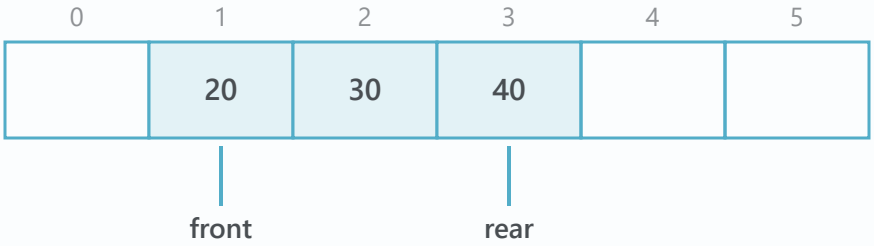
enqueue(40) → rear = 3

`enqueue(40)` : `rear++` → 3 → data[3].

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



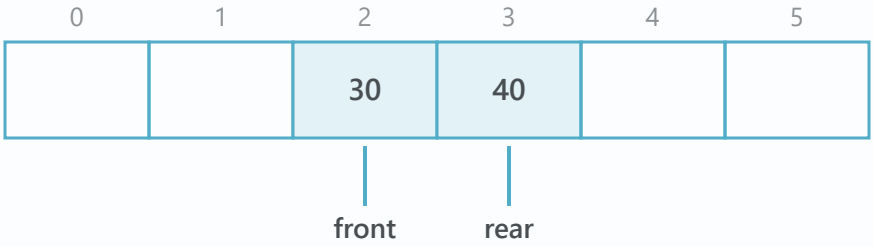
dequeue() → 10, front = 1

`dequeue()` → 10: `front++` → 1. The freed slot is never reused.

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



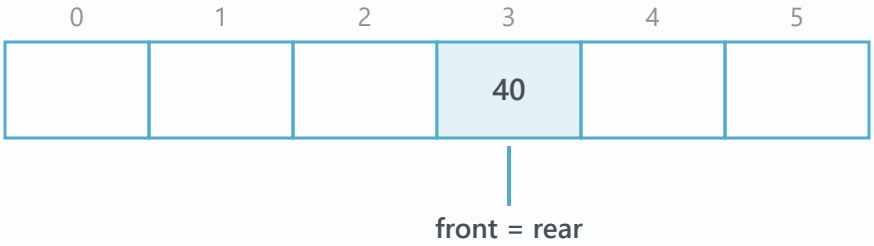
dequeue() → 20, front = 2

`dequeue()` → 20: `front++` → 2. The freed slot is never reused.

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



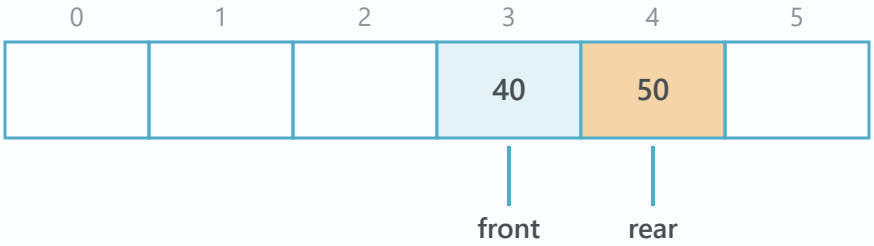
dequeue() → 30, front = 3

`dequeue()` → 30: `front++` → 3. The freed slot is never reused.

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



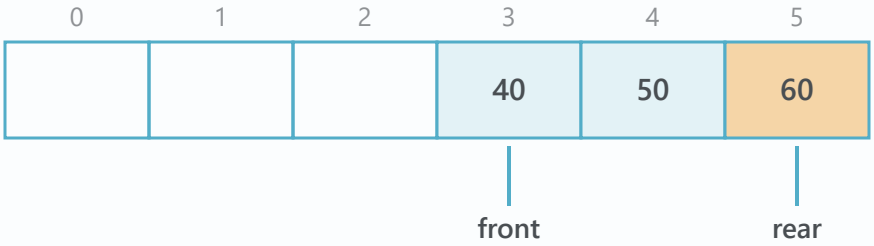
enqueue(50) → rear = 4

`enqueue(50)` : rear = 4.

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



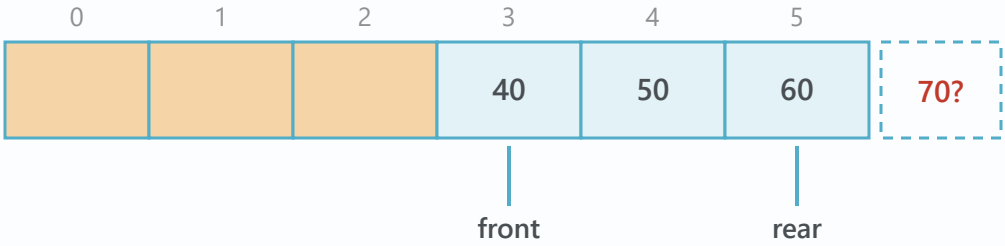
enqueue(60) → rear = 5

`enqueue(60)` : rear = 5 = N − 1, the last index of the array.

Fix 1: Shifting
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array
Indices wrap around to the start → every operation is O(1).

The Trouble with a Simple Array Queue



enqueue(70): rear + 1 = 6 = N → “full”!

“Full” with only 3 items: the 3 orange slots are wasted. Shifting costs O(n); the fix is a circular queue.

Fix 1: Shifting

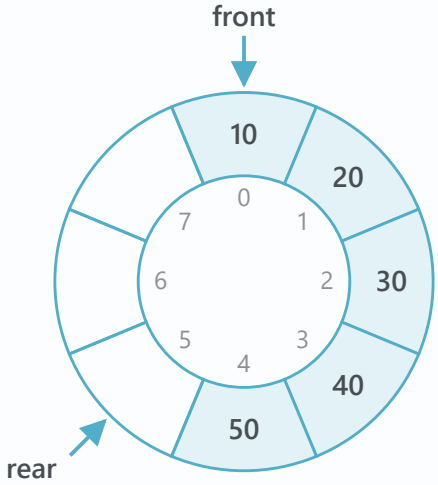
Shift the items left on every dequeue → dequeue becomes O(n).

Fix 2: A circular array

Indices wrap around to the start → every operation is O(1).

The Circular Queue

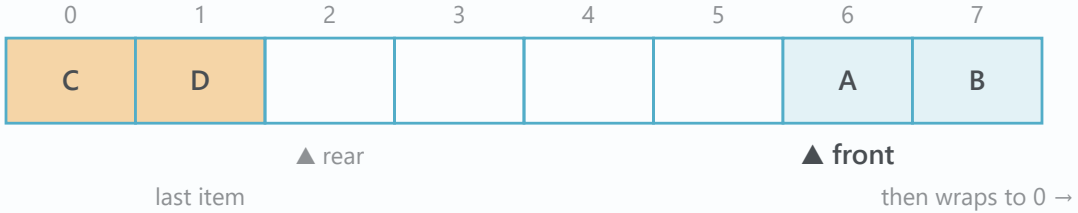
- FIFO like an ordinary queue, but after the last slot of the array comes the **first slot** again.
- The two ends of the array are logically joined into a **ring**.
- Indices advance **modulo N**: `(i + 1) % N`.
- Freed slots are reused, so items never need shifting.
- Also known as a **ring buffer**.



Here **rear** is the free slot where the next item will go: rear = (front + size) % N = (0 + 5) % 8 = 5.

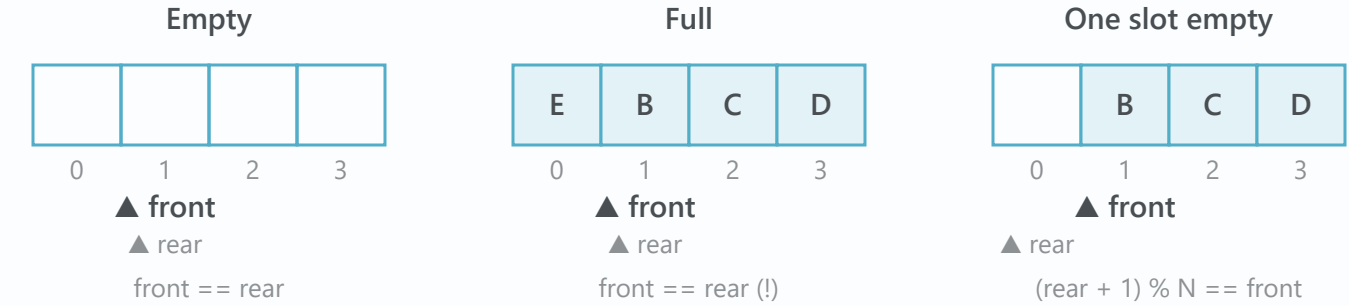
Moving the Indices with Mod

What	Formula	Example: N = 8, front = 6, size = 4
Next free slot (where to insert)	`(front + size) % N`	(6 + 4) % 8 = **2**
Last item	`(front + size - 1) % N`	(6 + 3) % 8 = **1**
k-th item from the front (k = 0, 1, ...)	`(front + k) % N`	k = 2 → (6 + 2) % 8 = **0**
front after a dequeue	`(front + 1) % N`	(6 + 1) % 8 = **7**



Front to rear: A (6), B (7), C (0), D (1). The mod turns the jump from the end of the array back to the start into one expression.

Full or Empty?



With only the front and rear indices, an **empty** and a **full** queue look the same: both indices point to the same slot. Two common fixes:

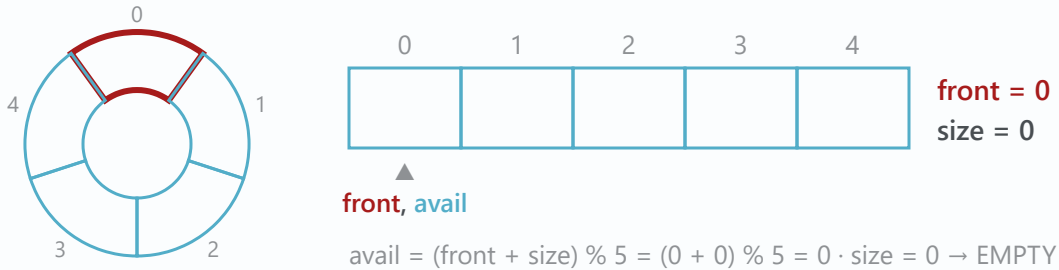
1. An item counter (size)

Empty: `size == 0` · Full: `size == N`
All N slots are used. This is the method used in this course.

2. Leave one slot empty

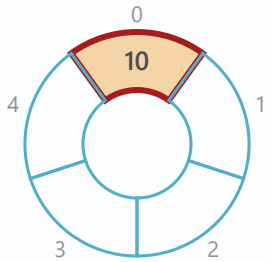
Empty: `front == rear` · Full: `(rear + 1) % N == front`
No counter is needed, but at most N − 1 items fit.

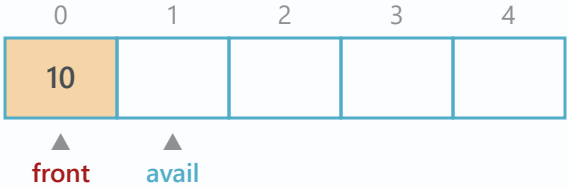
Circular Queue, Step by Step (N = 5)



Empty queue: size = 0. On the ring, the red-edged slot is **front**; in the array, ▲avail marks the next write position.

Circular Queue, Step by Step (N = 5)



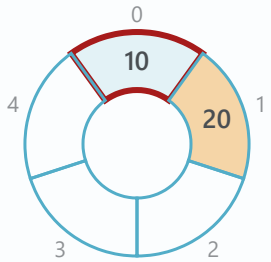


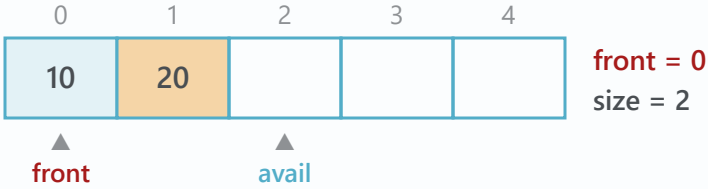
front = 0
size = 1

write slot = (0 + 0) % 5 = 0

`enqueue(10)` → data[0]; size = 1.

Circular Queue, Step by Step (N = 5)



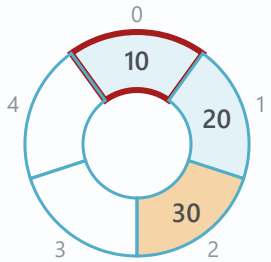


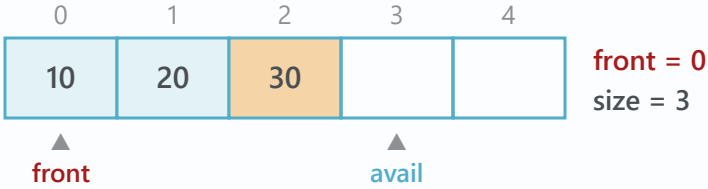
front = 0
size = 2

write slot = (0 + 1) % 5 = 1

`enqueue(20)` → data[1]; size = 2.

Circular Queue, Step by Step (N = 5)



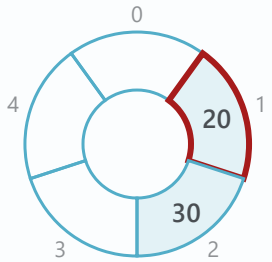


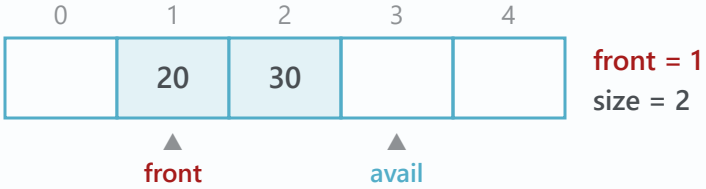
front = 0
size = 3

write slot = (0 + 2) % 5 = 2

`enqueue(30)` → data[2]; size = 3.

Circular Queue, Step by Step (N = 5)



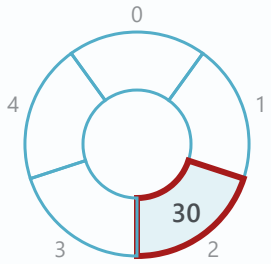


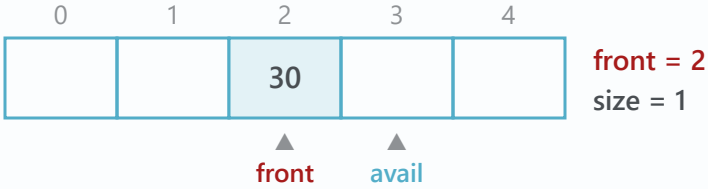
front = 1
size = 2

front = (0 + 1) % 5 = 1

`dequeue()` → 10; front = (0 + 1) % 5 = 1. The items do not move.

Circular Queue, Step by Step (N = 5)



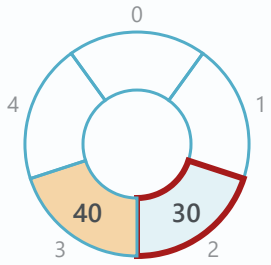


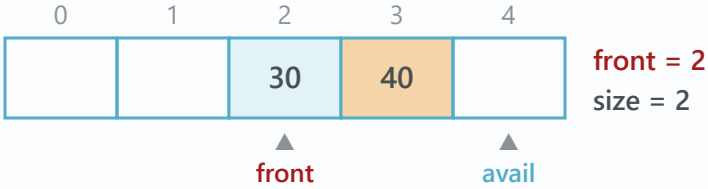
front = 2
size = 1

front = (1 + 1) % 5 = 2

`dequeue()` → 20; front = (1 + 1) % 5 = 2. The items do not move.

Circular Queue, Step by Step (N = 5)



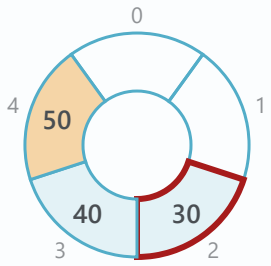


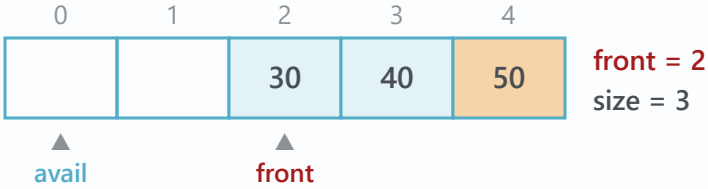
front = 2
size = 2

write slot = (2 + 1) % 5 = 3

`enqueue(40)` → data[3].

Circular Queue, Step by Step (N = 5)



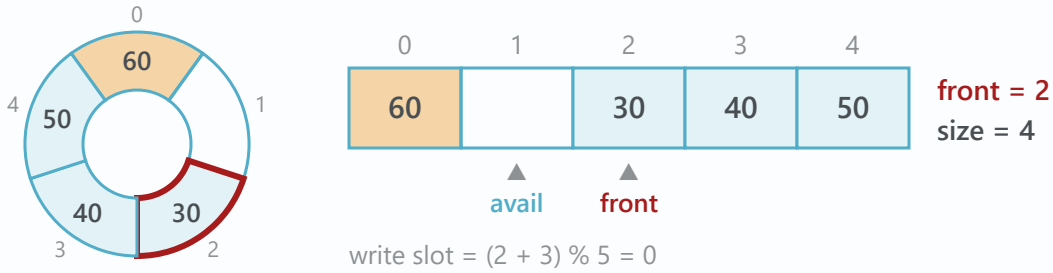


front = 2
size = 3

write slot = (2 + 2) % 5 = 4

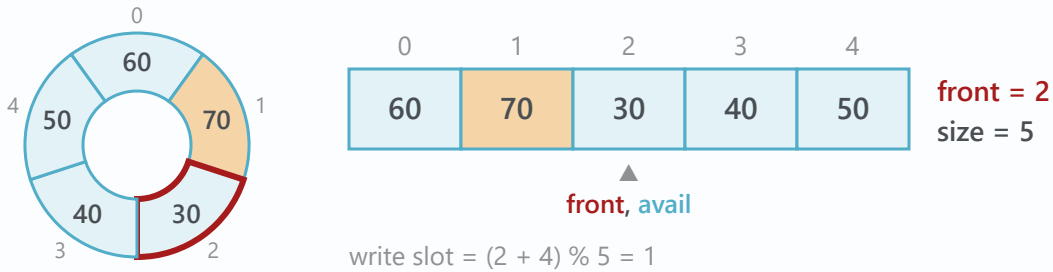
`enqueue(50)` → data[4].

Circular Queue, Step by Step (N = 5)



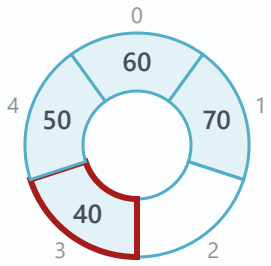
`enqueue(60)` → data[0]: the index **wraps around**, reusing a slot that a simple array queue would call “full.”

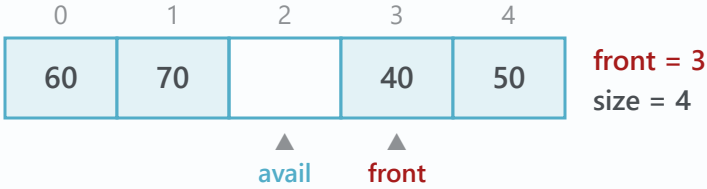
Circular Queue, Step by Step (N = 5)



`enqueue(70)` → data[1]. size == N → **full**; front == avail, but thanks to size the queue is not mistaken for empty.

Circular Queue, Step by Step (N = 5)



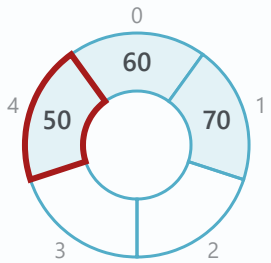


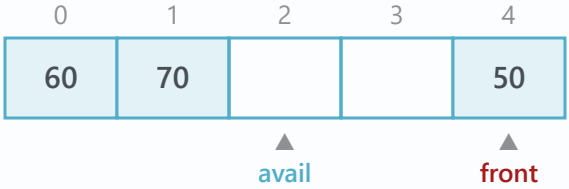
front = 3
size = 4

front = (2 + 1) % 5 = 3

`dequeue()` → 30; front = 3.

Circular Queue, Step by Step (N = 5)



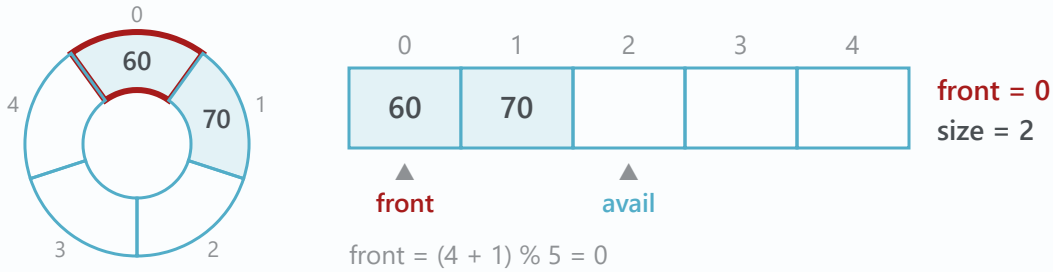


front = 4
size = 3

front = (3 + 1) % 5 = 4

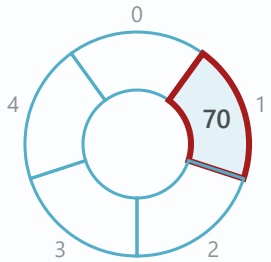
`dequeue()` → 40; front = 4.

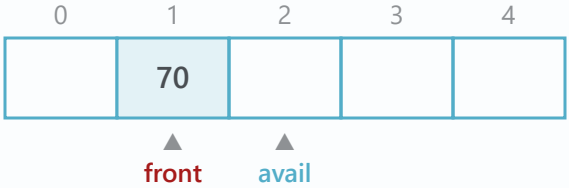
Circular Queue, Step by Step (N = 5)



`dequeue()` → 50; front = 0. (4 + 1) % 5 = 0, so front wraps around too.

Circular Queue, Step by Step (N = 5)



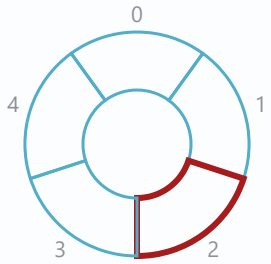


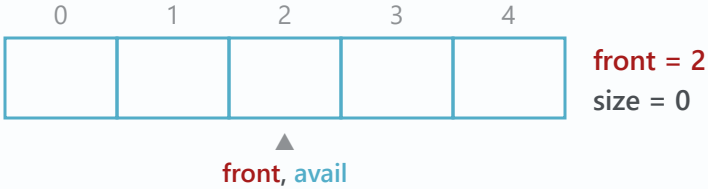
front = 1
size = 1

front = (0 + 1) % 5 = 1

`dequeue()` → 60; front = 1.

Circular Queue, Step by Step (N = 5)





front = (1 + 1) % 5 = 2

`dequeue()` → 70; front = 2. size = 0 → the queue is **empty** again. Items left in the order 30, 40, 50, 60, 70, the order they arrived (FIFO).

Circular Queue: Class and Status Queries

```
public class CircularQueue<E> {
    private E[] data;       // stores the items
    private int front = 0;    // index of the first item
    private int size = 0;  // number of items

    @SuppressWarnings("unchecked")
    public CircularQueue(int capacity) {
        data = (E[]) new Object[capacity];
    }
    public int size()      { return size; }
    public boolean isEmpty()  { return size == 0; }
    public boolean isFull() { return size == data.length; }
    public E first()          { return isEmpty() ? null : data[front]; }
    // enqueue(), dequeue()
}
```

The rear index is not stored but computed when needed as `(front + size) % data.length`, so two fields can never disagree.

Circular Queue: `enqueue` and `dequeue`

```
public void enqueue(E item) {           // enqueue
    if (isFull())
        throw new IllegalStateException("full");
    int avail = (front + size) % data.length;
    data[avail] = item;
    size++;
}
```

```
public E dequeue() {                    // dequeue
    if (isEmpty())
        throw new IllegalStateException("empty");
    E item = data[front];
    data[front] = null;                 // drop the refere
    front = (front + 1) % data.length;
    size--;
    return item;
}
```

Where to insert

N = 5, front = 3, size = 2

`avail = (3 + 2) % 5 = 0`

Advancing the front

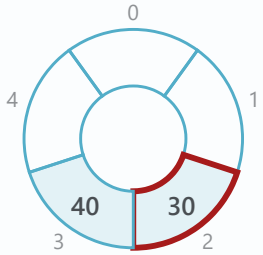
N = 5, front = 4

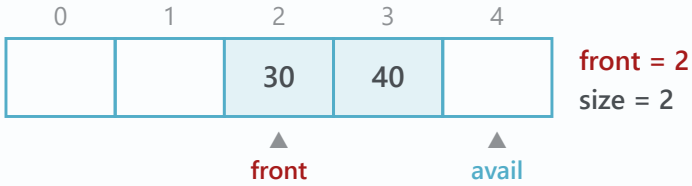
`front = (4 + 1) % 5 = 0`

Cost

Both operations take a constant number of steps: **O(1)**. No loops, no shifting.

Code Trace: Circular `enqueue()`





Start: data.length = 5, **front = 2**, **size = 2** (30 and 40). Next we enqueue 50, 60, 70, 80.

Step 1 / 15

```
▶ public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

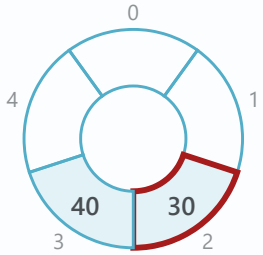
VARIABLES

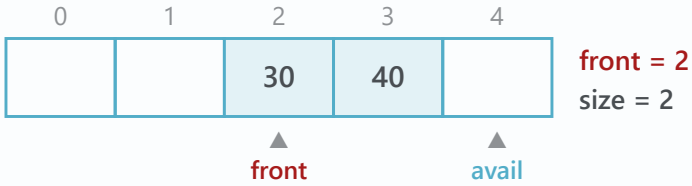
`item = –` `front = 2` `size = 2` `avail = –` `data.length = 5`

CALL STACK

`main()`

Code Trace: Circular `enqueue()`





front = 2
size = 2

`enqueue(50)` : `isFull()` → size 2 == 5? **no**.

Step 2 / 15

```
1 public void enqueue(E item) {
▶   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

VARIABLES

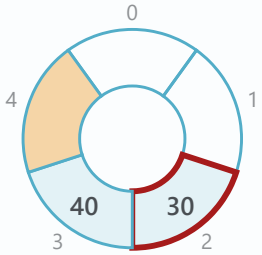
`item = 50` `front = 2` `size = 2` `avail = –` `data.length = 5`

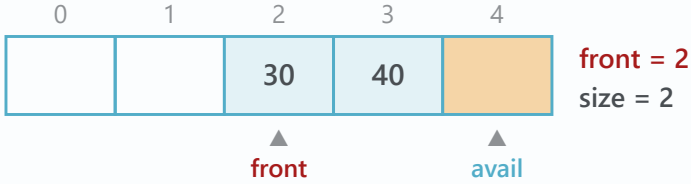
CALL STACK

`enqueue(50)`

`main()`

Code Trace: Circular `enqueue()`





avail = (2 + 2) % 5 = 4 % 5 = 4.

Step 3 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
▶   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

VARIABLES

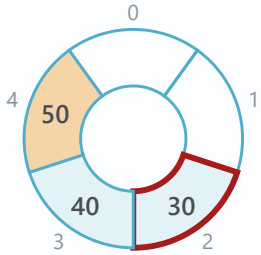
`item = 50` `front = 2` `size = 2` `avail = 4` `data.length = 5`

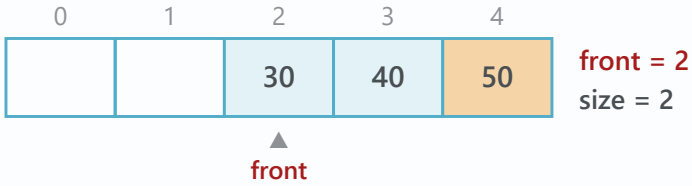
CALL STACK

`enqueue(50)`

`main()`

Code Trace: Circular `enqueue()`





`data[4] = 50`. The other items do not move.

Step 4 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
▶   data[avail] = item;
6   size++;
7 }
```

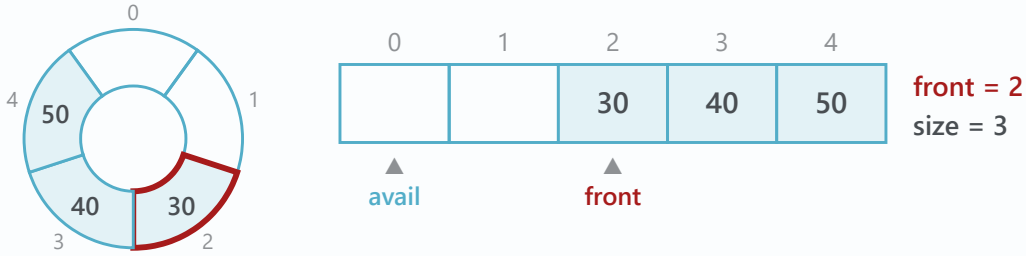
VARIABLES

`item = 50` `front = 2` `size = 2` `avail = 4` `data.length = 5`

CALL STACK

- **`enqueue(50)`**
- `main()`

Code Trace: Circular `enqueue()`



`size++` → 3. front is unchanged (data[2]).

Step 5 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
▶   size++;
7 }
```

VARIABLES

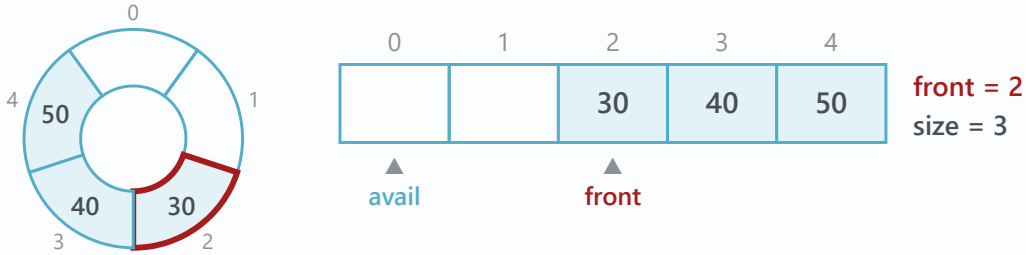
`item = 50` `front = 2` `size = 3` `avail = 4` `data.length = 5`

CALL STACK

`enqueue(50)`

`main()`

Code Trace: Circular `enqueue()`



`enqueue(60)` : `isFull()` → size 3 == 5? **no**.

Step 6 / 15

```
1 public void enqueue(E item) {
▶   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

VARIABLES

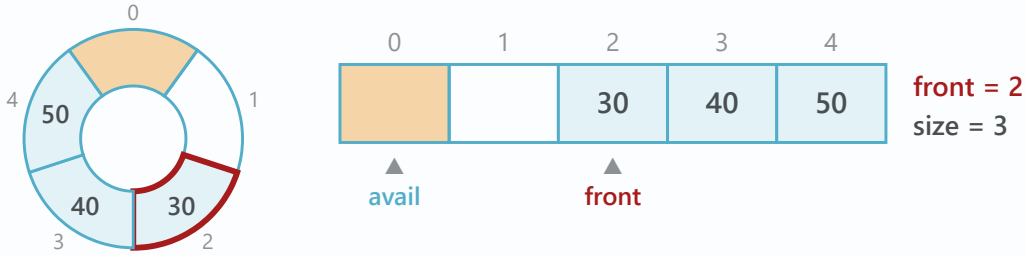
`item = 60` `front = 2` `size = 3` `avail = –` `data.length = 5`

CALL STACK

`enqueue(60)`

`main()`

Code Trace: Circular `enqueue()`



avail = (2 + 3) % 5 = 5 % 5 = 0 → past the end of the array, the index **wraps around**.

Step 7 / 15

```
1  public void enqueue(E item) {
2    if (isFull())
3      throw new IllegalStateException("full");
▶    int avail = (front + size) % data.length;
5    data[avail] = item;
6    size++;
7  }
```

VARIABLES

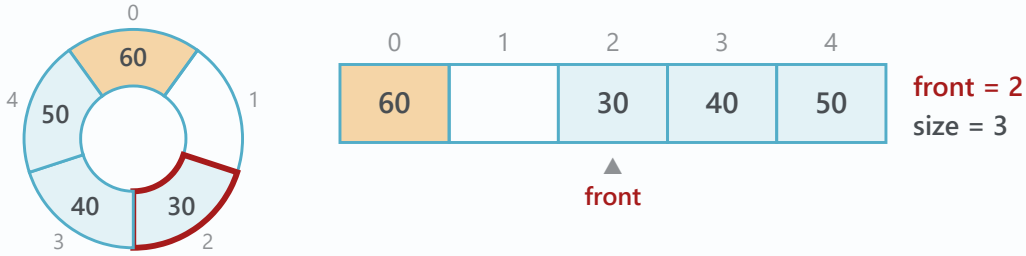
`item = 60` `front = 2` `size = 3` `avail = 0` `data.length = 5`

CALL STACK

`enqueue(60)`

`main()`

Code Trace: Circular `enqueue()`



`data[0] = 60`. The other items do not move.

Step 8 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
▶   data[avail] = item;
6   size++;
7 }
```

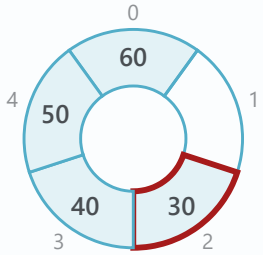
VARIABLES

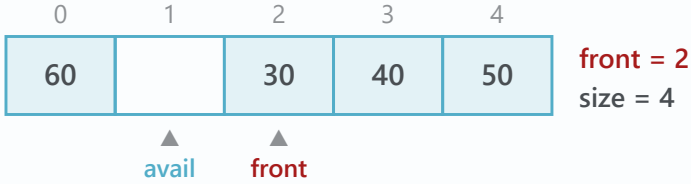
`item = 60` `front = 2` `size = 3` `avail = 0` `data.length = 5`

CALL STACK

- **`enqueue(60)`**
- `main()`

Code Trace: Circular `enqueue()`





front = 2
size = 4

`size++` → 4. front is unchanged (data[2]).

Step 9 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
▶   size++;
7 }
```

VARIABLES

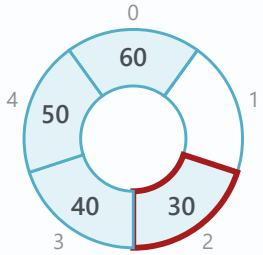
`item = 60` `front = 2` `size = 4` `avail = 0` `data.length = 5`

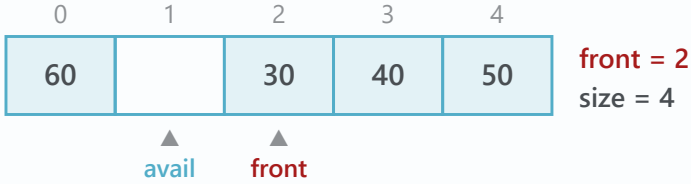
CALL STACK

`enqueue(60)`

`main()`

Code Trace: Circular `enqueue()`





front = 2
size = 4

`enqueue(70)` : `isFull()` → size 4 == 5? **no**.

Step 10 / 15

```
1 public void enqueue(E item) {
▶   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

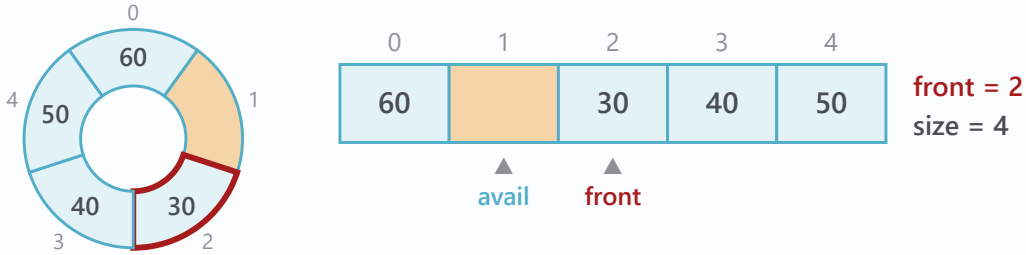
VARIABLES

`item = 70` `front = 2` `size = 4`
`avail = –` `data.length = 5`

CALL STACK

`enqueue(70)`
`main()`

Code Trace: Circular `enqueue()`



avail = (2 + 4) % 5 = 6 % 5 = 1 → past the end of the array, the index **wraps around**.

Step 11 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
▶   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

VARIABLES

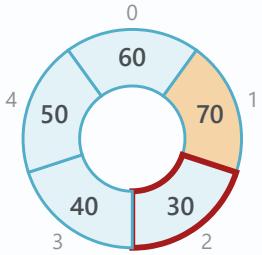
`item = 70` `front = 2` `size = 4` `avail = 1` `data.length = 5`

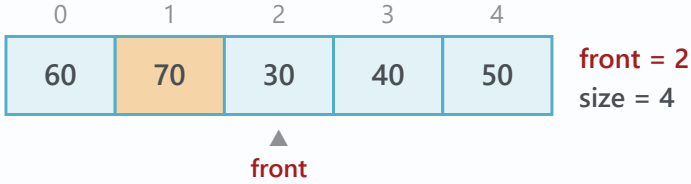
CALL STACK

`enqueue(70)`

`main()`

Code Trace: Circular `enqueue()`





`data[1] = 70`. The other items do not move.

Step 12 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
▶   data[avail] = item;
6   size++;
7 }
```

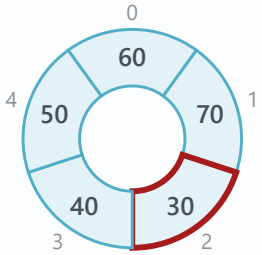
VARIABLES

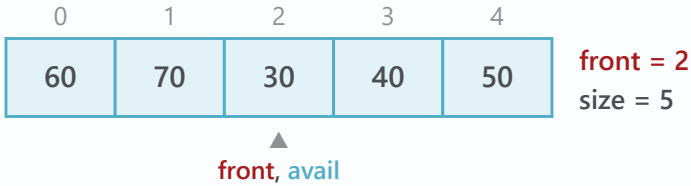
`item = 70` `front = 2` `size = 4`
`avail = 1` `data.length = 5`

CALL STACK

`enqueue(70)`
`main()`

Code Trace: Circular `enqueue()`





`size++` → 5. front is unchanged (data[2]). The queue is now full.

Step 13 / 15

```
1  public void enqueue(E item) {
2    if (isFull())
3      throw new IllegalStateException("full");
4    int avail = (front + size) % data.length;
5    data[avail] = item;
▶    size++;
7  }
```

VARIABLES

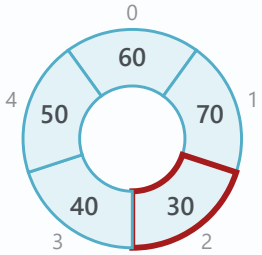
`item = 70` `front = 2` `size = 5` `avail = 1` `data.length = 5`

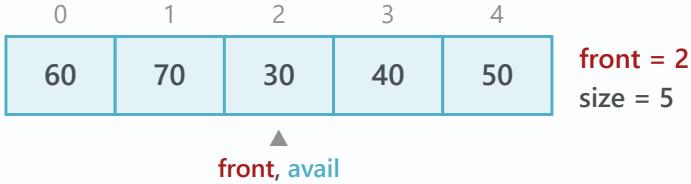
CALL STACK

`enqueue(70)`

`main()`

Code Trace: Circular `enqueue()`





`enqueue(80)` : `isFull()` → size 5 == 5? **yes**.

Step 14 / 15

```
1 public void enqueue(E item) {
▶   if (isFull())
3     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

VARIABLES

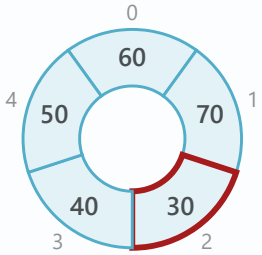
`item = 80` `front = 2` `size = 5` `avail = –` `data.length = 5`

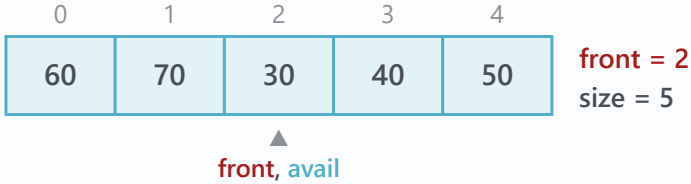
CALL STACK

`enqueue(80)`

`main()`

Code Trace: Circular `enqueue()`





The queue is full → `IllegalStateException("full")` is thrown and the array is unchanged. All 5 slots were used; none was wasted.

Step 15 / 15

```
1 public void enqueue(E item) {
2   if (isFull())
▶     throw new IllegalStateException("full");
4   int avail = (front + size) % data.length;
5   data[avail] = item;
6   size++;
7 }
```

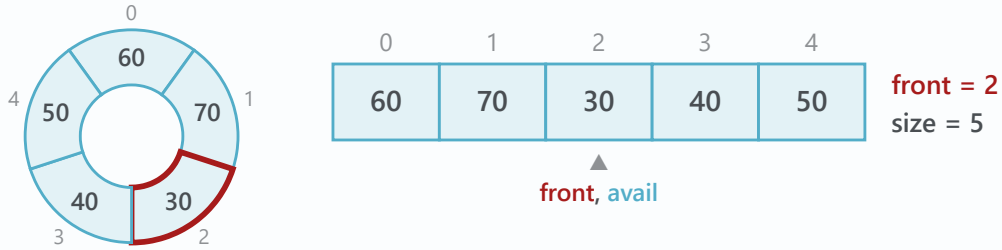
VARIABLES

`item = 80` `front = 2` `size = 5` `avail = –` `data.length = 5`

CALL STACK

- `enqueue(80)`
- `main()`

Code Trace: Circular `dequeue()`



Start: a full queue, front = 2. Front to rear: 30, 40, 50, 60, 70. We call `dequeue()` three times.

Step 1 / 16

```
▶ public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

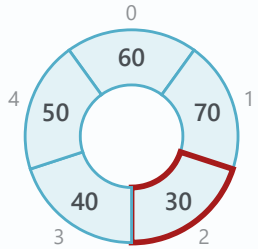
VARIABLES

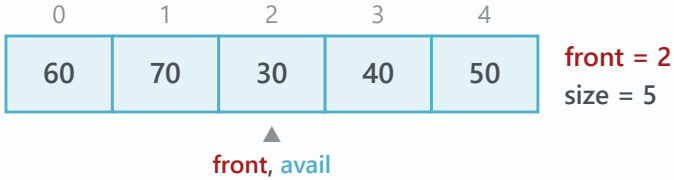
`item = –` `front = 2` `size = 5` `data.length = 5`

CALL STACK

`main()`

Code Trace: Circular `dequeue()`





`isEmpty()` : size = 5 ≠ 0 → not empty.

Step 2 / 16

```
1 public E dequeue() {
▶   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

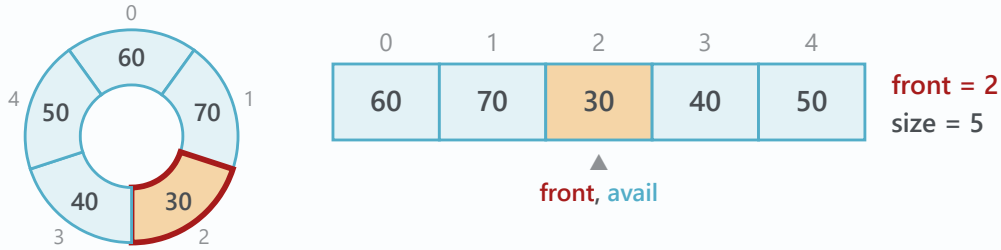
VARIABLES

`item = –` `front = 2` `size = 5` `data.length = 5`

CALL STACK

- **dequeue()**
- main()

Code Trace: Circular `dequeue()`



item = data[2] = 30: the front (oldest) item.

Step 3 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
▶   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

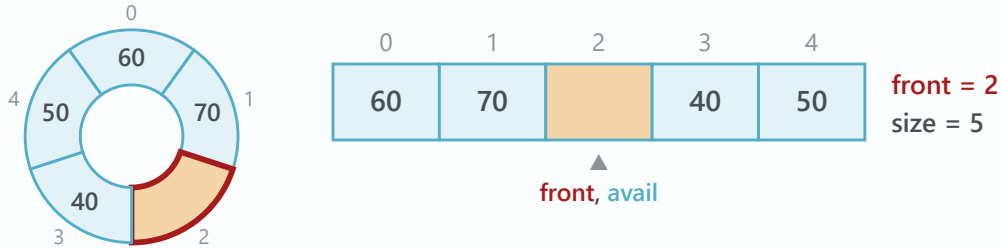
`item = 30` `front = 2` `size = 5` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



`data[2] = null` : the slot is cleared, so the garbage collector can reclaim the object.

Step 4 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
▶   data[front] = null;            // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

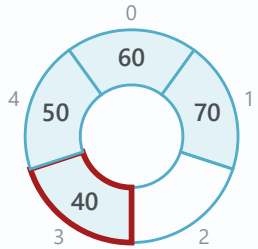
`item = 30` `front = 2` `size = 5` `data.length = 5`

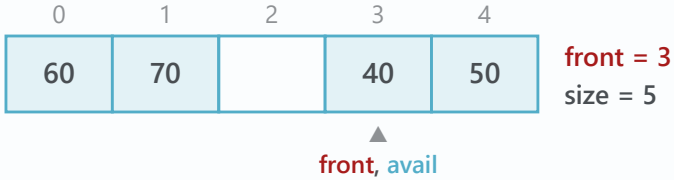
CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`





front = (2 + 1) % 5 = 3.

Step 5 / 16

```
1  public E dequeue() {
2    if (isEmpty())
3      throw new IllegalStateException("empty");
4    E item = data[front];
5    data[front] = null;           // drop the reference
▶    front = (front + 1) % data.length;
7    size--;
8    return item;
9  }
```

VARIABLES

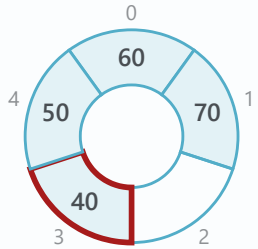
`item = 30` `front = 3` `size = 5` `data.length = 5`

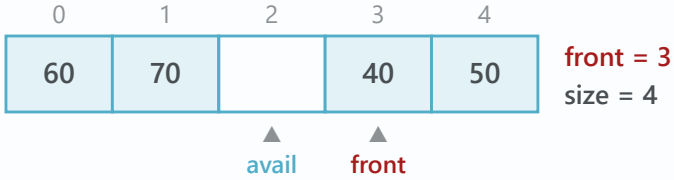
CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`





`size--` → 4; **return 30**.

Step 6 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
▶   size--;
▶   return item;
9 }
```

VARIABLES

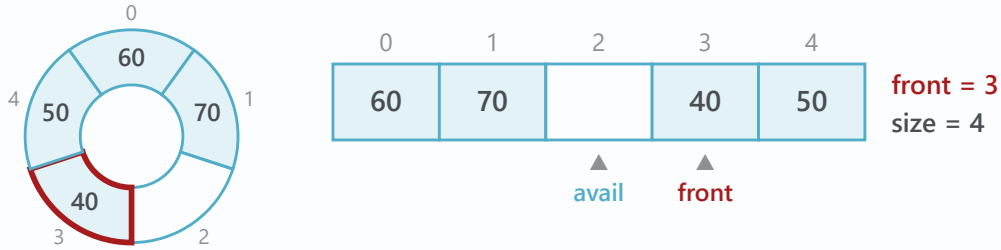
`item = 30` `front = 3` `size = 4` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



`isEmpty()` : size = 4 ≠ 0 → not empty.

Step 7 / 16

```
1 public E dequeue() {
▶   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

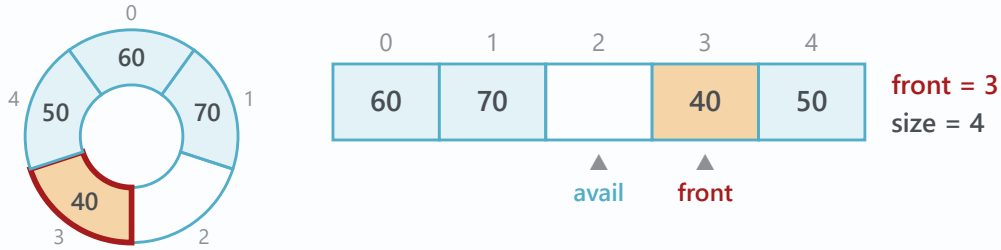
`item = –` `front = 3` `size = 4` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



item = data[3] = 40: the front (oldest) item.

Step 8 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
▶   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

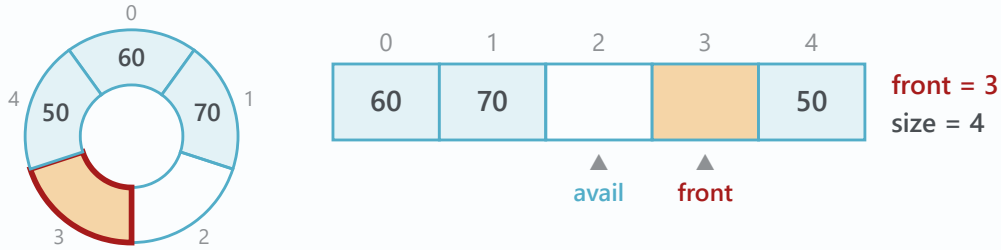
`item = 40` `front = 3` `size = 4` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



`data[3] = null` : the slot is cleared, so the garbage collector can reclaim the object.

Step 9 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
▶   data[front] = null;            // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

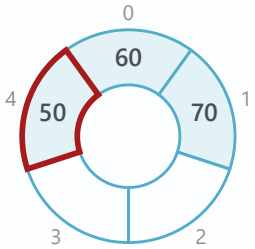
`item = 40` `front = 3` `size = 4` `data.length = 5`

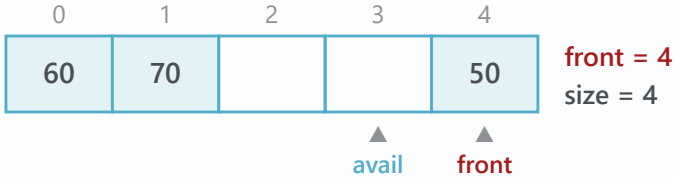
CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`





front = (3 + 1) % 5 = 4.

Step 10 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
▶   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

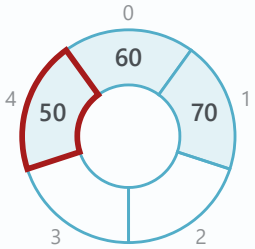
`item = 40` `front = 4` `size = 4` `data.length = 5`

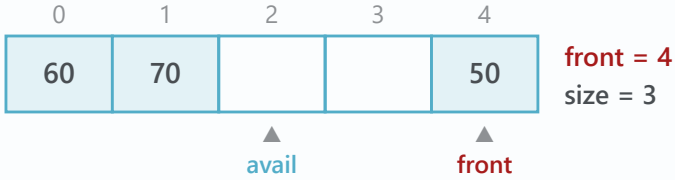
CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`





`size--` → 3; **return 40**.


```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
▶   size--;
▶   return item;
9 }
```

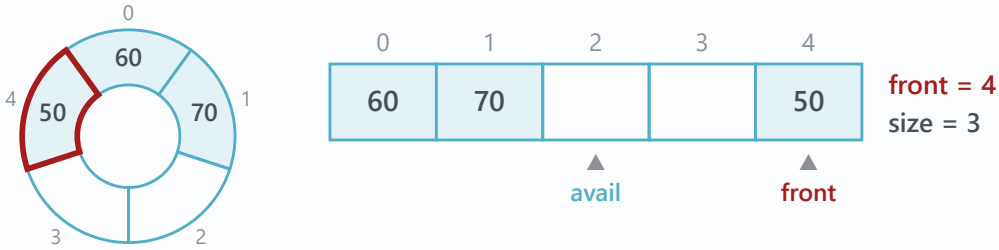
VARIABLES

`item = 40` `front = 4` `size = 3` `data.length = 5`

CALL STACK

- **`dequeue()`**
- `main()`

Code Trace: Circular `dequeue()`



`isEmpty()` : size = 3 ≠ 0 → not empty.

Step 12 / 16

```
1 public E dequeue() {
▶   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

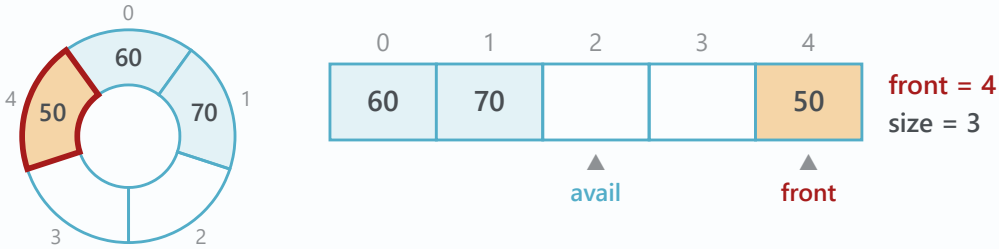
`item = –` `front = 4` `size = 3` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



item = data[4] = 50: the front (oldest) item.

Step 13 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

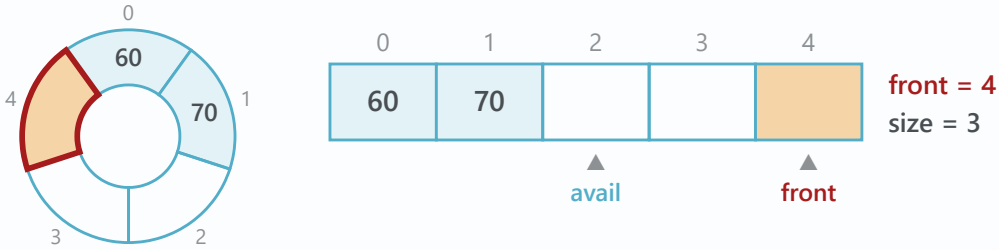
`item = 50` `front = 4` `size = 3` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



`data[4] = null` : the slot is cleared, so the garbage collector can reclaim the object.

Step 14 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
▶   data[front] = null;            // drop the reference
6   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

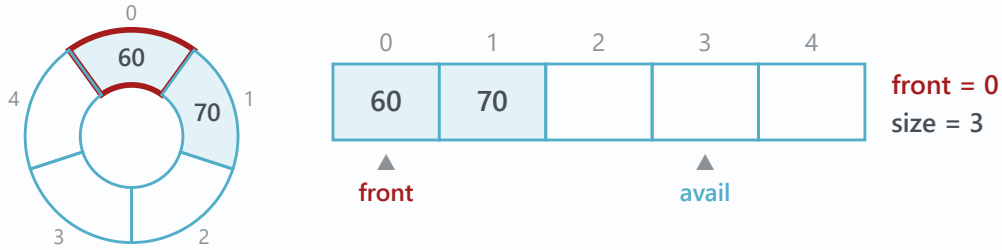
`item = 50` `front = 4` `size = 3` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



front = (4 + 1) % 5 = 0 → 5 % 5 = 0: front **wraps around**.

Step 15 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
▶   front = (front + 1) % data.length;
7   size--;
8   return item;
9 }
```

VARIABLES

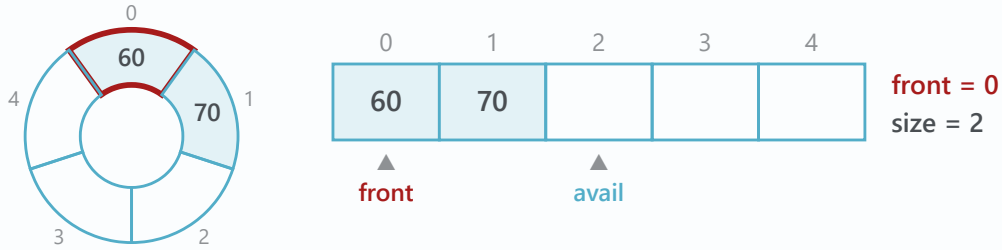
`item = 50` `front = 0` `size = 3` `data.length = 5`

CALL STACK

`dequeue()`

`main()`

Code Trace: Circular `dequeue()`



`size--` → 2; **return 50**. A constant number of steps: **O(1)**.

Step 16 / 16

```
1 public E dequeue() {
2   if (isEmpty())
3     throw new IllegalStateException("empty");
4   E item = data[front];
5   data[front] = null;           // drop the reference
6   front = (front + 1) % data.length;
▶   size--;
▶   return item;
9 }
```

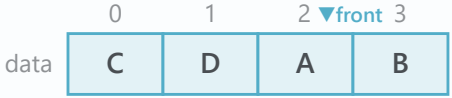
VARIABLES

`item = 50` `front = 0` `size = 2` `data.length = 5`

CALL STACK

- **dequeue()**
- main()

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

order: A B C D

Full queue (front = 2): the order is A, B, C, D, but the array holds **C, D, A, B**. Now enqueue(E).

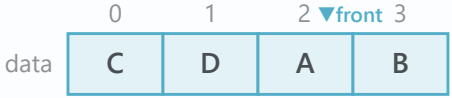
Step 1 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
▶ public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = **E** | front = **2** | size = **4** | data = **[C, D, A, B]**

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

full!

4 == 4 → full. Instead of failing, grow to **twice** the capacity.

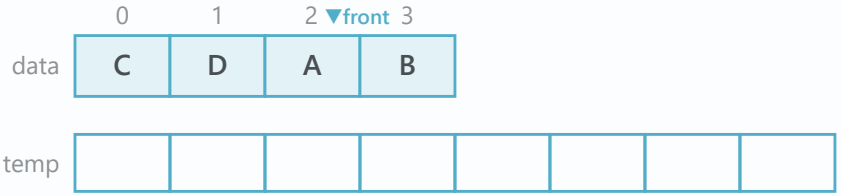
Step 2 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
▶    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = **E** | front = **2** | size = **4** | data = **[C, D, A, B]**

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

A new array of 8 cells is allocated.

Step 3 / 11

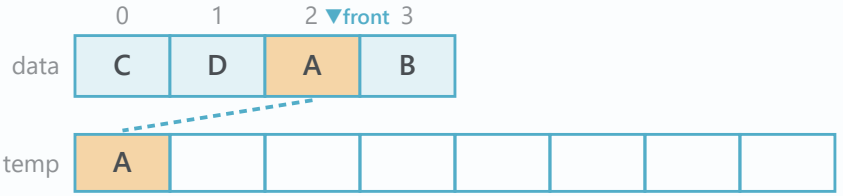
```
1  private void resize(int newCap) {
▶    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = E | front = 2 | size = 4 | data = [C, D, A, B] | newCap = 8

tem [null, null, null, null, null,
p = null, null, null]

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

k = 0: temp[0] = data[(2 + 0) % 4] = data[2] = **A**.

Step 4 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
▶    for (int k = 0; k < size; k++)
▶      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

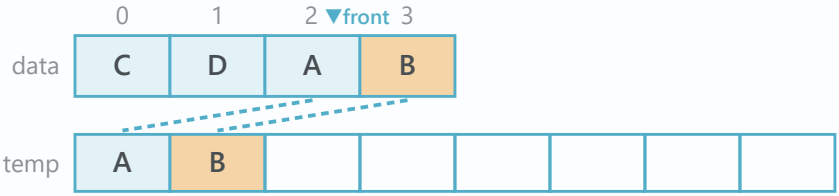
item = **E** | front = **2** | size = **4**

data = **[C, D, A, B]** | newCap = **8**

k = **0**

temp = **[A, null, null, null, null, null, null, null]**

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

k = 1: temp[1] = data[(2 + 1) % 4] = data[3] = **B**.

Step 5 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
▶    for (int k = 0; k < size; k++)
▶      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

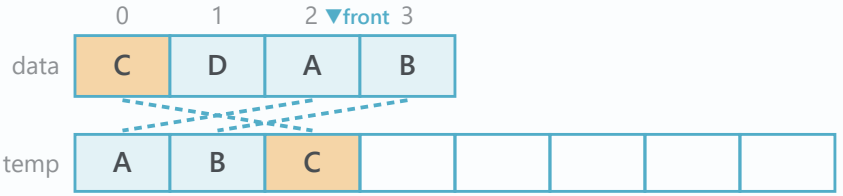
item = **E** · front = **2** · size = **4**

data = **[C, D, A, B]** · newCap = **8**

k = **1**

temp = **[A, B, null, null, null, null, null, null]**

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

k = 2: temp[2] = data[(2 + 2) % 4] = data[0] = **C**. The mod wrapped around to the start.


```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
▶    for (int k = 0; k < size; k++)
▶      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

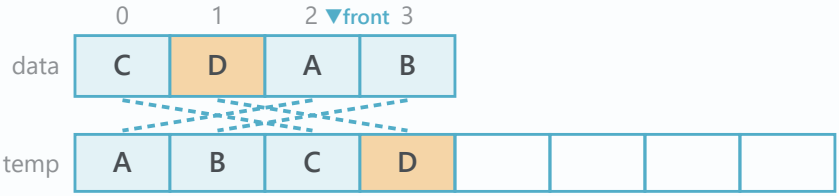
item = **E** | front = **2** | size = **4**

data = **[C, D, A, B]** | newCap = **8**

k = **2**

temp = **[A, B, C, null, null, null, null, null]**

Code Trace: Growing a Full Circular Queue



front = 2 · size = 4 · N = 4

k = 3: temp[3] = data[(2 + 3) % 4] = data[1] = **D**.

Step 7 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
▶    for (int k = 0; k < size; k++)
▶      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = **E** | front = **2** | size = **4** | data = **[C, D, A, B]** | newCap = **8** | k = **3**

temp = **[A, B, C, D, null, null, null, null]**

Code Trace: Growing a Full Circular Queue



data = temp: the items are now **in order** (Arrays.copyOf would have kept them scrambled).

Step 8 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
▶    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = **E** | front = **2** | size = **4**

data = **[A, B, C, D, null, null, null, null]**

newCap = **8**

Code Trace: Growing a Full Circular Queue



`front = 0` : the front item (A) is now in data[0].

Step 9 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
▶    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
11   size++;
12 }
```

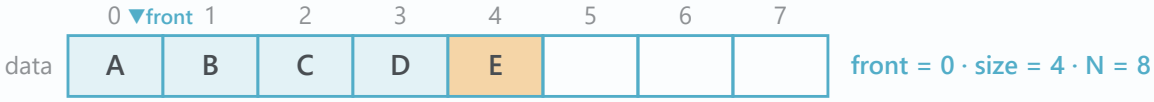
VARIABLES

item = **E** | front = **0** | size = **4**

data = **[A, B, C, D, null, null, null, null]**

newCap = **8**

Code Trace: Growing a Full Circular Queue



Back in enqueue: (0 + 4) % 8 = **4** → data[4] = E.

Step 10 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
▶    data[(front + size) % data.length] = item;
11   size++;
12 }
```

VARIABLES

item = **E** | front = **0** | size = **4**

data = **[A, B, C, D, E, null, null, null]**

Code Trace: Growing a Full Circular Queue



size → 5. Growing costs O(n), but because the capacity doubles, enqueue is **amortized O(1)**.

Step 11 / 11

```
1  private void resize(int newCap) {
2    E[] temp = (E[]) new Object[newCap];
3    for (int k = 0; k < size; k++)
4      temp[k] = data[(front + k) % data.length];
5    data = temp;
6    front = 0;
7  }
8  public void enqueue(E item) {
9    if (size == data.length) resize(2 * data.length);
10   data[(front + size) % data.length] = item;
▶    size++;
12 }
```

VARIABLES

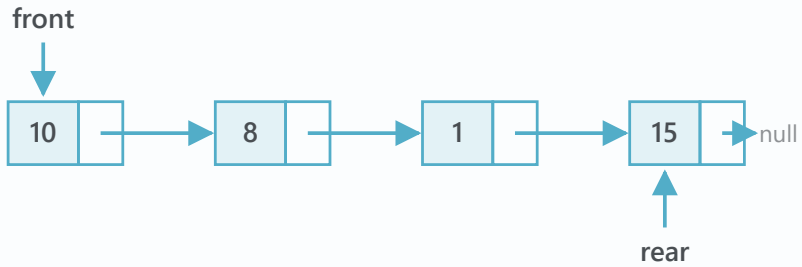
item = **E** front = **0** size = **5**

data = **[A, B, C, D, E, null, null, null]**

A Queue in a Linked List

- Each queue item is a **node** (data plus a next reference).
- `front` marks the head and `rear` the tail: insert at the tail, remove at the head.
- The size is dynamic; there is no capacity limit (no full queue).
- Each node costs one extra reference and an object allocation.

```
class Node {
    int data;
    Node next;
    Node(int data) {
        this.data = data;
    }
}
// fields:
Node front, rear;
int size;
```



Why insert at the tail and remove at the head? In a singly linked list, removing the head is O(1), but removing the tail means finding its predecessor: O(n).

Enqueue and Dequeue Algorithms

enqueue(x)

1. Is the queue full? (fixed-capacity array only)
2. If so, report an **overflow** and stop.
3. Otherwise, write x at the rear position.
4. Advance the rear to the next position.
5. Increment the item count and report success.

dequeue()

1. Is the queue empty?
2. If so, report an **underflow** and stop.
3. Otherwise, take the front item.
4. Advance the front to the next item.
5. Decrement the item count and return the item.

Step	Circular array	Linked list
Write at the rear	`data[(front + size) % N] = x`	`rear.next = temp; rear = temp`
Advance the front	`front = (front + 1) % N`	`front = front.next`
Full check	`size == N`	not needed (while memory lasts)

Code Trace: Linked Queue `enqueue`

front = null · rear = null · size = 0

Empty queue: front = rear = null. We call `enqueue(10)`, `enqueue(15)`, `enqueue(20)` in turn.

Step 1 / 16

```
▶ public void enqueue(int data) {
2     Node temp = new Node(data);
3     if (isEmpty()) {
4         front = temp;
5     } else {
6         rear.next = temp;
7     }
8     rear = temp;
9     size++;
10 }
```

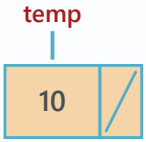
VARIABLES

`data = –` `temp = –` `front = null` `rear = null` `size = 0`

CALL STACK

`main()`

Code Trace: Linked Queue `enqueue`



front = null · rear = null · size = 0

`new Node(10)`: a new node with next = null, not yet in the list.

Step 2 / 16

```
1  public void enqueue(int data) {
▶      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

VARIABLES

`data = 10` `temp = node(10)`

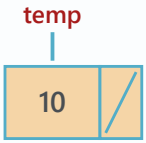
`front = null` `rear = null` `size = 0`

CALL STACK

`enqueue(10)`

`main()`

Code Trace: Linked Queue `enqueue`



front = null · rear = null · size = 0

`isEmpty()` → **true** (size 0).

Step 3 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
▶      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

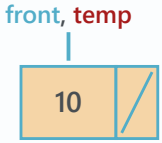
VARIABLES

`data = 10` `temp = node(10)` `front = null` `rear = null` `size = 0`

CALL STACK

- **enqueue(10)**
- main()

Code Trace: Linked Queue `enqueue`



rear = null · size = 0

First node: `front = temp`.

Step 4 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
▶          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

VARIABLES

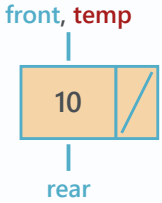
`data = 10` `temp = node(10)` `front = node(10)` `rear = null` `size = 0`

CALL STACK

`enqueue(10)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 0

`rear = temp` : rear is now the new node. The lone node is both front and rear.

Step 5 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
▶      rear = temp;
9      size++;
10 }
```

VARIABLES

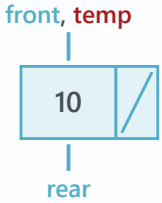
`data = 10` `temp = node(10)` `front = node(10)` `rear = node(10)` `size = 0`

CALL STACK

`enqueue(10)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 1

`size++` → 1.

Step 6 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
▶      size++;
10 }
```

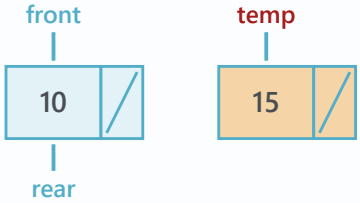
VARIABLES

`data = 10` `temp = node(10)`
`front = node(10)` `rear = node(10)`
`size = 1`

CALL STACK

`enqueue(10)`
`main()`

Code Trace: Linked Queue `enqueue`



size = 1

`new Node(15)`: a new node with next = null, not yet in the list.

Step 7 / 16

```
1  public void enqueue(int data) {
▶       Node temp = new Node(data);
3       if (isEmpty()) {
4           front = temp;
5       } else {
6           rear.next = temp;
7       }
8       rear = temp;
9       size++;
10 }
```

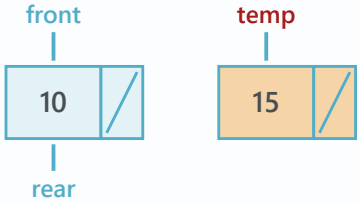
VARIABLES

`data = 15` `temp = node(15)` `front = node(10)` `rear = node(10)` `size = 1`

CALL STACK

- `enqueue(15)`
- `main()`

Code Trace: Linked Queue `enqueue`



size = 1

`isEmpty()` → **false**; take the else branch.

Step 8 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
▶      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

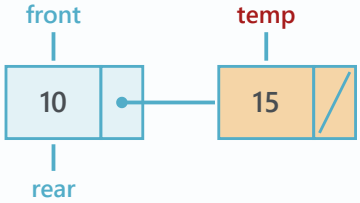
VARIABLES

`data = 15` `temp = node(15)` `front = node(10)` `rear = node(10)` `size = 1`

CALL STACK

- **enqueue(15)**
- main()

Code Trace: Linked Queue `enqueue`



size = 1

`rear.next = temp` : the old rear (10) links to the new node.

Step 9 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
▶          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

VARIABLES

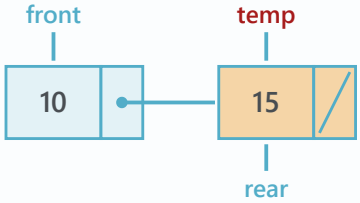
`data = 15` `temp = node(15)` `front = node(10)` `rear = node(10)` `size = 1`

CALL STACK

`enqueue(15)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 1

`rear = temp` : rear is now the new node.

Step 10 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
▶      rear = temp;
9      size++;
10 }
```

VARIABLES

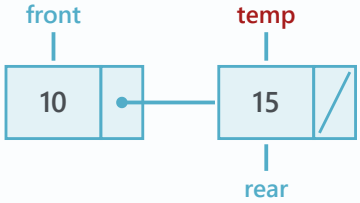
`data = 15` `temp = node(15)` `front = node(10)` `rear = node(15)` `size = 1`

CALL STACK

`enqueue(15)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 2

`size++` → 2.

Step 11 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
▶      size++;
10 }
```

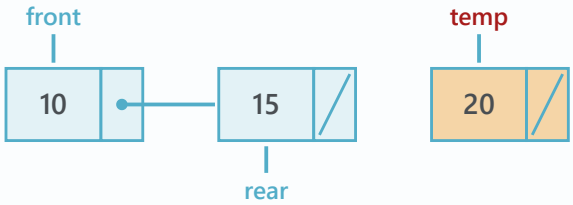
VARIABLES

`data = 15` `temp = node(15)`
`front = node(10)` `rear = node(15)`
`size = 2`

CALL STACK

`enqueue(15)`
`main()`

Code Trace: Linked Queue `enqueue`



size = 2

`new Node(20)`: a new node with next = null, not yet in the list.

Step 12 / 16

```
1  public void enqueue(int data) {
▶       Node temp = new Node(data);
3       if (isEmpty()) {
4           front = temp;
5       } else {
6           rear.next = temp;
7       }
8       rear = temp;
9       size++;
10 }
```

VARIABLES

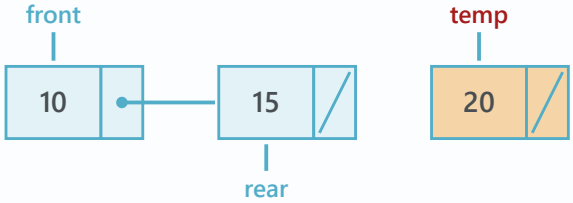
`data = 20` `temp = node(20)` `front = node(10)` `rear = node(15)` `size = 2`

CALL STACK

`enqueue(20)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 2

`isEmpty()` → **false**; take the else branch.

Step 13 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
▶      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

VARIABLES

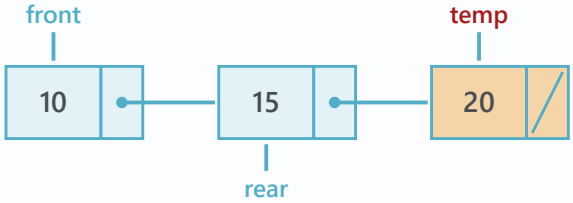
`data = 20` `temp = node(20)` `front = node(10)` `rear = node(15)` `size = 2`

CALL STACK

`enqueue(20)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 2

`rear.next = temp` : the old rear (15) links to the new node.

Step 14 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
▶          rear.next = temp;
7      }
8      rear = temp;
9      size++;
10 }
```

VARIABLES

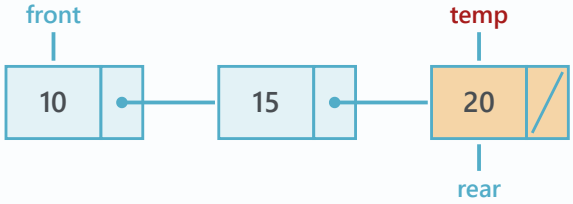
`data = 20` `temp = node(20)` `front = node(10)` `rear = node(15)` `size = 2`

CALL STACK

`enqueue(20)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 2

`rear = temp` : rear is now the new node.

Step 15 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
▶      rear = temp;
9      size++;
10 }
```

VARIABLES

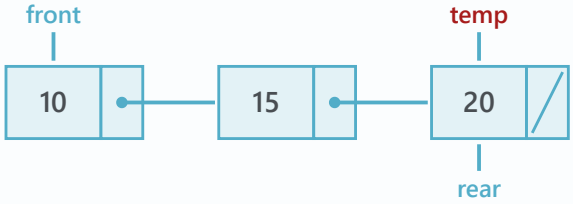
`data = 20` `temp = node(20)` `front = node(10)` `rear = node(20)` `size = 2`

CALL STACK

`enqueue(20)`

`main()`

Code Trace: Linked Queue `enqueue`



size = 3

`size++` → 3. The list was never traversed: each enqueue is **O(1)**.

Step 16 / 16

```
1  public void enqueue(int data) {
2      Node temp = new Node(data);
3      if (isEmpty()) {
4          front = temp;
5      } else {
6          rear.next = temp;
7      }
8      rear = temp;
▶      size++;
10 }
```

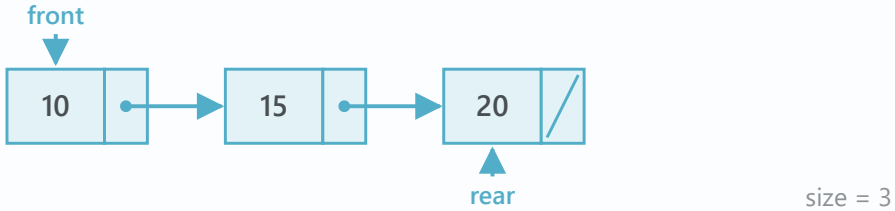
VARIABLES

`data = 20` `temp = node(20)`
`front = node(10)` `rear = node(20)`
`size = 3`

CALL STACK

`enqueue(20)`
`main()`

Code Trace: Linked Queue `dequeue`



Queue: 10, 15, 20. We call `dequeue()` three times.

Step 1 / 17

```
▶ public int dequeue() {
2     if (isEmpty())
3         throw new NoSuchElementException();
4     int result = front.data;
5     front = front.next;
6     if (front == null)
7         rear = null;
8     size--;
9     return result;
10 }
```

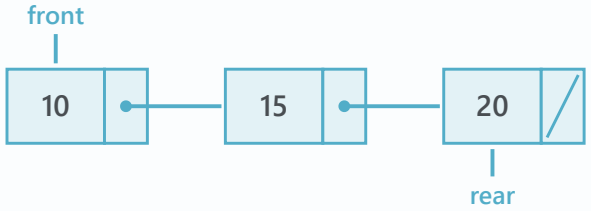
VARIABLES

`result = –` `front = node(10)` `rear = node(20)` `size = 3`

CALL STACK

`main()`

Code Trace: Linked Queue `dequeue`



size = 3

`isEmpty()` → false (size 3).

Step 2 / 17

```
1  public int dequeue() {
▶      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

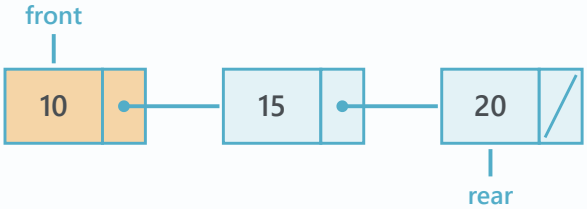
VARIABLES

`result = –` `front = node(10)` `rear = node(20)` `size = 3`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`



size = 3

result = front.data = 10.

Step 3 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
▶      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

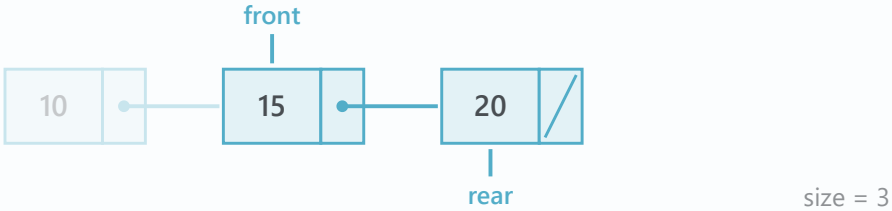
result = 10 | front = node(10) | rear = node(20) | size = 3

CALL STACK

dequeue()

main()

Code Trace: Linked Queue `dequeue`



`front = front.next` → node(15). Nothing refers to the old node now; the garbage collector reclaims it.

Step 4 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
▶      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

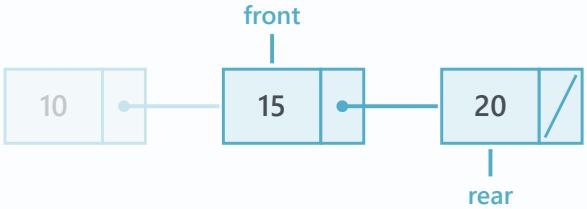
`result = 10` `front = node(15)` `rear = node(20)` `size = 3`

CALL STACK

`dequeue()`

`main()`

Code Trace: Linked Queue `dequeue`



size = 3

`front == null` → false; rear stays the same.

Step 5 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
▶      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

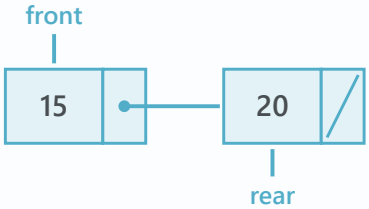
VARIABLES

`result = 10` `front = node(15)` `rear = node(20)` `size = 3`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`



size = 2

`size--` → 2; **return 10**.

Step 6 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
▶      size--;
▶      return result;
10 }
```

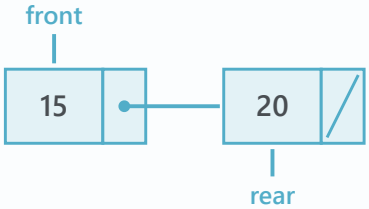
VARIABLES

`result = 10` `front = node(15)`
`rear = node(20)` `size = 2`

CALL STACK

dequeue()
main()

Code Trace: Linked Queue `dequeue`



size = 2

`isEmpty()` → false (size 2).

Step 7 / 17

```
1  public int dequeue() {
▶      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

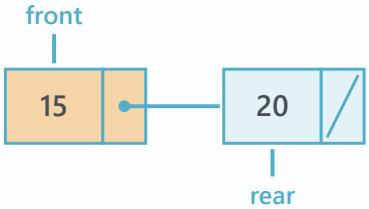
`result = –` `front = node(15)` `rear = node(20)` `size = 2`

CALL STACK

`dequeue()`

`main()`

Code Trace: Linked Queue `dequeue`



size = 2

result = front.data = 15.

Step 8 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
▶      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

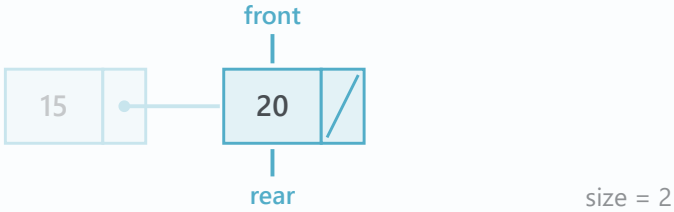
result = 15 | front = node(15) | rear = node(20) | size = 2

CALL STACK

dequeue()

main()

Code Trace: Linked Queue `dequeue`



`front = front.next` → node(20). Nothing refers to the old node now; the garbage collector reclaims it.

Step 9 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
▶      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

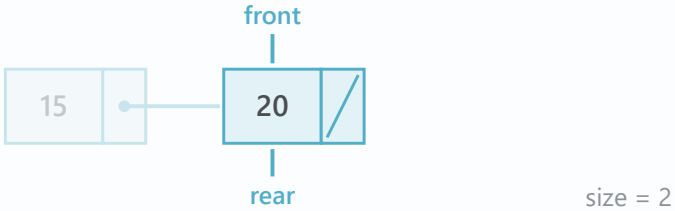
VARIABLES

`result = 15` `front = node(20)` `rear = node(20)` `size = 2`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`



`front == null` → false; rear stays the same.

Step 10 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
▶      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

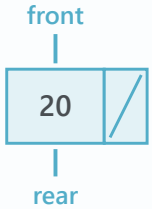
VARIABLES

`result = 15` `front = node(20)`
`rear = node(20)` `size = 2`

CALL STACK

dequeue()
main()

Code Trace: Linked Queue `dequeue`



size = 1

`size--` → 1; **return 15**.

Step 11 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
▶      size--;
▶      return result;
10 }
```

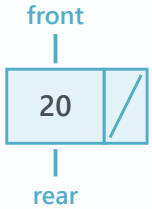
VARIABLES

`result = 15` `front = node(20)`
`rear = node(20)` `size = 1`

CALL STACK

dequeue()
main()

Code Trace: Linked Queue `dequeue`



size = 1

`isEmpty()` → false (size 1).

Step 12 / 17

```
1  public int dequeue() {
▶      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

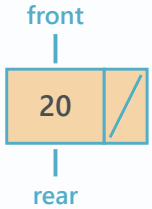
`result = –` `front = node(20)` `rear = node(20)` `size = 1`

CALL STACK

`dequeue()`

`main()`

Code Trace: Linked Queue `dequeue`



size = 1

result = front.data = 20.

Step 13 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
▶      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

result = 20 | front = node(20) | rear = node(20) | size = 1

CALL STACK

dequeue()

main()

Code Trace: Linked Queue `dequeue`



front = null · size = 1

`front = front.next` → **null**. But `rear` still points to the removed node!

Step 14 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
▶      front = front.next;
6      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

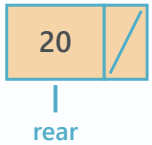
VARIABLES

`result = 20` `front = null` `rear = node(20)` `size = 1`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`



front = null · size = 1

`front == null` → true: the queue is now empty.

Step 15 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
▶      if (front == null)
7          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

`result = 20` `front = null`
`rear = node(20)` `size = 1`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`



front = null · rear = null · size = 1

`rear = null` : without this line, rear would go on pointing to the removed node in an empty queue (an inconsistent state).

Step 16 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
▶          rear = null;
8      size--;
9      return result;
10 }
```

VARIABLES

`result = 20` `front = null`
`rear = null` `size = 1`

CALL STACK

- **dequeue()**
- main()

Code Trace: Linked Queue `dequeue`

front = null · rear = null · size = 0

`size--` → 0; **return 20**. Items left in the order 10, 15, 20 (FIFO), each in **O(1)**.

Step 17 / 17

```
1  public int dequeue() {
2      if (isEmpty())
3          throw new NoSuchElementException();
4      int result = front.data;
5      front = front.next;
6      if (front == null)
7          rear = null;
▶      size--;
▶      return result;
10 }
```

VARIABLES

`result = 20` `front = null` `rear = null` `size = 0`

CALL STACK

- **dequeue()**
- main()

Circular Array or Linked List?

Circular array

+ Contiguous in memory: cache-friendly and fast.
+ No extra reference per item.
+ Fixed capacity: when full, either overflow or copy into a larger array (O(n), amortized O(1)).

Linked list

+ No capacity limit; grows as needed.
+ Every insertion and removal is strictly O(1).
+ Each node costs an extra reference and an object allocation.

Operation	Circular array	Linked list
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
first / size / isEmpty	O(1)	O(1)
Memory	O(N) (the capacity)	O(n) (the number of items)

Kinds of Queues

Kind	Insert	Remove	Note
Simple queue	rear	front	FIFO
Input-restricted	rear only	front and rear	restricted deque
Output-restricted	front and rear	front only	restricted deque
Circular	rear	front	array slots are reused
Double-ended (deque)	front and rear	front and rear	not bound to FIFO
Priority	anywhere	highest priority	ordered by priority

Deque, short for *double-ended queue*, is pronounced “deck.” Don't confuse it with `dequeue`, the remove-from-queue operation.

KINDS OF QUEUES

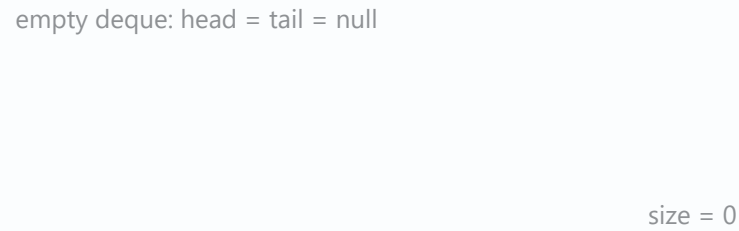
The Double-Ended Queue (Deque)



- A double-ended queue supports insertion and removal at both the front and the rear.
- Not bound to FIFO: used only at the rear, a deque is a **stack** (LIFO); inserting at the rear and removing at the front makes it a **queue**.

Operation	Java `ArrayDeque`	Time
addFirst(e)	`offerFirst(e)`	O(1)
addLast(e)	`offerLast(e)`	O(1)
removeFirst()	`pollFirst()`	O(1)
removeLast()	`pollLast()`	O(1)
first() / last()	`peekFirst()` / `peekLast()`	O(1)

Deque Operations, Step by Step

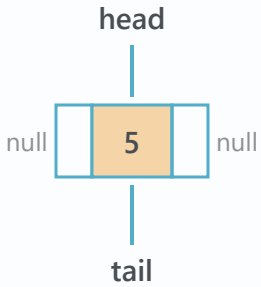


An empty deque. We implement it with a doubly linked list: every node has `prev` and `next`.

Step 1 / 7

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step



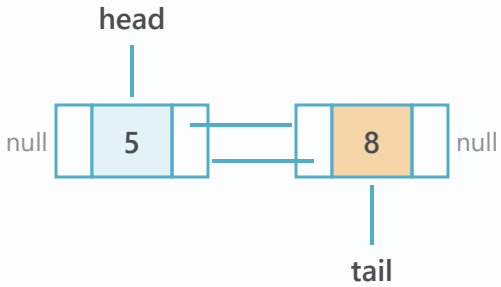
addLast(5) size = 1

First node: `head = tail = node`.

Step 2 / 7

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step



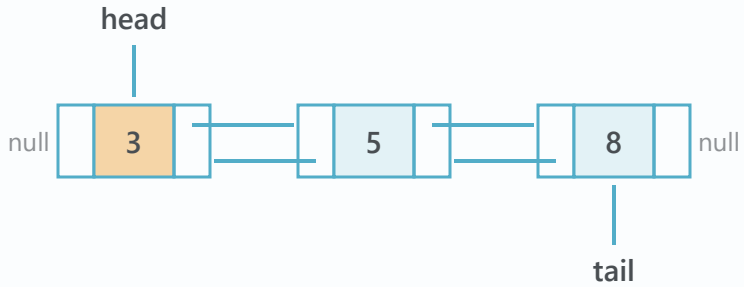
addLast(8) size = 2

Insert at the tail: `node.prev = tail; tail.next = node; tail = node`.

Step 3 / 7

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step



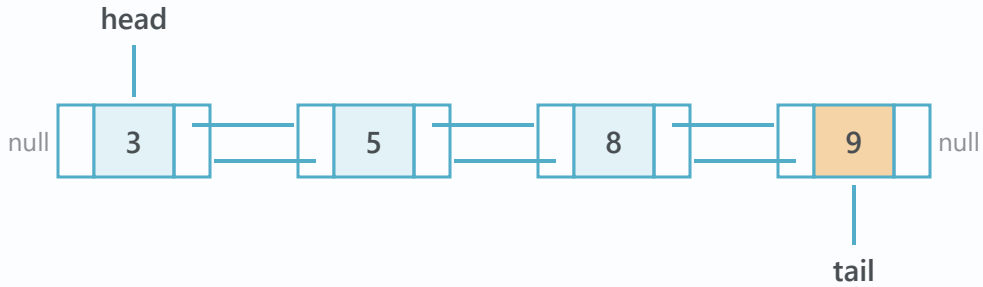
addFirst(3) size = 3

Insert at the head: `node.next = head; head.prev = node; head = node`.

Step 4 / 7

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step



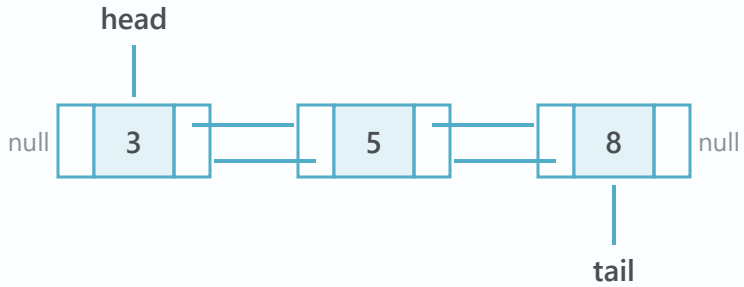
addLast(9)

size = 4

Both ends are held by references, so every insertion is O(1).

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step

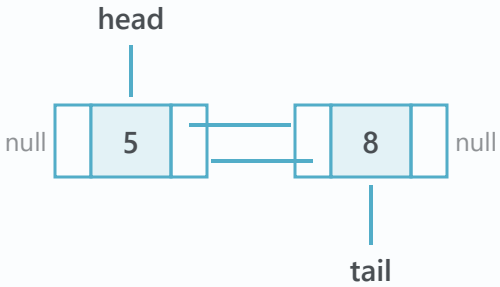


removeLast() returned 9 · size = 3

Remove from the tail: `tail = tail.prev; tail.next = null`. The prev reference makes this O(1).

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque Operations, Step by Step



removeFirst() returned 3 · size = 2

Remove from the head: `head = head.next; head.prev = null`.

Step 7 / 7

In a singly linked list, removing the tail is O(n), since we must find the node before it. The `prev` reference makes it O(1).

Deque: Inserting at Either End

```java
public void addFirst(E data) {
    Node<E> node = new Node<>(data);
    if (isEmpty()) {
        head = tail = node;
    } else {
        node.next = head;
        head.prev = node;
        head = node;
    }
    size++;
}
```

```java
public void addLast(E data) {
    Node<E> node = new Node<>(data);
    if (isEmpty()) {
        head = tail = node;
    } else {
        node.prev = tail;
        tail.next = node;
        tail = node;
    }
    size++;
}
```

```java
class Node<E> {
    E data;
    Node<E> prev, next;
    Node(E data) { this.data = data; }
}
```

The two methods mirror each other: `head` ↔ `tail`, `prev` ↔ `next`.

Code Trace: Deque `addFirst()`

head = null · tail = null · size = 0

Empty deque. First `addFirst(5)`, then `addFirst(3)`: 3 will end up in front.

Step 1 / 11

```
▶ public void addFirst(E data) {
2     Node<E> node = new Node<>(data);
3     if (isEmpty()) {
4         head = tail = node;
5     } else {
6         node.next = head;
7         head.prev = node;
8         head = node;
9     }
10    size++;
11 }
```

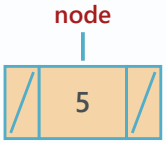
VARIABLES

data = – node = – head = **null** tail = **null** size = **0**

CALL STACK

main()

Code Trace: Deque `addFirst()`



head = null · tail = null · size = 0

`new Node<>(5)` : prev = next = null.

Step 2 / 11

```
1  public void addFirst(E data) {
▶      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

VARIABLES

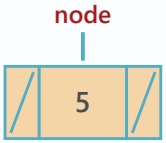
`data = 5` `node = node(5)` `head = null` `tail = null` `size = 0`

CALL STACK

`addFirst(5)`

`main()`

Code Trace: Deque `addFirst()`



head = null · tail = null · size = 0

`isEmpty()` → **true**.

Step 3 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
▶      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

VARIABLES

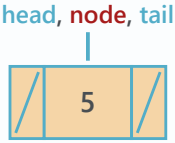
`data = 5` `node = node(5)` `head = null` `tail = null` `size = 0`

CALL STACK

`addFirst(5)`

`main()`

Code Trace: Deque `addFirst()`



size = 0

`head = tail = node` : a single node forms both ends.

Step 4 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
▶          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

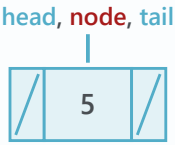
VARIABLES

`data = 5` `node = node(5)`
`head = node(5)` `tail = node(5)`
`size = 0`

CALL STACK

`addFirst(5)`
`main()`

Code Trace: Deque `addFirst()`



size = 1

`size++` → 1.

Step 5 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
▶      size++;
11 }
```

VARIABLES

`data = 5` `node = node(5)`
`head = node(5)` `tail = node(5)`
`size = 1`

CALL STACK

`addFirst(5)`
`main()`

Code Trace: Deque `addFirst()`



size = 1

`new Node<>(3)` : prev = next = null.

Step 6 / 11

```
1  public void addFirst(E data) {
▶      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

VARIABLES

`data = 3` `node = node(3)` `head = node(5)` `tail = node(5)` `size = 1`

CALL STACK

- `addFirst(3)`
- `main()`

Code Trace: Deque `addFirst()`



size = 1

`isEmpty()` → **false**.

Step 7 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
▶      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

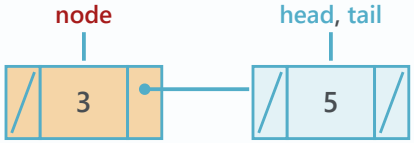
VARIABLES

`data = 3` `node = node(3)`
`head = node(5)` `tail = node(5)`
`size = 1`

CALL STACK

`addFirst(3)`
`main()`

Code Trace: Deque `addFirst()`



size = 1

`node.next = head` : the new node points to the old head (5).

Step 8 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
▶          node.next = head;
7          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

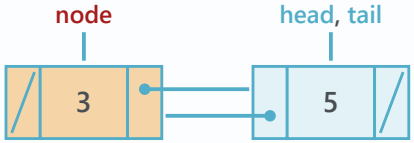
VARIABLES

`data = 3` `node = node(3)` `head = node(5)` `tail = node(5)` `size = 1`

CALL STACK

- **addFirst(3)**
- main()

Code Trace: Deque `addFirst()`



size = 1

`head.prev = node` : the back link; the list can now be walked in both directions.

Step 9 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
▶          head.prev = node;
8          head = node;
9      }
10     size++;
11 }
```

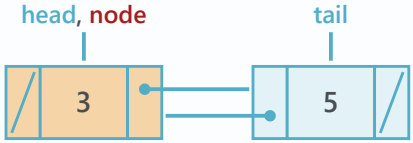
VARIABLES

`data = 3` `node = node(3)` `head = node(5)` `tail = node(5)` `size = 1`

CALL STACK

- **addFirst(3)**
- main()

Code Trace: Deque `addFirst()`



size = 1

`head = node` : the head reference moves to the new node.

Step 10 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
▶          head = node;
9      }
10     size++;
11 }
```

VARIABLES

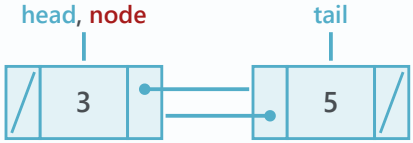
`data = 3` `node = node(3)` `head = node(3)` `tail = node(5)` `size = 1`

CALL STACK

`addFirst(3)`

`main()`

Code Trace: Deque `addFirst()`



size = 2

`size++` → 2. No traversal: **O(1)**. `addLast` is its mirror image.

Step 11 / 11

```
1  public void addFirst(E data) {
2      Node<E> node = new Node<>(data);
3      if (isEmpty()) {
4          head = tail = node;
5      } else {
6          node.next = head;
7          head.prev = node;
8          head = node;
9      }
▶      size++;
11 }
```

VARIABLES

`data = 3` `node = node(3)`
`head = node(3)` `tail = node(5)`
`size = 2`

CALL STACK

`addFirst(3)`
`main()`

KINDS OF QUEUES

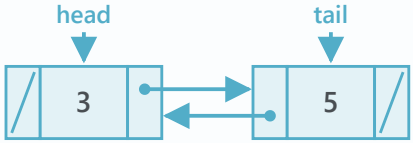
Deque: Removing from Either End

```
public E removeFirst() {
    if (isEmpty()) return null;
    E data = head.data;
    head = head.next;
    if (head == null)
        tail = null;        // now empty
    else
        head.prev = null;
    size--;
    return data;
}
```

```
public E removeLast() {
    if (isEmpty()) return null;
    E data = tail.data;
    tail = tail.prev;
    if (tail == null)
        head = null;        // now empty
    else
        tail.next = null;
    size--;
    return data;
}
```

When the last item leaves, **both ends** must become null. Otherwise `tail` still points to the removed node once `head` is null: the node cannot be collected, and `last()` returns a wrong value. An empty deque returns `null`, as in Goodrich.

Code Trace: Deque `removeLast()`



size = 2

Deque: 3 ⇄ 5. Two calls to `removeLast()`: 5 comes out first, then 3, and the deque is left empty.

Step 1 / 13

```
▶ public E removeLast() {
2     if (isEmpty()) return null;
3     E data = tail.data;
4     tail = tail.prev;
5     if (tail == null)
6         head = null;        // now empty
7     else
8         tail.next = null;
9     size--;
10    return data;
11 }
```

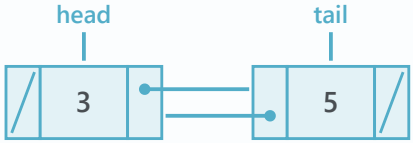
VARIABLES

`data = –` `head = node(3)` `tail = node(5)` `size = 2`

CALL STACK

`main()`

Code Trace: Deque `removeLast()`



size = 2

`isEmpty()` → false (size 2).

Step 2 / 13

```
1  public E removeLast() {
▶       if (isEmpty()) return null;
3       E data = tail.data;
4       tail = tail.prev;
5       if (tail == null)
6           head = null;        // now empty
7       else
8           tail.next = null;
9       size--;
10      return data;
11  }
```

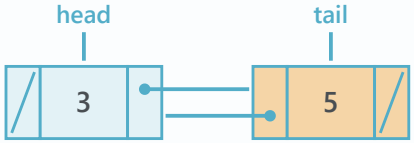
VARIABLES

`data = –` `head = node(3)` `tail = node(5)` `size = 2`

CALL STACK

- **`removeLast()`**
- `main()`

Code Trace: Deque `removeLast()`



size = 2

data = tail.data = 5.

Step 3 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
▶      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

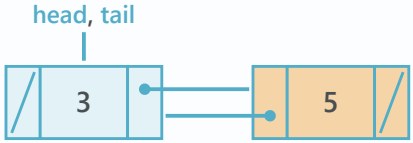
`data = 5` `head = node(3)` `tail = node(5)` `size = 2`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



size = 2

`tail = tail.prev` → node(3). Thanks to `prev`, there is no need to search for the second-to-last node.

Step 4 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
▶      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

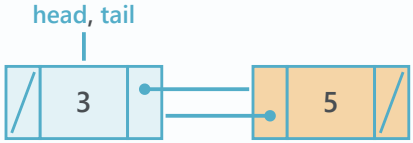
`data = 5` `head = node(3)` `tail = node(3)` `size = 2`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



size = 2

`tail == null` → false: the deque is not empty yet.

Step 5 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
▶      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

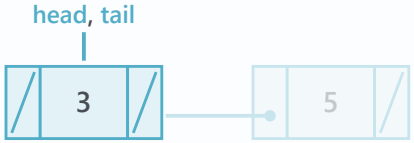
VARIABLES

`data = 5` `head = node(3)` `tail = node(3)` `size = 2`

CALL STACK

- **`removeLast()`**
- `main()`

Code Trace: Deque `removeLast()`



size = 2

`tail.next = null` : the link to node 5 is cut, and the node leaves the list.

Step 6 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
▶          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

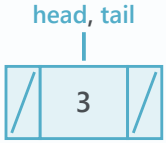
`data = 5` `head = node(3)` `tail = node(3)` `size = 2`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



size = 1

`size--` → 1; **return 5**.

Step 7 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
▶      size--;
▶      return data;
11 }
```

VARIABLES

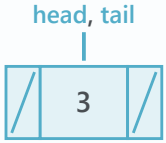
`data = 5` `head = node(3)` `tail = node(3)` `size = 1`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



size = 1

`isEmpty()` → false (size 1).

Step 8 / 13

```
1  public E removeLast() {
▶      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

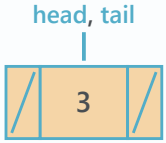
`data = –` `head = node(3)` `tail = node(3)` `size = 1`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



size = 1

data = tail.data = 3.

Step 9 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
▶      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

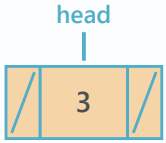
`data = 3` `head = node(3)` `tail = node(3)` `size = 1`

CALL STACK

`removeLast()`

`main()`

Code Trace: Deque `removeLast()`



tail = null · size = 1

`tail = tail.prev` → **null** (there was only one node). But head still points to it!


```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
▶      tail = tail.prev;
5      if (tail == null)
6          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

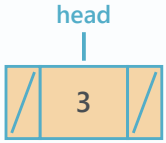
VARIABLES

`data = 3` `head = node(3)` `tail = null` `size = 1`

CALL STACK

- `removeLast()`
- `main()`

Code Trace: Deque `removeLast()`



tail = null · size = 1

`tail == null` → true: the deque is now empty.

Step 11 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
▶      if (tail == null)
6          head = null;       // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

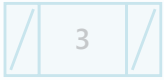
VARIABLES

`data = 3` `head = node(3)` `tail = null` `size = 1`

CALL STACK

- **`removeLast()`**
- `main()`

Code Trace: Deque `removeLast()`



head = null · tail = null · size = 1

`head = null` : both ends must be null; otherwise head would point to the removed node.

Step 12 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
▶          head = null;        // now empty
7      else
8          tail.next = null;
9      size--;
10     return data;
11 }
```

VARIABLES

`data = 3` `head = null` `tail = null` `size = 1`

CALL STACK

- `removeLast()`
- `main()`

Code Trace: Deque `removeLast()`

head = null · tail = null · size = 0

`size--` → 0; **return 3**. **O(1)** in both cases.

Step 13 / 13

```
1  public E removeLast() {
2      if (isEmpty()) return null;
3      E data = tail.data;
4      tail = tail.prev;
5      if (tail == null)
6          head = null;       // now empty
7      else
8          tail.next = null;
▶      size--;
▶      return data;
11 }
```

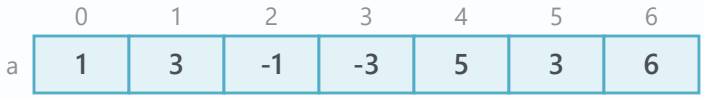
VARIABLES

`data = 3` `head = null` `tail = null` `size = 0`

CALL STACK

- **removeLast()**
- main()

A Deque Application: Sliding Window Maximum



d: index:value

d (front) empty

maxes = []

k = 3. d holds indices whose values **decrease** from front to rear, so the maximum is always at the front.

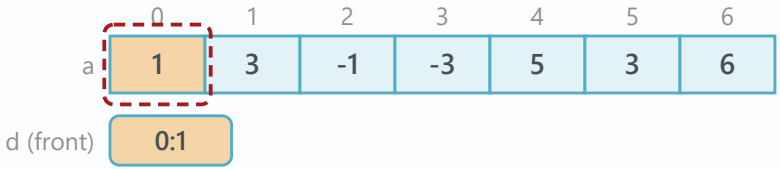
Step 1 / 9

```
1  int[] windowMax(int[] a, int k) {
▶    int[] maxes = new int[a.length - k + 1];
▶    Deque<Integer> d = new ArrayDeque<>(); // indices
4    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
9      d.offerLast(i);
10     if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

k = 3 | i = – | d = [] | maxes = []

A Deque Application: Sliding Window Maximum



a[0] = 1: 0 goes in at the rear.

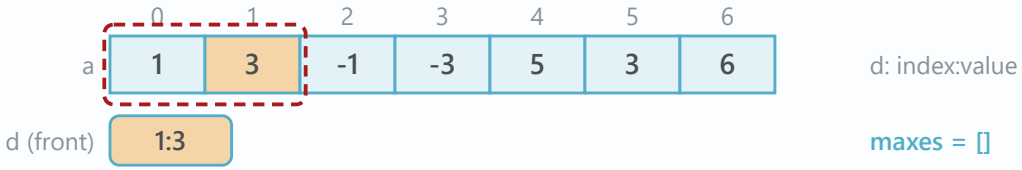
Step 2 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
▶      d.offerLast(i);
10     if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

k = 3 | i = 0 | d = [0] | maxes = []

A Deque Application: Sliding Window Maximum



a[1] = 3: drop 0:1 from the rear. 1 goes in at the rear.

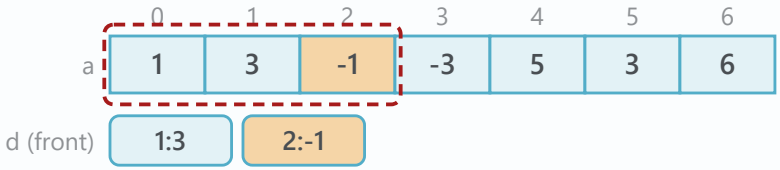
Step 3 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // not a candidate
▶      d.offerLast(i);
10     if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

k = 3 | i = 1 | d = [1] | maxes = []

A Deque Application: Sliding Window Maximum



a[2] = -1: 2 goes in at the rear. Max = **3**.

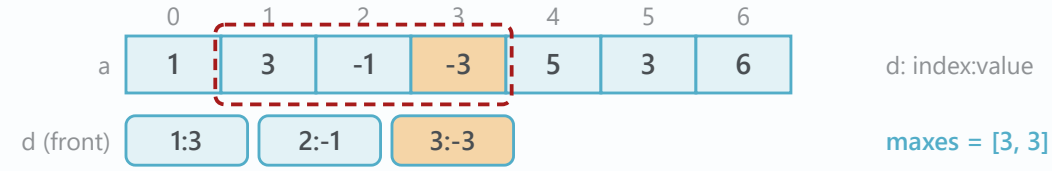
Step 4 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
▶      d.offerLast(i);
▶      if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

`k = 3` `i = 2` `d = [1, 2]` `maxes = [3]`

A Deque Application: Sliding Window Maximum



a[3] = -3: 3 goes in at the rear. Max = **3**.

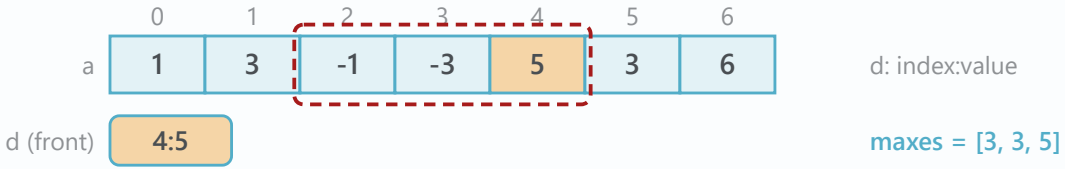
Step 5 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
▶      d.offerLast(i);
▶      if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

`k = 3` `i = 3` `d = [1, 2, 3]` `maxes = [3, 3]`

A Deque Application: Sliding Window Maximum



a[4] = 5: drop 1 from the front; drop 3:-3 and then 2:-1 from the rear. 4 goes in at the rear. Max = **5**.

Step 6 / 9

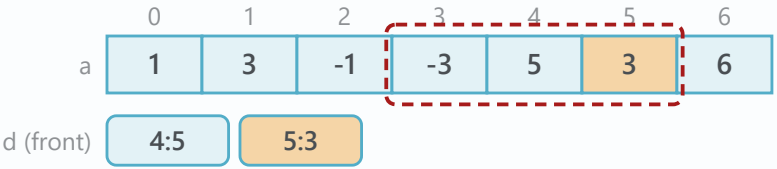
```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
▶        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // not a candidate
▶      d.offerLast(i);
▶      if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

`k = 3` `i = 4` `d = [4]`

`maxes = [3, 3, 5]`

A Deque Application: Sliding Window Maximum



d: index:value

maxes = [3, 3, 5, 5]

a[5] = 3: 5 goes in at the rear. Max = **5**.

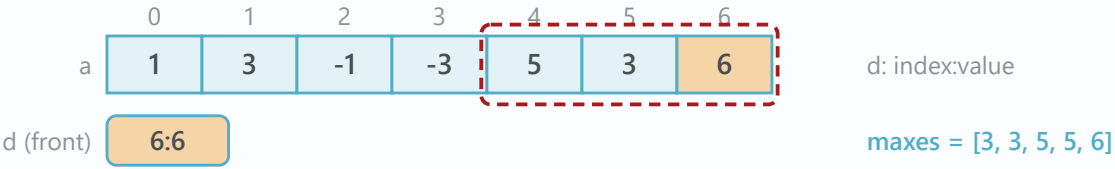
Step 7 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
▶      d.offerLast(i);
▶      if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

k = 3 | i = 5 | d = [4, 5] | maxes = [3, 3, 5, 5]

A Deque Application: Sliding Window Maximum



a[6] = 6: drop 5:3 and then 4:5 from the rear. 6 goes in at the rear. Max = **6**.

Step 8 / 9

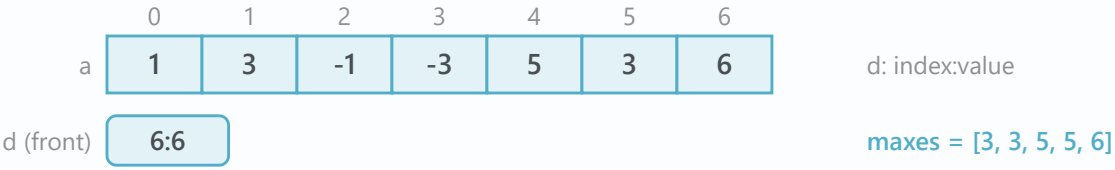
```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
▶    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
▶        d.pollLast();         // not a candidate
▶      d.offerLast(i);
▶      if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
12   return maxes;
13 }
```

VARIABLES

`k = 3` `i = 6` `d = [6]`

`maxes = [3, 3, 5, 5, 6]`

A Deque Application: Sliding Window Maximum



Each index enters once and leaves at most once → **O(n)** (brute force takes O(n·k)).

Step 9 / 9

```
1  int[] windowMax(int[] a, int k) {
2    int[] maxes = new int[a.length - k + 1];
3    Deque<Integer> d = new ArrayDeque<>(); // indices
4    for (int i = 0; i < a.length; i++) {
5      if (!d.isEmpty() && d.peekFirst() <= i - k)
6        d.pollFirst();        // out of window
7      while (!d.isEmpty() && a[d.peekLast()] <= a[i])
8        d.pollLast();         // not a candidate
9      d.offerLast(i);
10     if (i >= k - 1) maxes[i - k + 1] = a[d.peekFirst()];
11   }
▶    return maxes;
13 }
```

VARIABLES

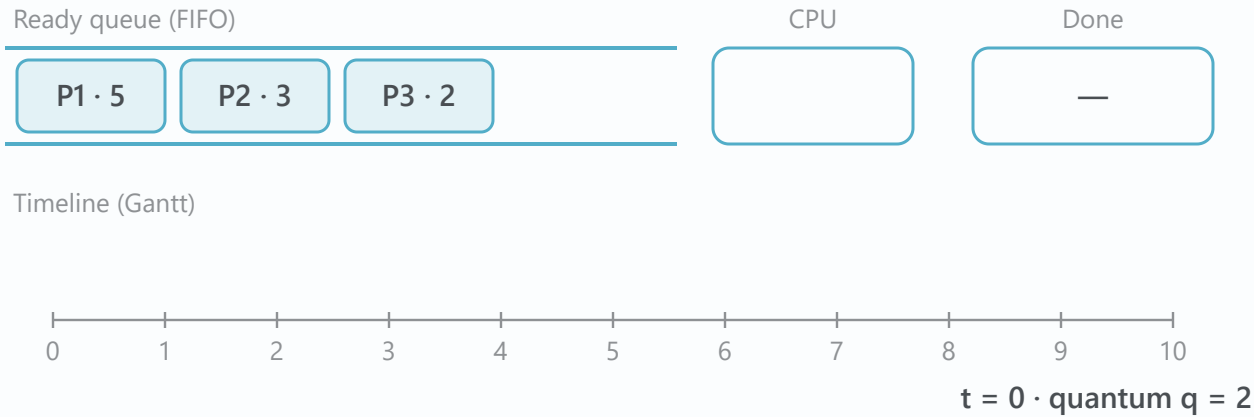
`k = 3` `i = –` `d = [6]`

`maxes = [3, 3, 5, 5, 6]`

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



Three processes in the ready queue, with 5, 3, and 2 units of work left. Time slice (quantum) q = 2.

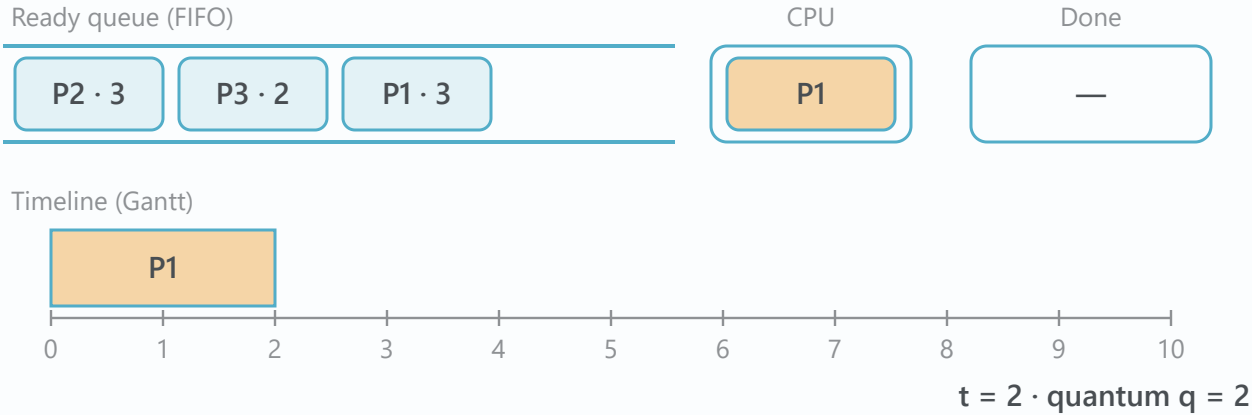
Step 1 / 7

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



P1 leaves the front (dequeue), runs for 2 units, and rejoins at the rear with 3 left (enqueue).

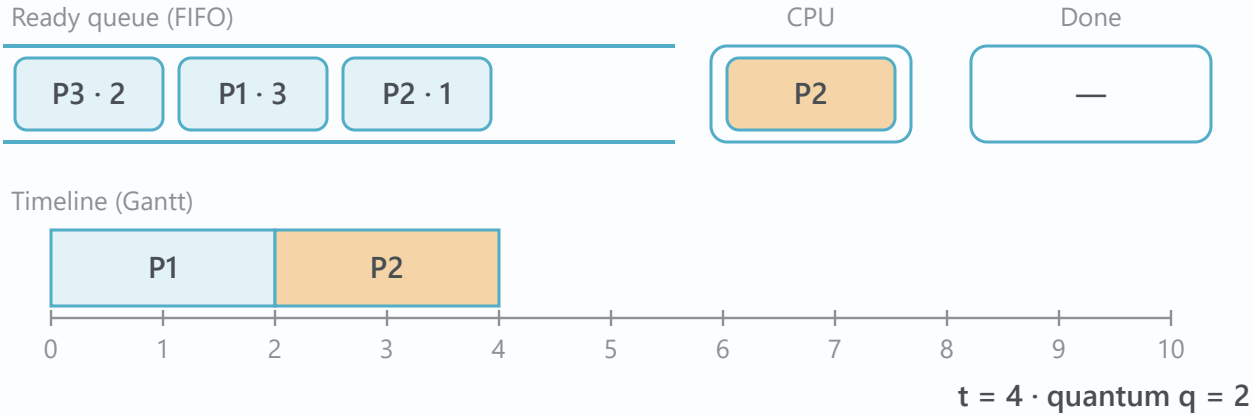
Step 2 / 7

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



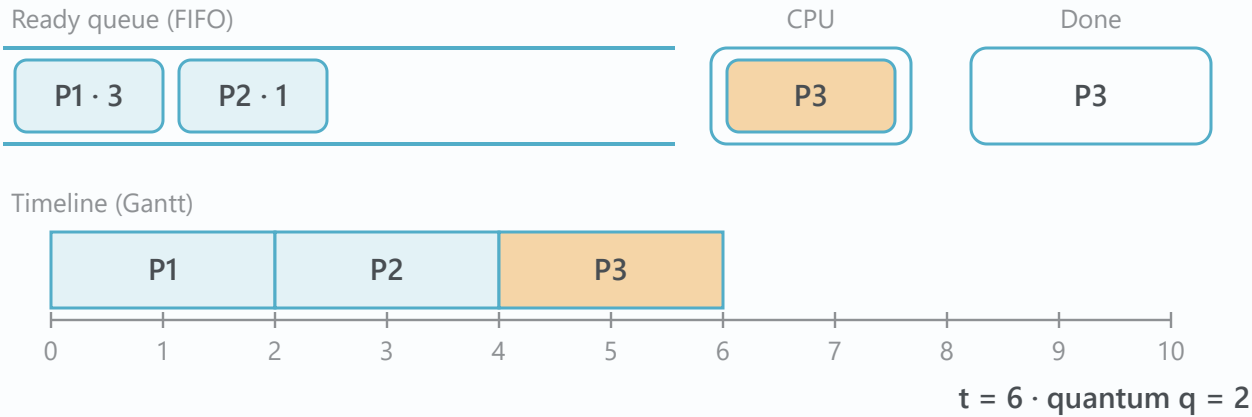
P2 runs for 2 units and goes back to the rear with 1 left.

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



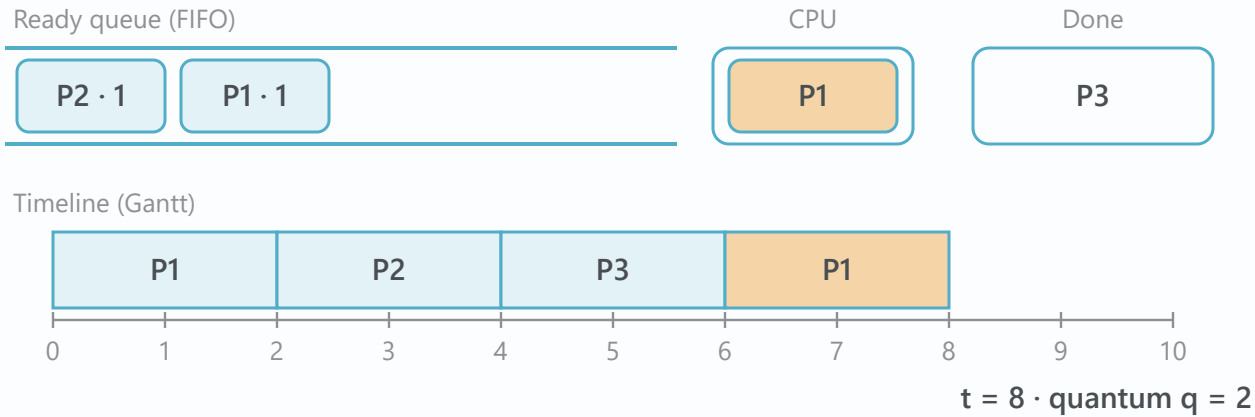
P3's 2 units fit in one slice → it finishes at t = 6 and does not rejoin.

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



P1 runs for another 2 units; 1 is left.

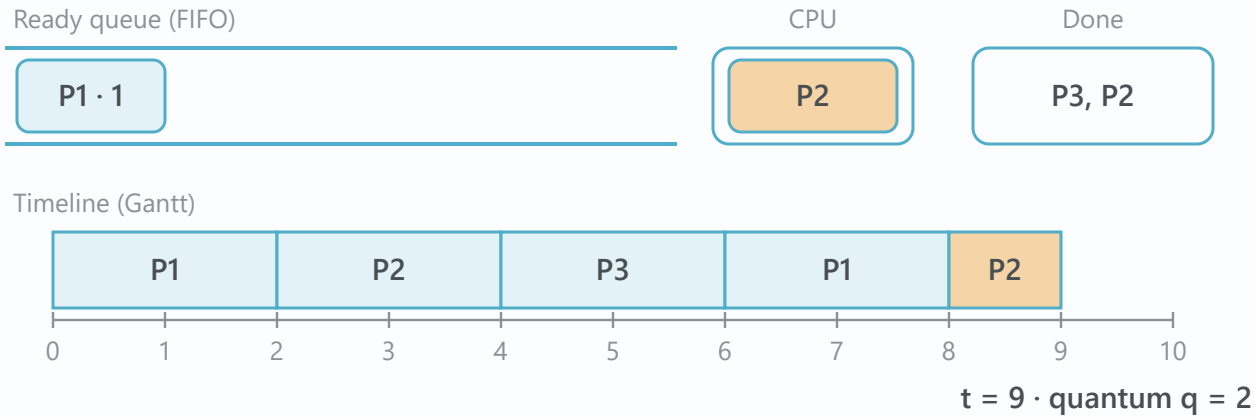
Step 5 / 7

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.



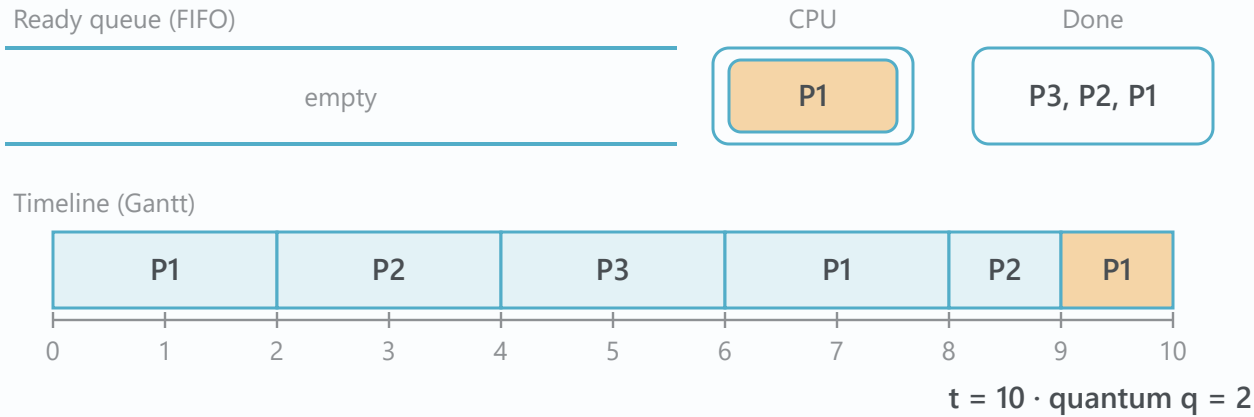
P2 finishes its last unit (t = 9) without using the whole slice.

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

APPLICATIONS

Application: A Round-Robin Scheduler

The operating system gives the CPU to processes in turn: the process at the front runs for at most **q** time units and, if unfinished, rejoins at the rear.

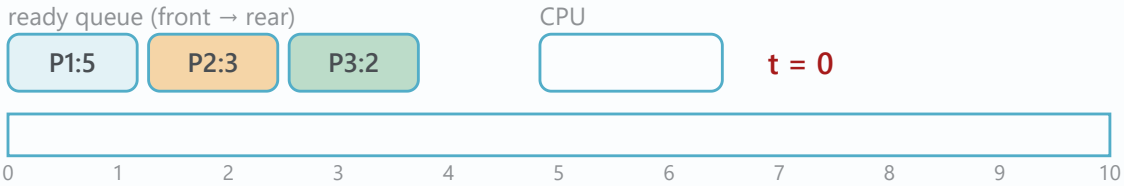


All done. Completion times: P3 = 6, P2 = 9, P1 = 10.

Step 7 / 7

Simulator: Round-Robin Scheduler (adjustable quantum and processes).

Code Trace: `roundRobin(q, 2)`



`roundRobin(q, 2)` : three processes with 5, 3, and 2 units left (Process: name, remaining). **t = 0**.

Step 1 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
▶    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

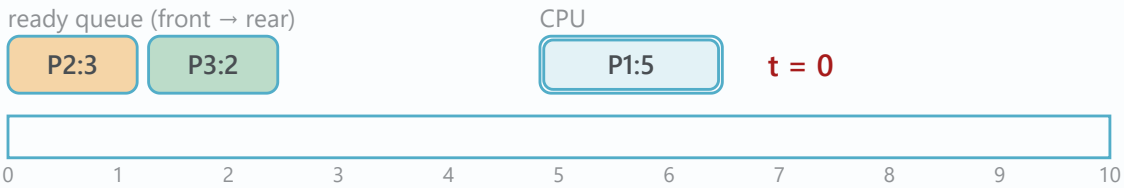
`quantum = 2` `t = 0`

`q = [P1:5, P2:3, P3:2]` `p = –`

`run = –`

OUTPUT

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P1 (5 left).

Step 2 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

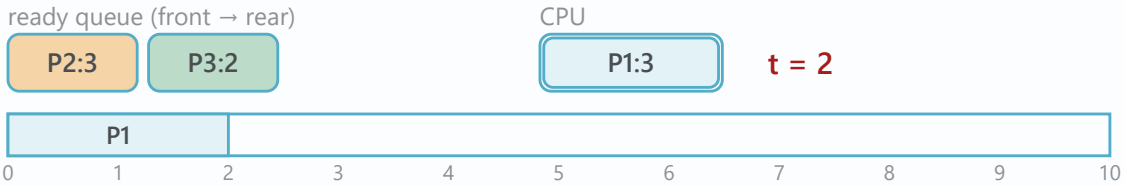
`quantum = 2` `t = 0`

`q = [P2:3, P3:2]` `p = P1:5`

`run = –`

OUTPUT

Code Trace: `roundRobin(q, 2)`



run = min(2, 5) = 2; P1 runs for 2 units → t = 2, 3 left.

Step 3 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

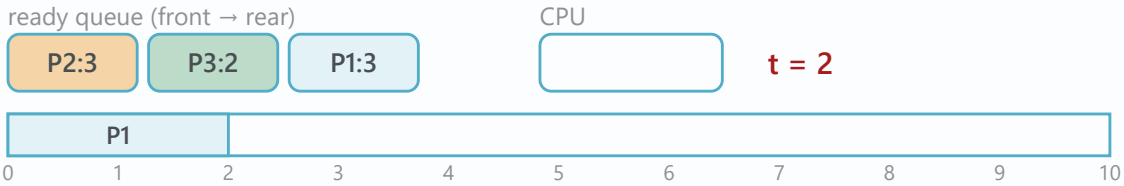
`quantum = 2` `t = 2`

`q = [P2:3, P3:2]` `p = P1:3`

`run = 2`

OUTPUT

Code Trace: `roundRobin(q, 2)`



remaining 3 > 0 → `q.offer(p)` : P1 rejoins at the **rear**.

Step 4 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
▶      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

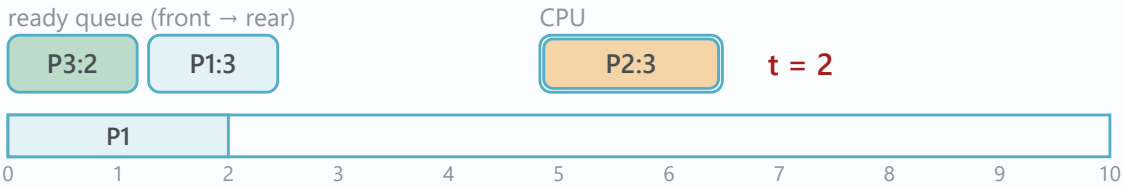
`quantum = 2` `t = 2`

`q = [P2:3, P3:2, P1:3]` `p = P1:3`

`run = 2`

OUTPUT

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P2 (3 left).

Step 5 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

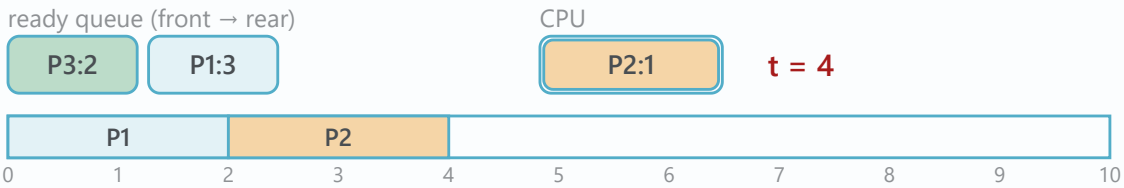
`quantum = 2` `t = 2`

`q = [P3:2, P1:3]` `p = P2:3`

`run = –`

OUTPUT

Code Trace: `roundRobin(q, 2)`



run = min(2, 3) = 2; P2 runs for 2 units → t = 4, 1 left.

Step 6 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

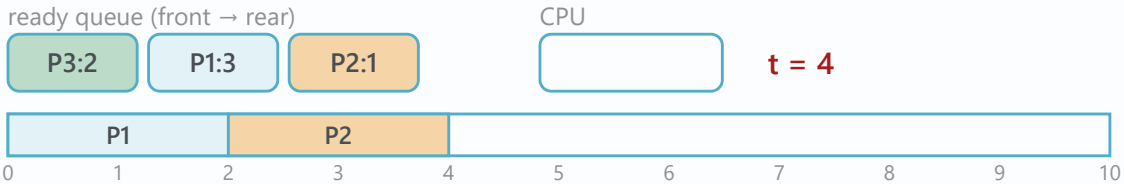
`quantum = 2` `t = 4`

`q = [P3:2, P1:3]` `p = P2:1`

`run = 2`

OUTPUT

Code Trace: `roundRobin(q, 2)`



remaining 1 > 0 → `q.offer(p)` : P2 rejoins at the **rear**.

Step 7 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
▶      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

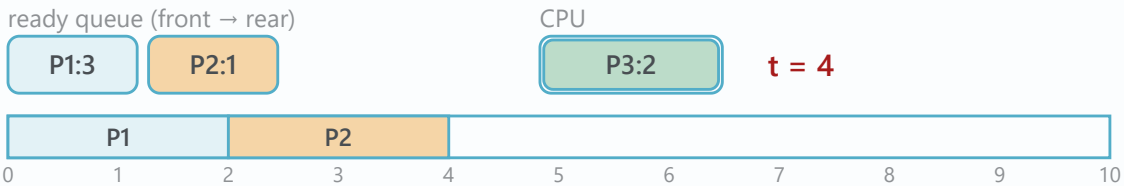
`quantum = 2` `t = 4`

`q = [P3:2, P1:3, P2:1]` `p = P2:1`

`run = 2`

OUTPUT

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P3 (2 left).

Step 8 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

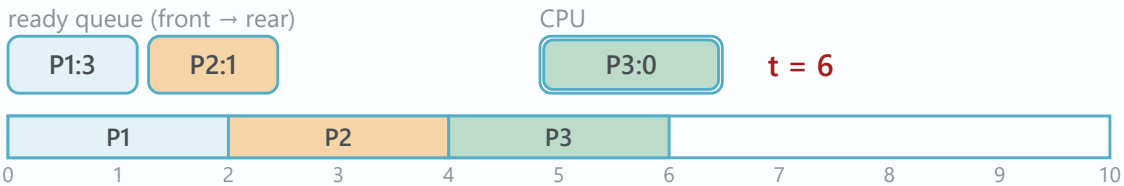
`quantum = 2` `t = 4`

`q = [P1:3, P2:1]` `p = P3:2`

`run = –`

OUTPUT

Code Trace: `roundRobin(q, 2)`



run = min(2, 2) = 2; P3 runs for 2 units → t = 6, 0 left.

Step 9 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

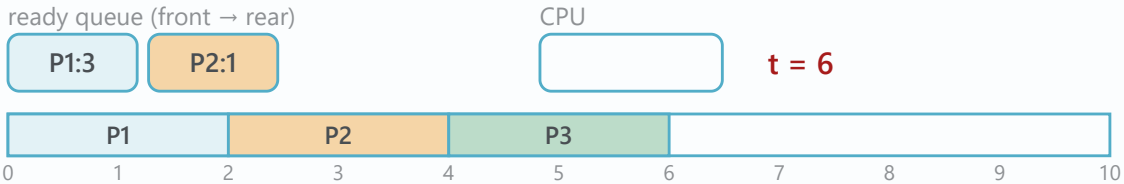
`quantum = 2` `t = 6`

`q = [P1:3, P2:1]` `p = P3:0`

`run = 2`

OUTPUT

Code Trace: `roundRobin(q, 2)`



remaining 0 → P3 is **done** (t = 6); it does not rejoin.

Step 10 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
▶      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

`quantum = 2` `t = 6`

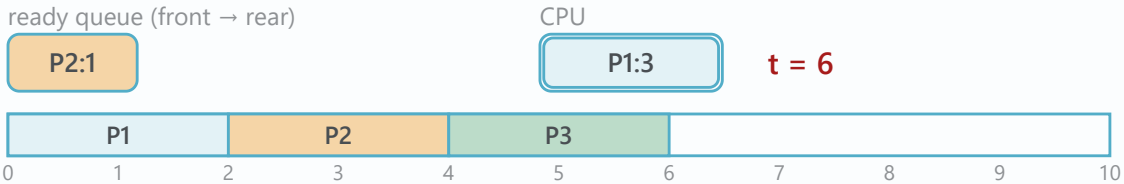
`q = [P1:3, P2:1]` `p = P3:0`

`run = 2`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P1 (3 left).

Step 11 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

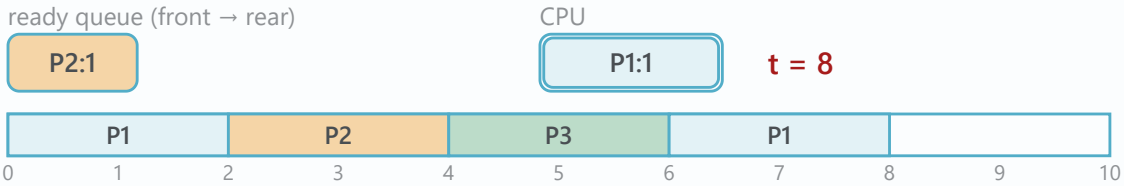
VARIABLES

`quantum = 2` `t = 6` `q = [P2:1]` `p = P1:3` `run = –`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



run = min(2, 3) = 2; P1 runs for 2 units → t = 8, 1 left.

Step 12 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

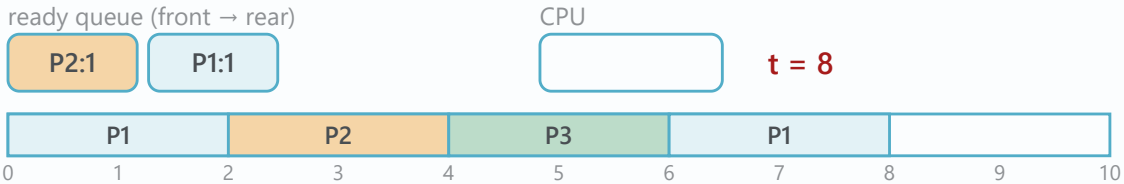
VARIABLES

`quantum = 2` `t = 8` `q = [P2:1]` `p = P1:1` `run = 2`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



remaining 1 > 0 → `q.offer(p)` : P1 rejoins at the **rear**.

Step 13 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
▶      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

`quantum = 2` `t = 8`

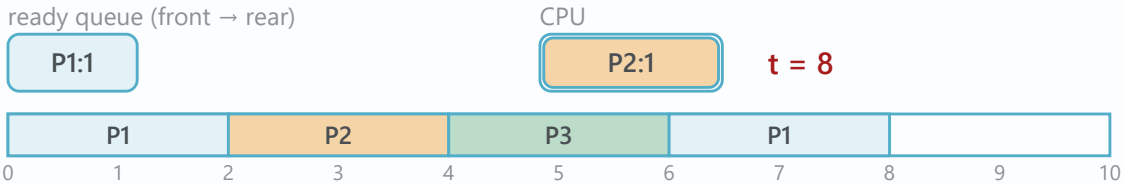
`q = [P2:1, P1:1]` `p = P1:1`

`run = 2`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P2 (1 left).

Step 14 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

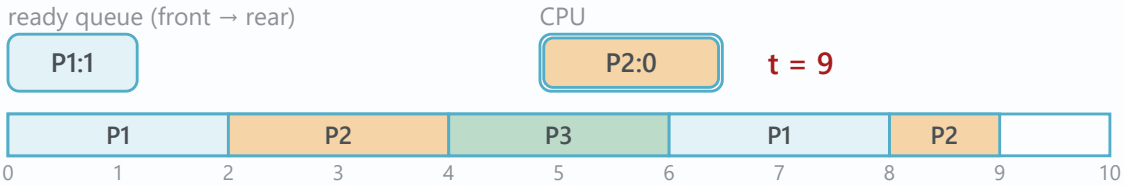
VARIABLES

`quantum = 2` `t = 8` `q = [P1:1]` `p = P2:1` `run = –`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



run = min(2, 1) = 1; P2 runs for 1 unit → t = 9, 0 left. The slice is not used up.

Step 15 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

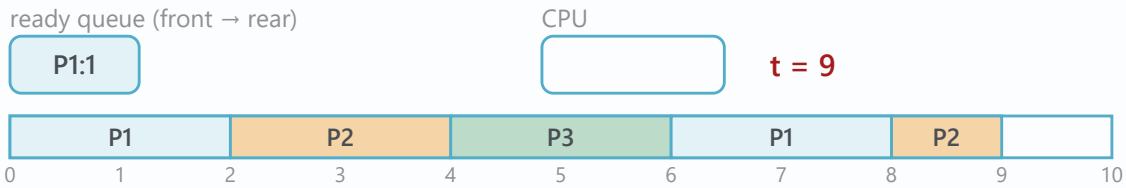
VARIABLES

`quantum = 2` `t = 9` `q = [P1:1]` `p = P2:0` `run = 1`

OUTPUT

```
P3 t=6
```

Code Trace: `roundRobin(q, 2)`



remaining 0 → P2 is **done** (t = 9); it does not rejoin.

Step 16 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
▶      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

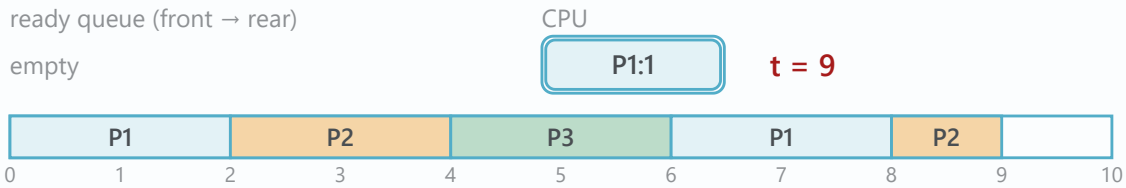
VARIABLES

`quantum = 2` `t = 9` `q = [P1:1]` `p = P2:0` `run = 1`

OUTPUT

```
P3 t=6
P2 t=9
```

Code Trace: `roundRobin(q, 2)`



Not empty → **p = q.poll()** = P1 (1 left).

Step 17 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
▶      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

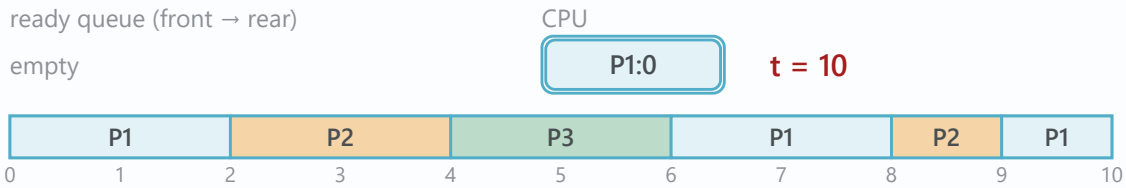
VARIABLES

`quantum = 2` `t = 9` `q = []`
`p = P1:1` `run = –`

OUTPUT

```
P3 t=6
P2 t=9
```

Code Trace: `roundRobin(q, 2)`



run = min(2, 1) = 1; P1 runs for 1 unit → t = 10, 0 left. The slice is not used up.

Step 18 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
▶      int run = Math.min(quantum, p.remaining);
▶      t += run;
▶      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

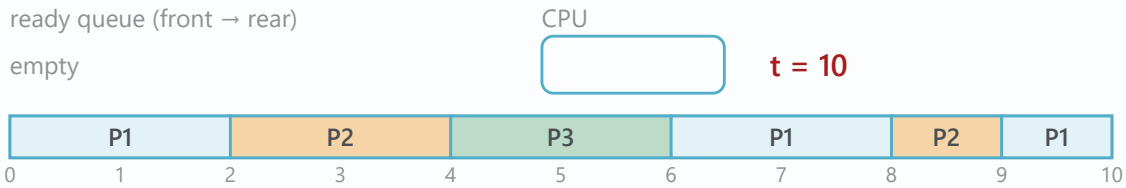
VARIABLES

`quantum = 2` `t = 10` `q = []`
`p = P1:0` `run = 1`

OUTPUT

```
P3 t=6
P2 t=9
```

Code Trace: `roundRobin(q, 2)`



remaining 0 → P1 is **done** (t = 10); it does not rejoin.

Step 19 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
3    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
▶      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

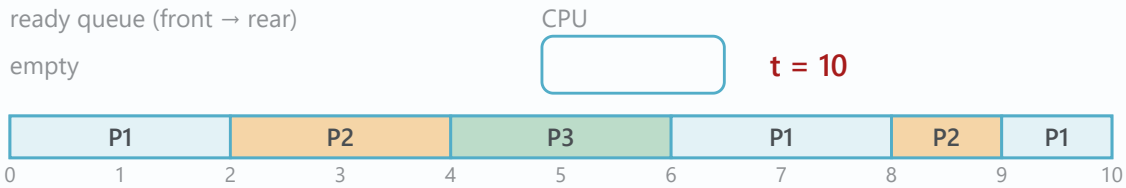
VARIABLES

`quantum = 2` `t = 10` `q = []` `p = P1:0` `run = 1`

OUTPUT

```
P3 t=6
P2 t=9
P1 t=10
```

Code Trace: `roundRobin(q, 2)`



The queue is empty → the loop ends. Completion: P3 = 6, P2 = 9, P1 = 10. Each round is one poll plus at most one offer: O(1).

Step 20 / 20

```
1  void roundRobin(Queue<Process> q, int quantum) {
2    int t = 0;
▶    while (!q.isEmpty()) {
4      Process p = q.poll();
5      int run = Math.min(quantum, p.remaining);
6      t += run;
7      p.remaining -= run;
8      if (p.remaining > 0) q.offer(p);
9      else System.out.println(p.name + " t=" + t);
10   }
11 }
```

VARIABLES

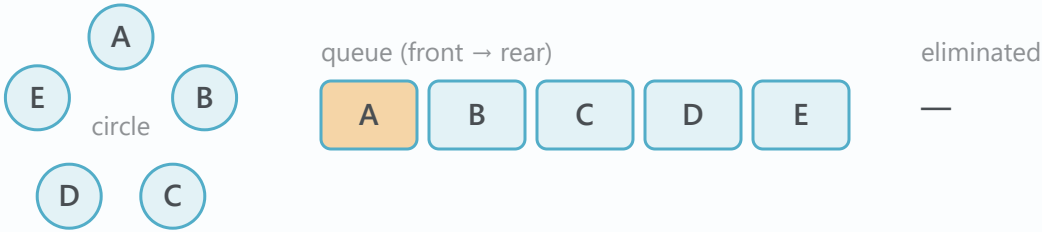
`quantum = 2` `t = 10` `q = []` `p = –` `run = –`

OUTPUT

```
P3 t=6
P2 t=9
P1 t=10
```

APPLICATIONS · STEP THROUGH THE CODE

Code Trace: The Josephus Problem (n = 5, k = 3)



Five people stand in a circle. Each round, the first **k − 1 = 2** move to the rear and the 3rd is eliminated. A queue rotates the circle naturally.

Step 1 / 14

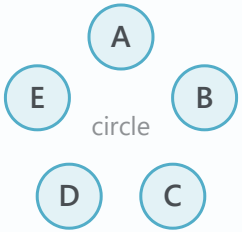
```
1 String josephus(Queue<String> q, int k) {
▶   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

VARIABLES

`k = 3` `q = [A, B, C, D, E]`

OUTPUT

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



A moved to the rear (poll + offer)

eliminated

—

i = 1: `q.offer(q.poll())` → A moves to the rear.

Step 2 / 14

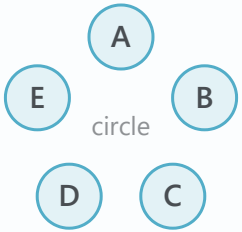
```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

VARIABLES

`k = 3` `q = [B, C, D, E, A]` `i = 1`

OUTPUT

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



B moved to the rear (poll + offer)

eliminated

—

i = 2: `q.offer(q.poll())` → B moves to the rear.

Step 3 / 14

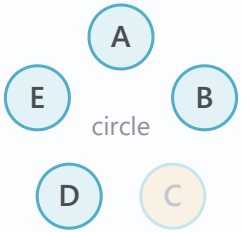
```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

VARIABLES

`k = 3` `q = [C, D, E, A, B]` `i = 2`

OUTPUT

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



C eliminated

eliminated

C

C leaves the front and is eliminated. 4 left.

Step 4 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
▶     String out = q.poll();
▶     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

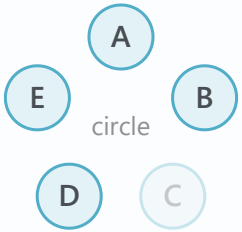
VARIABLES

`k = 3` `q = [D, E, A, B]` `out = C`

OUTPUT

```
C eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



D moved to the rear (poll + offer)

eliminated

C

i = 1: `q.offer(q.poll())` → D moves to the rear.

Step 5 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

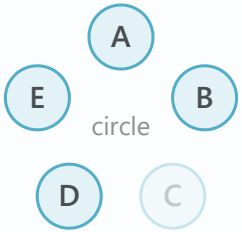
VARIABLES

`k = 3` `q = [E, A, B, D]` `i = 1`

OUTPUT

```
C eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



E moved to the rear (poll + offer)

eliminated

C

i = 2: `q.offer(q.poll())` → E moves to the rear.

Step 6 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

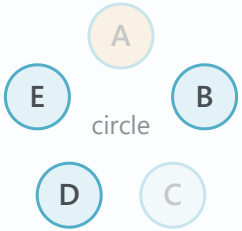
VARIABLES

`k = 3` `q = [A, B, D, E]` `i = 2`

OUTPUT

```
C eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)

B D E

A eliminated

eliminated

C A

A leaves the front and is eliminated. 3 left.

Step 7 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
▶     String out = q.poll();
▶     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

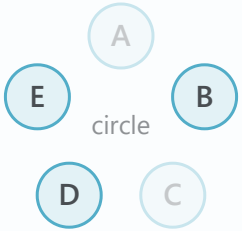
VARIABLES

`k = 3` `q = [B, D, E]` `out = A`

OUTPUT

```
C eliminated
A eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



B moved to the rear (poll + offer)

eliminated

C A

i = 1: `q.offer(q.poll())` → B moves to the rear.

Step 8 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

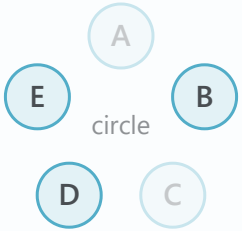
VARIABLES

`k = 3` `q = [D, E, B]` `i = 1`

OUTPUT

```
C eliminated
A eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



D moved to the rear (poll + offer)

eliminated

C A

i = 2: `q.offer(q.poll())` → D moves to the rear.

Step 9 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

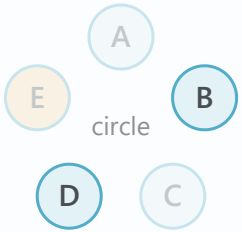
VARIABLES

`k = 3` `q = [E, B, D]` `i = 2`

OUTPUT

```
C eliminated
A eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)

B D

E eliminated

eliminated

C A E

E leaves the front and is eliminated. 2 left.

Step 10 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
▶     String out = q.poll();
▶     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

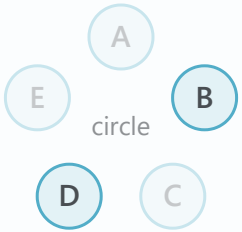
VARIABLES

`k = 3` `q = [B, D]` `out = E`

OUTPUT

```
C eliminated
A eliminated
E eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)



B moved to the rear (poll + offer)

eliminated

C A E

i = 1: `q.offer(q.poll())` → B moves to the rear.

Step 11 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

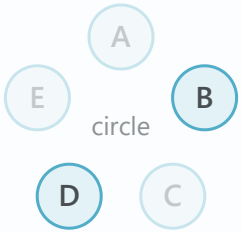
VARIABLES

`k = 3` `q = [D, B]` `i = 1`

OUTPUT

```
C eliminated
A eliminated
E eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)

B D

D moved to the rear (poll + offer)

eliminated

C A E

i = 2: `q.offer(q.poll())` → D moves to the rear.


```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
▶     for (int i = 1; i < k; i++)
▶       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

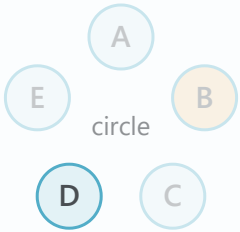
VARIABLES

`k = 3` `q = [B, D]` `i = 2`

OUTPUT

```
C eliminated
A eliminated
E eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)

D

B eliminated

eliminated

C A E B

B leaves the front and is eliminated. 1 left.

Step 13 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
▶     String out = q.poll();
▶     System.out.println(out + " eliminated");
7   }
8   return q.poll();
9 }
```

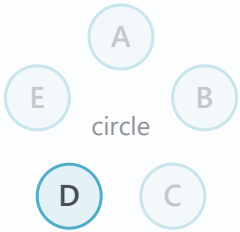
VARIABLES

`k = 3` `q = [D]` `out = B`

OUTPUT

```
C eliminated
A eliminated
E eliminated
B eliminated
```

Code Trace: The Josephus Problem (n = 5, k = 3)



queue (front → rear)

D

eliminated

C A E B

One person is left: the winner is **D**. n people, k steps: O(n·k) queue operations.

Step 14 / 14

```
1 String josephus(Queue<String> q, int k) {
2   while (q.size() > 1) {
3     for (int i = 1; i < k; i++)
4       q.offer(q.poll());      // front to rear
5     String out = q.poll();
6     System.out.println(out + " eliminated");
7   }
▶   return q.poll();
9 }
```

VARIABLES

`k = 3` `q = [D]`

OUTPUT

```
C eliminated
A eliminated
E eliminated
B eliminated
```

APPLICATIONS

Example: Binary Numbers from 1 to n

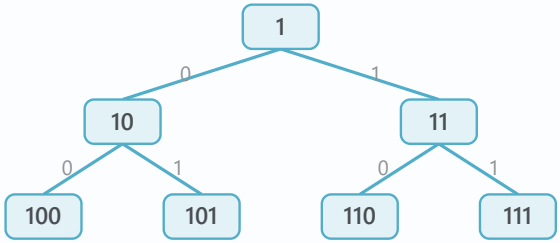
Use a queue to generate the binary representations of the numbers from 1 to n.

Example 1

n = 3 → {"1", "10", "11"}

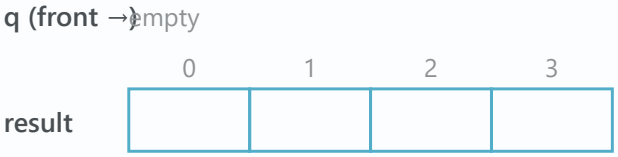
Example 2

n = 5 → {"1", "10", "11", "100", "101"}



Idea: x has two successors, `x + "0"` (2x) and `x + "1"` (2x + 1). Dequeue a number, output it, and enqueue its successors: the numbers come out level by level, in increasing order (breadth-first).

Code Trace: `generateBinary(4)`



`result` : an array with 4 empty slots.

Step 1 / 17

```
1  String[] generateBinary(int n) {
▶      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

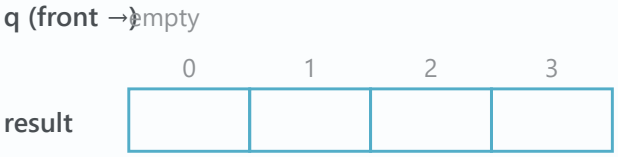
`n = 4` `i = –` `q = –`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



An empty queue is created (a LinkedList behind the Queue interface).

Step 2 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
▶      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

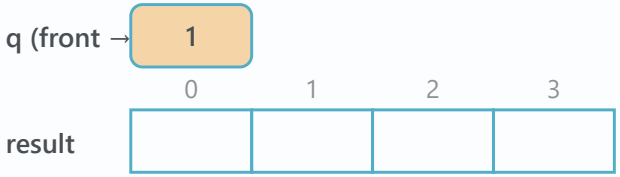
VARIABLES

`n = 4` `i = –` `q = []`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



The root “1” is enqueued.

Step 3 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
▶      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

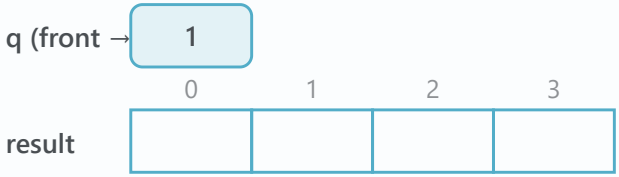
VARIABLES

`n = 4` `i = –` `q = [1]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



i = 0 < 4 → loop body.

Step 4 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

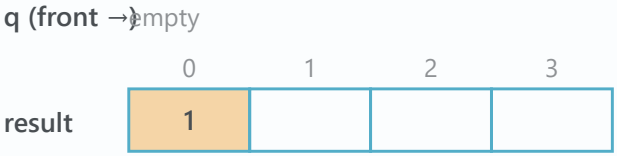
VARIABLES

`n = 4` `i = 0` `q = [1]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



result[0] = q.poll() = "1": the smallest number, at the front of the queue.

Step 5 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

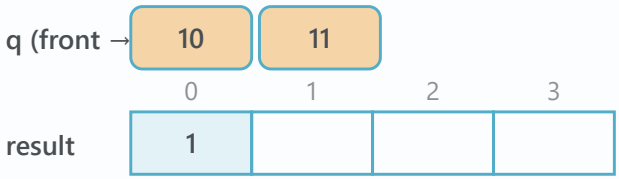
`n = 4` `i = 0` `q = []`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



“10” and “11” are enqueued: the successors of “1”, to come out in turn.

Step 6 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
▶          q.offer(result[i] + "0");
▶          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

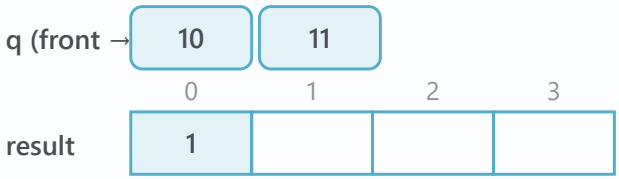
VARIABLES

`n = 4` `i = 0` `q = [10, 11]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



i = 1 < 4 → loop body.

Step 7 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

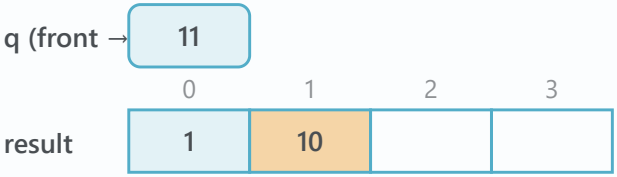
VARIABLES

`n = 4` `i = 1` `q = [10, 11]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



result[1] = q.poll() = "10": the smallest number, at the front of the queue.

Step 8 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

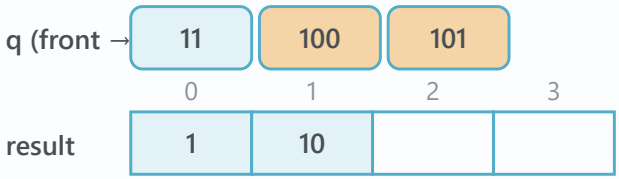
`n = 4` `i = 1` `q = [11]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



“100” and “101” are enqueued: the successors of “10”, to come out in turn.

Step 9 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
▶          q.offer(result[i] + "0");
▶          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

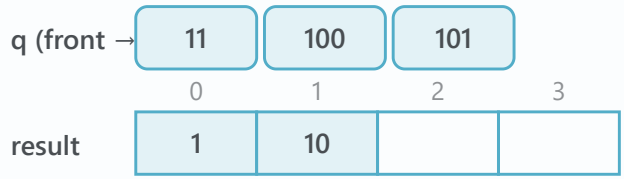
VARIABLES

`n = 4` `i = 1` `q = [11, 100, 101]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



i = 2 < 4 → loop body.

Step 10 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

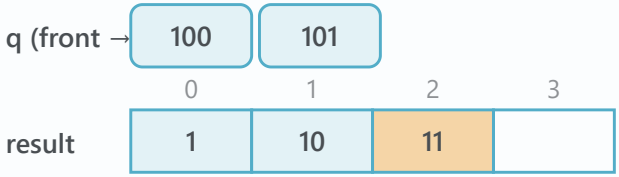
VARIABLES

`n = 4` `i = 2` `q = [11, 100, 101]`

CALL STACK

- `generateBinary(4)`
- `main()`

Code Trace: `generateBinary(4)`



result[2] = q.poll() = "11": the smallest number, at the front of the queue.

Step 11 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

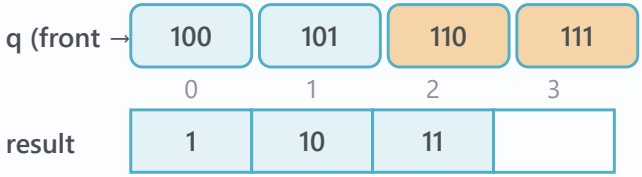
`n = 4` `i = 2` `q = [100, 101]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



“110” and “111” are enqueued: the successors of “11”, to come out in turn.

Step 12 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
▶          q.offer(result[i] + "0");
▶          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

`n = 4` `i = 2`

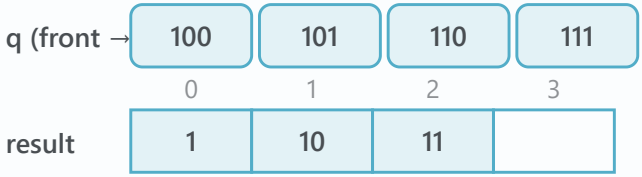
`q = [100, 101, 110, 111]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



i = 3 < 4 → loop body.

Step 13 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

`n = 4` `i = 3`

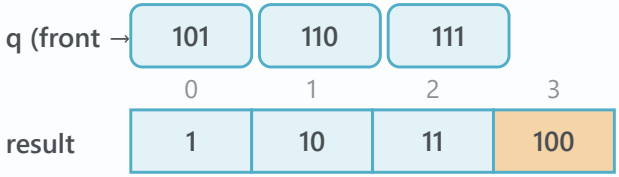
`q = [100, 101, 110, 111]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



result[3] = q.poll() = "100": the smallest number, at the front of the queue.

Step 14 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
▶          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

`n = 4` `i = 3` `q = [101, 110, 111]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



“1000” and “1001” are enqueued: the successors of “100”, to come out in turn.

Step 15 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
▶          q.offer(result[i] + "0");
▶          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

`n = 4` `i = 3`

`q = [101, 110, 111, 1000, 1001]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



i = 4 < 4 is false → the loop ends.


```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
▶      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
10     return result;
11 }
```

VARIABLES

`n = 4` `i = 4`

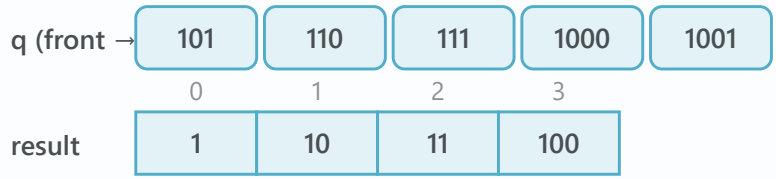
`q = [101, 110, 111, 1000, 1001]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: `generateBinary(4)`



return {"1", "10", "11", "100"}. n polls + 2n offers → O(n) queue operations (the strings have length O(log n), so the total character work is O(n log n)).

Step 17 / 17

```
1  String[] generateBinary(int n) {
2      String[] result = new String[n];
3      Queue<String> q = new LinkedList<>();
4      q.offer("1");
5      for (int i = 0; i < n; i++) {
6          result[i] = q.poll();
7          q.offer(result[i] + "0");
8          q.offer(result[i] + "1");
9      }
▶      return result;
11 }
```

VARIABLES

`n = 4` `i = –`

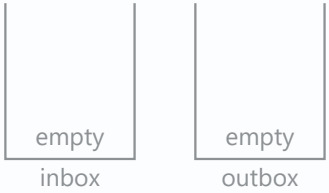
`q = [101, 110, 111, 1000, 1001]`

CALL STACK

`generateBinary(4)`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

inbox takes insertions and **outbox** serves removals (stack tops are drawn at the top).

Step 1 / 17

```
▶ Deque<Integer> inbox = new ArrayDeque<>();
▶ Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

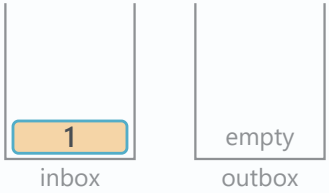
VARIABLES

`inbox = []` `outbox = []`

CALL STACK

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) | enqueue(2) | enqueue(3) | dequeue()
enqueue(4) | dequeue() | dequeue() | dequeue()

output order: —

enqueue(1): `inbox.push(1)` → O(1).

Step 2 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
▶    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

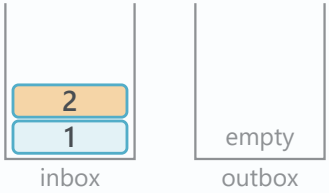
inbox = [1] | outbox = [] | x = 1

CALL STACK

enqueue(1)

main()

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

enqueue(2): `inbox.push(2)` → O(1).

Step 3 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
▶    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

`inbox = [1, 2]` `outbox = []`

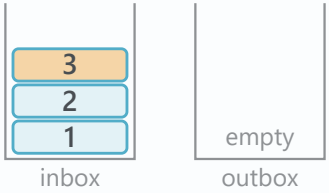
`x = 2`

CALL STACK

`enqueue(2)`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

enqueue(3): `inbox.push(3)` → O(1).

Step 4 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
▶    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

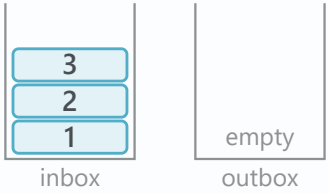
`inbox = [1, 2, 3]` `outbox = []` `x = 3`

CALL STACK

`enqueue(3)`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

dequeue: outbox is **empty** → move everything from inbox to outbox.

Step 5 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
▶    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

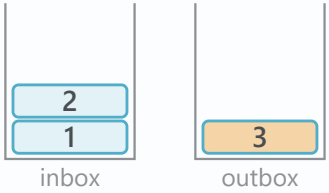
`inbox = [1, 2, 3]` `outbox = []`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

3: pop from inbox, push onto outbox. The transfer **reverses** the order; the oldest item ends up on top.

Step 6 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
▶      while (!inbox.isEmpty())
▶        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

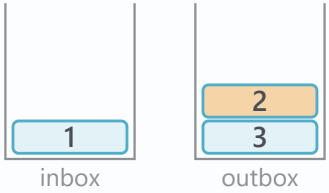
`inbox = [1, 2]` `outbox = [3]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

2: pop from inbox, push onto outbox. The transfer **reverses** the order; the oldest item ends up on top.

Step 7 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
▶      while (!inbox.isEmpty())
▶        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

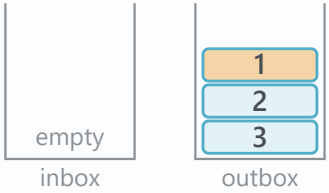
`inbox = [1]` `outbox = [3, 2]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: —

1: pop from inbox, push onto outbox. The transfer **reverses** the order; the oldest item ends up on top.

Step 8 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
▶      while (!inbox.isEmpty())
▶        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

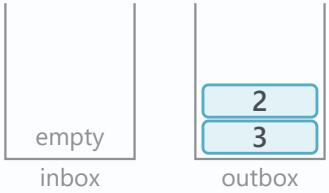
`inbox = []` `outbox = [3, 2, 1]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1

`outbox.pop()` → **1** (FIFO: oldest first).

Step 9 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
   ▶  return outbox.pop();
11 }
```

VARIABLES

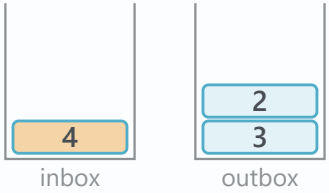
`inbox = []` `outbox = [3, 2]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(4): `inbox.push(4)` → O(1).

enqueue(1)	enqueue(2)	enqueue(3)	dequeue()
enqueue(4)	dequeue()	dequeue()	dequeue()

output order: 1

Step 10 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
▶    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

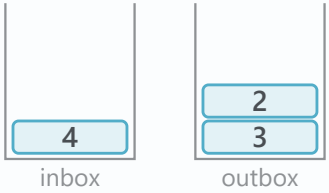
VARIABLES

`inbox = [4]` `outbox = [3, 2]` `x = 4`

CALL STACK

- **enqueue(4)**
- main()

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1

dequeue: outbox is not empty → **no** transfer.

Step 11 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
▶    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

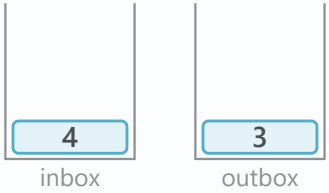
`inbox = [4]` `outbox = [3, 2]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1)	enqueue(2)	enqueue(3)	dequeue()
enqueue(4)	dequeue()	dequeue()	dequeue()

output order: 1 2

`outbox.pop()` → **2** (FIFO: oldest first).

Step 12 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
▶    return outbox.pop();
11 }
```

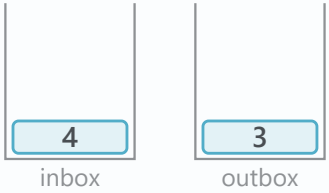
VARIABLES

`inbox = [4]` `outbox = [3]`

CALL STACK

- **dequeue()**
- main()

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1 2

dequeue: outbox is not empty → **no** transfer.


```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
▶    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

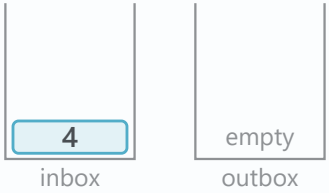
`inbox = [4]` `outbox = [3]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1)	enqueue(2)	enqueue(3)	dequeue()
enqueue(4)	dequeue()	dequeue()	dequeue()

output order: 1 2 3

`outbox.pop()` → **3** (FIFO: oldest first).

Step 14 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
▶    return outbox.pop();
11 }
```

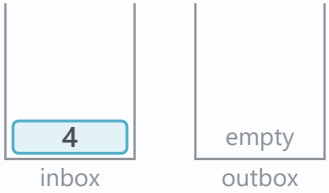
VARIABLES

`inbox = [4]` `outbox = []`

CALL STACK

- **dequeue()**
- main()

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1 2 3

dequeue: outbox is **empty** → move everything from inbox to outbox.

Step 15 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
▶    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

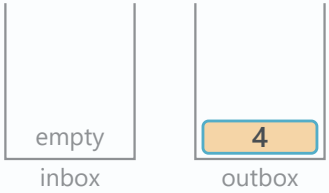
`inbox = [4]` `outbox = []`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1 2 3

4: pop from inbox, push onto outbox. The transfer **reverses** the order; the oldest item ends up on top.


```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
▶      while (!inbox.isEmpty())
▶        outbox.push(inbox.pop());
10   return outbox.pop();
11 }
```

VARIABLES

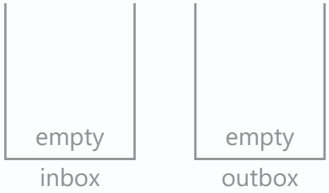
`inbox = []` `outbox = [4]`

CALL STACK

`dequeue()`

`main()`

Code Trace: A Queue from Two Stacks



enqueue(1) enqueue(2) enqueue(3) dequeue()
enqueue(4) dequeue() dequeue() dequeue()

output order: 1 2 3 4

`outbox.pop()` → **4**. Output 1, 2, 3, 4 matches the input order. Each item is transferred at most once → dequeue is **amortized O(1)** (a single call is O(n) in the worst case).

Step 17 / 17

```
1  Deque<Integer> inbox = new ArrayDeque<>();
2  Deque<Integer> outbox = new ArrayDeque<>();
3  void enqueue(int x) {
4    inbox.push(x);
5  }
6  int dequeue() {
7    if (outbox.isEmpty())
8      while (!inbox.isEmpty())
9        outbox.push(inbox.pop());
▶    return outbox.pop();
11 }
```

VARIABLES

`inbox = []` `outbox = []`

CALL STACK

dequeue()

main()

SUMMARY

Common Mistakes

Forgetting rear on the last removal

When a linked queue loses its last node, `front = null`; without `rear = null`, rear points to a removed node.

Mixing up full and empty

With only front and rear, `front == rear` both when empty and when full; add a `size` counter or leave one slot empty.

`Arrays.copyOf` when growing

It scrambles a circular queue that has wrapped around; copy the items in `(front + k) % N` order.

add / offer, remove / poll

`add`, `remove`, and `element` throw exceptions; `offer`, `poll`, and `peek` return false / null.

`PriorityQueue` is not FIFO

`Queue<Integer> q = new PriorityQueue<>()` hands out items by priority, not by arrival order.

`ArrayDeque` and null

`ArrayDeque` rejects null items (`NullPointerException`); a null would be confused with the “empty” result.

SUMMARY

Summary: Costs of Queue Implementations

Implementation	Insert	Remove	Front	Note
Simple array (with shifting)	O(1)	O(n)	O(1)	shifts on every removal
Circular array	O(1)	O(1)	O(1)	fixed capacity, `% N`
Linked list	O(1)	O(1)	O(1)	front and rear references
Deque (doubly linked list)	O(1) both ends	O(1) both ends	O(1)	`prev` reference
Queue from two stacks	O(1)	O(1) amortized	O(1) amortized	each item is transferred once

FIFO
Insert at the rear, remove from the front.

Circular array
Telling empty from full: a `size` counter or one empty slot.

Edge cases
Removing from an empty queue (underflow); `rear = null` when the last item leaves.

SUMMARY

Interactive Simulators

Queue Simulator

enqueue, dequeue, front, rear, and search, step by step.

Deque Simulator

The effect of inserting and removing at both ends.

Circular Queue Simulator

front, size, wraparound with mod, and growing a full array, in array and ring views.

Round-Robin Scheduler

A FIFO ready queue, adjustable quantum, and process metrics.

SUMMARY

Sample Problems

1. Starting from an empty queue: enqueue(4), enqueue(7), dequeue(), enqueue(1), enqueue(9), dequeue(). What is first(), and what is left in the queue?
2. N = 6, front = 4, size = 3: which indices hold the items, and where does the next enqueue write?
3. With the “one empty slot” method and N = 10, how many items fit at most? What is the full condition?
4. What happens if `if (front == null) rear = null;` is removed from dequeue?
5. How do you build a queue from two stacks? What is the amortized cost of dequeue?
6. How can a deque check whether a word is a palindrome?

▶ **Show answers**

SUMMARY

Exercises

1. Reverse a queue using only queue operations and one stack.
2. Build a stack from one queue: how do you write push, and at what cost?
3. Josephus: people numbered 1..7, k = 3. In what order are they eliminated, and who wins?
4. A circular queue with N = 5, front = 3, size = 5 is grown. In what order do the items appear in the new array?
5. Reverse the first k items of a queue without changing the order of the rest.
6. What changes in the deque algorithm for the sliding window minimum?

▶ **Hints / answers**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Lafore, *Data Structures and Algorithms in Java*, 2nd ed., 2002.
- Oracle, Java SE 17 API documentation for `Queue`, `Deque`, `ArrayDeque`, and `PriorityQueue`.

The End

Chapter 8: Queues and Deques

Next deck: Chapter 9: Trees · Study questions · Simulators: Queue, Deque, Circular Queue, Round-Robin Scheduler · Glossary