Chapter 7: Queues

Data Structures and Algorithms with C

FIFO · Queue ADT · Linear queue · Circular queue · Linked queue · Deque · Scheduling · BFS

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/dsa_c/slides/Chapter_07_Queues.html

INTRODUCTION

Learning Outcomes

Define FIFO

Model a queue as $Q = \{a_0, \ldots, a_{n-1}\}$ and identify its front and rear.

See the linear queue's flaw

Explain why cells freed at the front can't be reused, and what shifting costs.

Write a circular queue

Wrap around with `% MAX` and tell full from empty in three ways.

Build a linked queue

Enqueue and dequeue in O(1) with `front` and `rear` pointers.

Know the deque

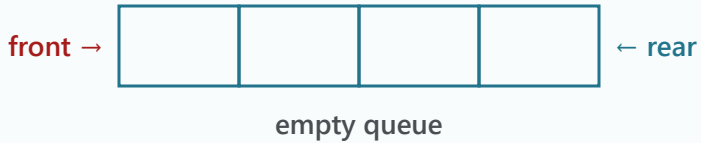
Implement a double-ended queue, with insertion and removal at both ends, in a circular array.

Apply queues

Use a queue in a time-sliced job scheduling simulation; build a queue from two stacks.

The Queue: First In, First Out

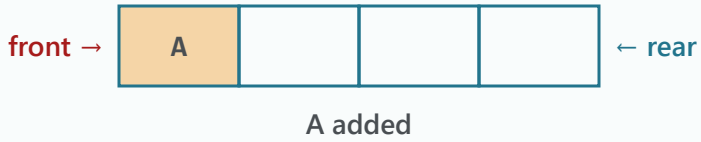
In a queue Q = {a_0, a_1, ..., a_{n-1}}, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



Empty queue: front and rear at the same place.

The Queue: First In, First Out

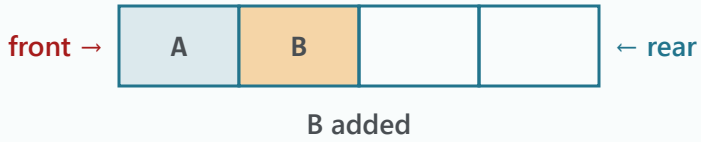
In a queue Q = {a_0, a_1, …, a_{n-1}}, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



A added at the rear.

The Queue: First In, First Out

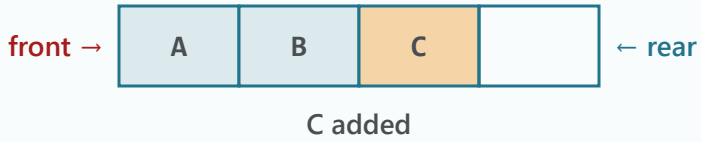
In a queue $Q = \{a_0, a_1, \ldots, a_{n-1}\}$, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



B added at the rear.

The Queue: First In, First Out

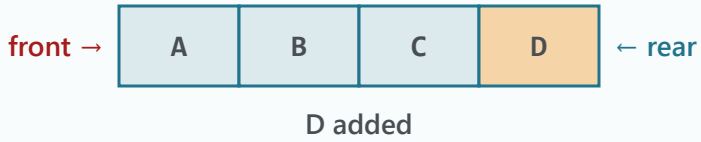
In a queue $Q = \{a_0, a_1, \ldots, a_{n-1}\}$, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



C added at the rear.

The Queue: First In, First Out

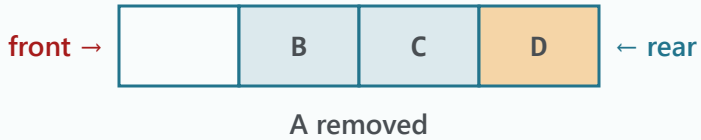
In a queue Q = {a_0, a_1, …, a_{n-1}}, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



D added at the rear.

The Queue: First In, First Out

In a queue Q = {a_0, a_1, …, a_{n-1}}, unlike a stack, the first item to be processed is a_0 (the **front**); a_{n-1} sits at the **rear**. Items are inserted at the rear and deleted at the front: **FIFO** (first-in-first-out).



The first one out is **A**, the first one in: **FIFO**. The front cell is now free.

Abstract Data Type: Queue

Operation	Meaning	Error case
`queue_create(max)`	Empty queue for at most max items	—
`queue_full(Q)`	True if no room is left	—
`enqueue(Q, x)`	Adds x at the **rear**	Overflow if full
`queue_empty(Q)`	True if it holds no items	—
`dequeue(Q)`	Removes and returns the **front** item	Underflow if empty
`queue_front(Q)` · peek	Reads the front item without removing it	Error if empty

Stack

One end: push and pop at the **top**. LIFO.

Queue

Two ends: insert at the **rear**, remove at the **front**. FIFO.

Where Queues Show Up

Operating systems

Ready processes, print jobs, and disk requests wait their turn in queues.

Buffers

Keyboard input, network packets, audio streams: a circular buffer between producer and consumer.

Breadth-first search

BFS and level-order tree traversal: the children of the current node go to the rear, the next node comes from the front.

Simulation

Bank tellers, traffic lights, job scheduling: arriving customers join the line.

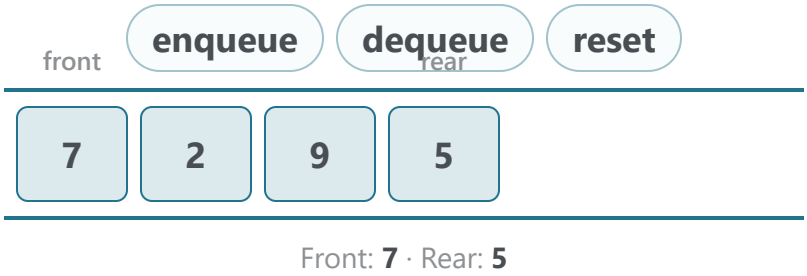
Message queues

Servers hand off work in order, without losing any.

Priority queue

Called a “queue” but not FIFO: the highest-priority item leaves first.

Add at the Rear, Remove at the Front



Front: **7** · Rear: **5**

Items leave in **the same order** they arrived. With a stack it was the reverse.

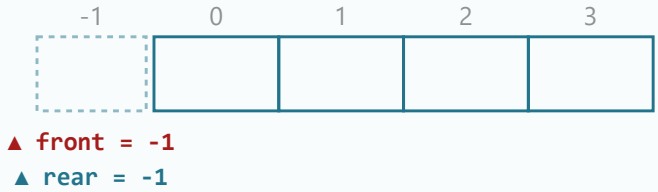
Linear Queue in an Array

```
#define MAX_QUEUE_SIZE 100
element queue[MAX_QUEUE_SIZE];
int front = -1;      /* one before the first item */
int rear = -1;       /* last item */

void enqueue(int *rear, element item) {
    if (*rear >= MAX_QUEUE_SIZE - 1)   /* any room? */
        queue_full();           /* warn and exit */
    queue[++*rear] = item;      /* bump, then store */
}
element dequeue(int *front, int rear) {
    if (*front == rear)
        queue_empty();          /* warn and exit if empty */
    return queue[++*front];     /* bump, return the front */
}
```

- `front` points **one before** the first item, not at it; `rear` points at the last item.
- Empty: **front == rear** (both −1 at the start).
- Full: **rear == MAX − 1**, even if there is room at the front.
- `dequeue` doesn't change `rear`, so it takes it **by value**.
- `queue_full` / `queue_empty`: print a message and `exit`, as with the stack.

Code Trace: The Linear Queue and “False Fullness”



Initially **front = rear = −1**: the queue is empty. MAX_QUEUE_SIZE = 4. Enqueue A, B, C; dequeue twice; enqueue D, E.

Step 1 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

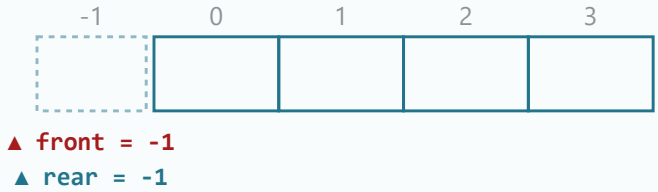
front = **−1** rear = **−1**

CALL STACK

main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



-1 ≥ 3? No.

Step 2 / 15

```
1  void enqueue(int *rear, element item) {
▶    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

front = -1 *rear = -1 item = A

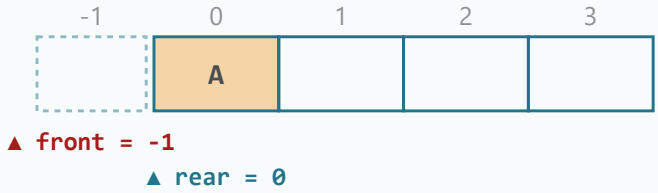
CALL STACK

enqueue(&rear, A)

main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*rear → 0; **queue[0] = A**.

Step 3 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();     /* "Queue is full!!!", exit */
▶    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();    /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

front = **-1** | *rear = **0** | item = **A**

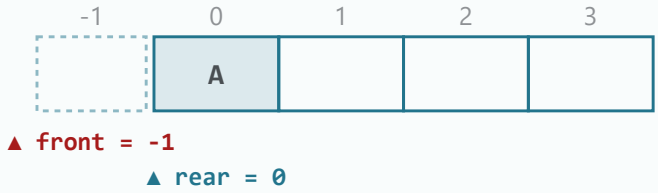
CALL STACK

enqueue(&rear, A)

main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



0 ≥ 3? No.

Step 4 / 15

```
1  void enqueue(int *rear, element item) {
▶    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`front = -1` `*rear = 0` `item = B`

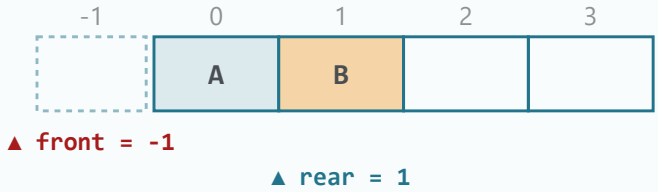
CALL STACK

`enqueue(&rear, B)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*rear → 1; **queue[1] = B**.

Step 5 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();     /* "Queue is full!!!", exit */
▶    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();    /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

front = **-1** | *rear = **1** | item = **B**

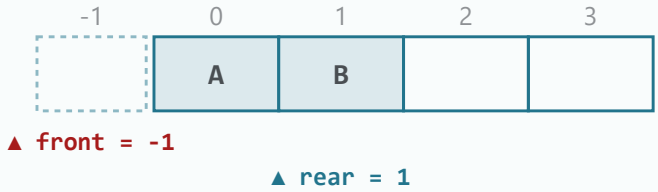
CALL STACK

enqueue(&rear, B)

main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



1 ≥ 3? No.

Step 6 / 15

```
1  void enqueue(int *rear, element item) {
▶    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

front = **-1** | *rear = **1** | item = **C**

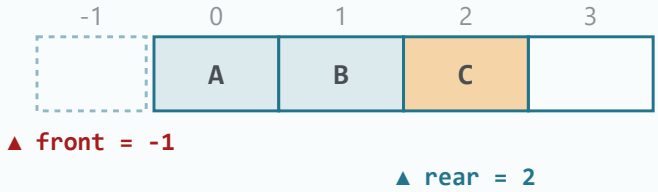
CALL STACK

enqueue(&rear, C)

main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*rear → 2; **queue[2] = C**.

Step 7 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
▶    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

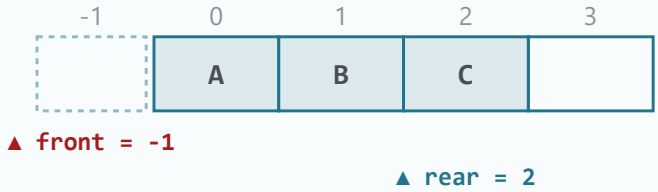
front = **-1** | *rear = **2** | item = **C**

CALL STACK

- enqueue(&rear, C)
- main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



front == rear (-1 == 2)? No, not empty.

Step 8 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();     /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
▶    if (*front == rear)
8      queue_empty();    /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`*front = -1` `rear = 2`

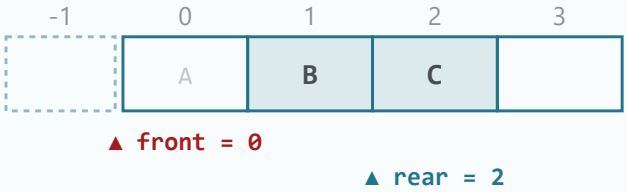
CALL STACK

`dequeue(&front, rear)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*front → 0; queue[0] = **A** is returned. The cell is no longer used.

Step 9 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
▶    return queue[++*front];
10 }
```

VARIABLES

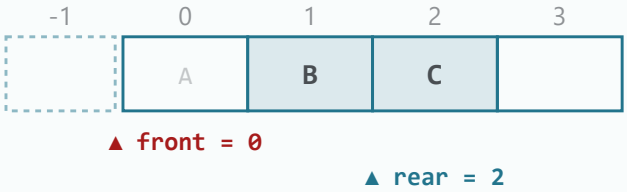
*front = 0 | rear = 2 | returned = A

CALL STACK

dequeue(&front, rear)
main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



front == rear (0 == 2)? No, not empty.

Step 10 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
▶    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`*front = 0` `rear = 2`

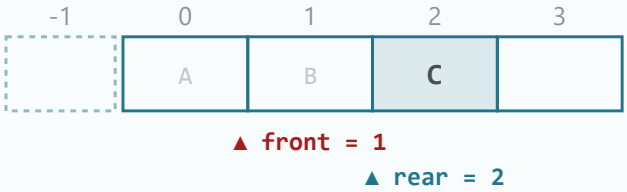
CALL STACK

`dequeue(&front, rear)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*front → 1; queue[1] = **B** is returned. The cell is no longer used.

Step 11 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
▶    return queue[++*front];
10 }
```

VARIABLES

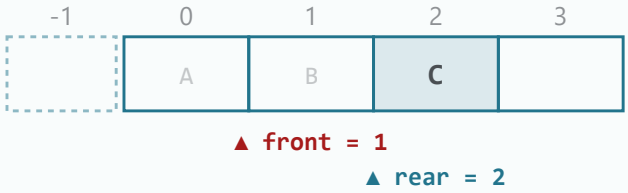
*front = 1 | rear = 2 | returned = B

CALL STACK

dequeue(&front, rear)
main()

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



2 ≥ 3? No.

Step 12 / 15

```
1  void enqueue(int *rear, element item) {
▶    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`front = 1` `*rear = 2` `item = D`

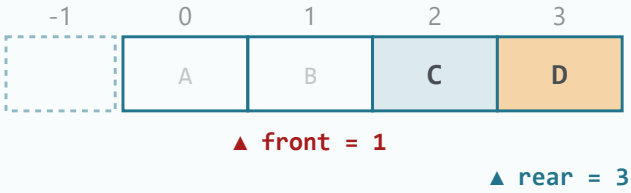
CALL STACK

`enqueue(&rear, D)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



++*rear → 3; **queue[3] = D**.

Step 13 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();     /* "Queue is full!!!", exit */
▶    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();    /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`front = 1` `*rear = 3` `item = D`

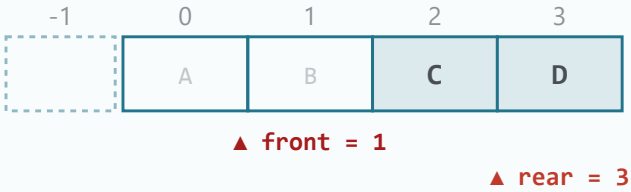
CALL STACK

`enqueue(&rear, D)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



3 ≥ 3 → counted as full! Yet cells 0 and 1 are empty.

Step 14 / 15

```
1  void enqueue(int *rear, element item) {
▶    if (*rear >= MAX_QUEUE_SIZE - 1)
3      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`front = 1` `*rear = 3` `item = E`

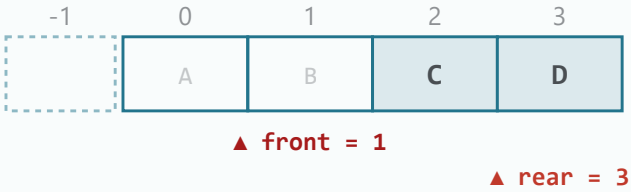
CALL STACK

`enqueue(&rear, E)`

`main()`

OUTPUT

Code Trace: The Linear Queue and “False Fullness”



`queue_full()` : the program ends. **The linear queue's flaw**: once rear hits the end, the free cells at the front can't be used.

Step 15 / 15

```
1  void enqueue(int *rear, element item) {
2    if (*rear >= MAX_QUEUE_SIZE - 1)
▶      queue_full();    /* "Queue is full!!!", exit */
4    queue[++*rear] = item;
5  }
6  element dequeue(int *front, int rear) {
7    if (*front == rear)
8      queue_empty();   /* "Queue is empty!!!", exit */
9    return queue[++*front];
10 }
```

VARIABLES

`front = 1` `*rear = 3` `item = E`

CALL STACK

`enqueue(&rear, E)`

`main()`

OUTPUT

```
Queue is full!!!
```

Free Space at the Front: Shift Everything?



front = 1, rear = 3: “full”

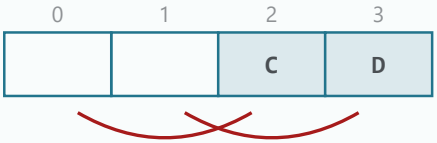
Two cells are free at the front, but rear has hit the end.

Step 1 / 3

> **INSTRUCTOR** “Shifting! The worst word in computer science! It means high cost!”

Shifting takes far too long, so the **circular queue** was devised: join the end of the array to its start, and the cells freed at the front get reused as if they were at the rear.

Free Space at the Front: Shift Everything?



shift every item to the start

Shifting moves each of the n items once → **O(n)**.

Step 2 / 3

INSTRUCTOR “Shifting! The worst word in computer science! It means high cost!”

Shifting takes far too long, so the **circular queue** was devised: join the end of the array to its start, and the cells freed at the front get reused as if they were at the rear.

Free Space at the Front: Shift Everything?



front = −1, rear = 1

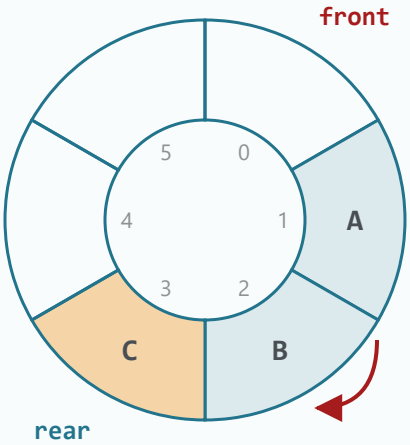
Room is made, but doing this on every “full” makes enqueue O(n).

Step 3 / 3

INSTRUCTOR “Shifting! The worst word in computer science! It means high cost!”

Shifting takes far too long, so the **circular queue** was devised: join the end of the array to its start, and the cells freed at the front get reused as if they were at the rear.

Circular Queue



- Think of the array's end as joined to its start: after 5 comes 0.
- Advance: `rear = (rear + 1) % MAX`, `front = (front + 1) % MAX`.
- `front` still points **one before** the first item; initially `front = rear = 0`.
- Empty: **front == rear**. Full: **(rear + 1) % MAX == front**.
- So **one** of the MAX cells **always stays empty**: at most MAX − 1 items.

Wrap-Around: `% MAX`

rear	rear + 1	(rear + 1) % 6	Meaning
3	4	4	normal step
4	5	5	last cell
5	6	**0**	**wrap**: back to the start
0	1	1	normal again

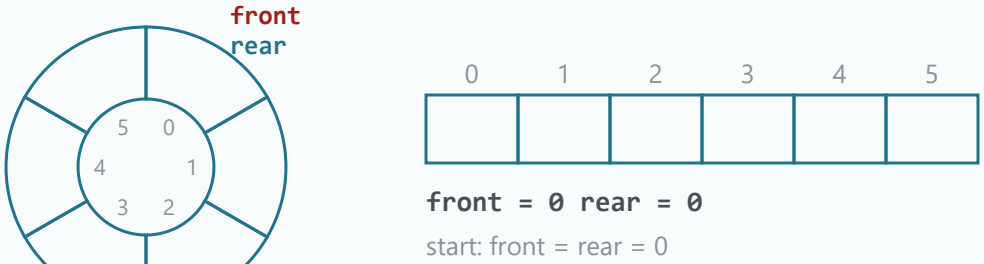
Stepping back

One step back: `(i - 1 + MAX) % MAX`. In C, `-1 % 6` is −1; adding `MAX` keeps the result non-negative (the deque uses this).

Item count

`(rear - front + MAX) % MAX`. Example: front = 4, rear = 1, MAX = 6 → (1 − 4 + 6) % 6 = 3.

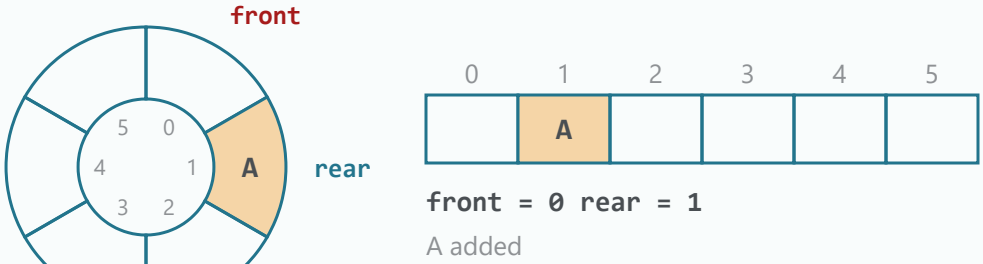
Circular Queue in Pictures: Wrapping with A…F



Empty queue: **front == rear** = 0.

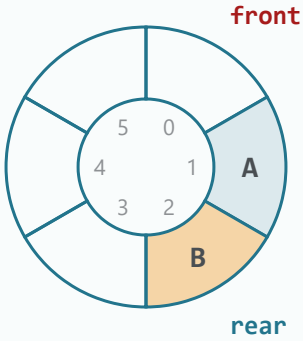
Step 1 / 8

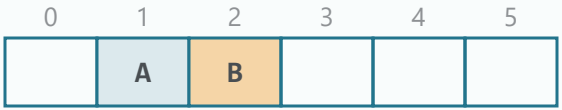
Circular Queue in Pictures: Wrapping with A…F



A added: rear = 1.

Circular Queue in Pictures: Wrapping with A…F



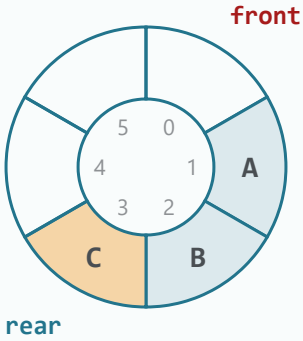


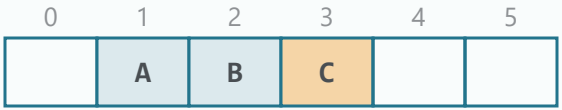
`front = 0 rear = 2`

B added

B added: rear = 2.

Circular Queue in Pictures: Wrapping with A…F



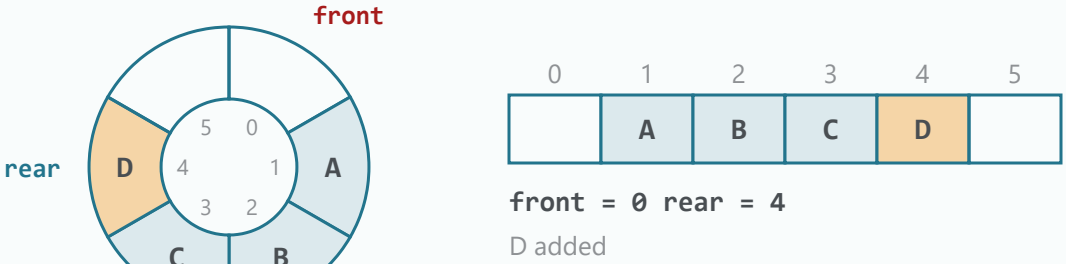


`front = 0 rear = 3`

C added

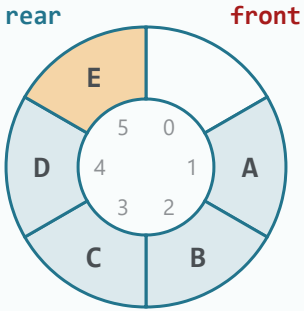
C added: rear = 3.

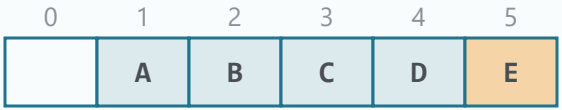
Circular Queue in Pictures: Wrapping with A…F



D added: rear = 4.

Circular Queue in Pictures: Wrapping with A…F



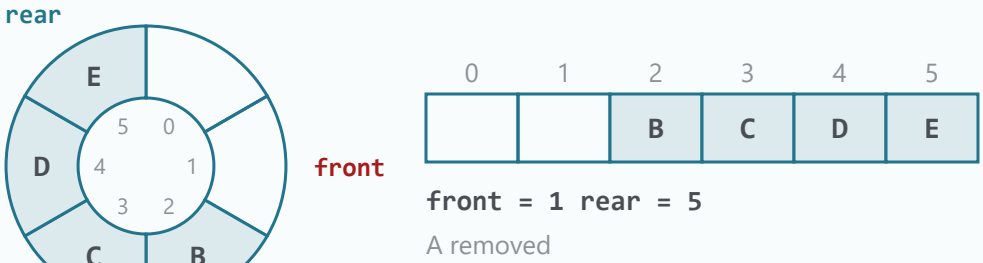


`front = 0 rear = 5`

E added

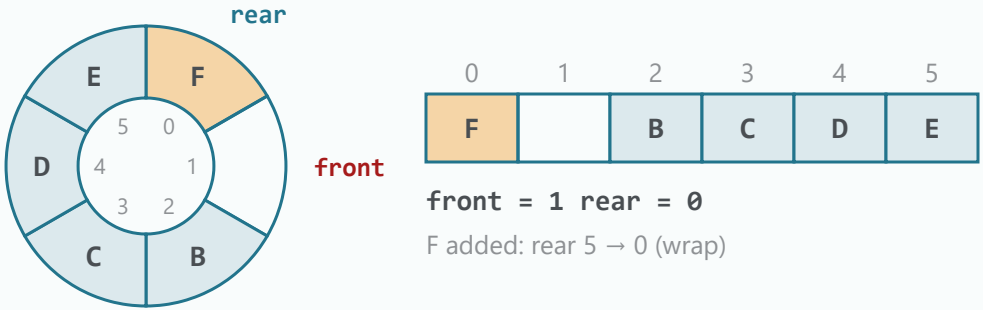
E added: rear = 5.

Circular Queue in Pictures: Wrapping with A…F



A removed: front = 1. Cells 0 and 1 are now both free.

Circular Queue in Pictures: Wrapping with A…F



F goes to rear = (5 + 1) % 6 = **0**: the cell freed at the front is reused as if it were at the rear. The queue is now **full**: (0 + 1) % 6 == front. 5 items in 6 cells.

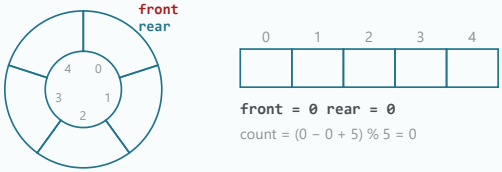
Circular Queue: C Code

```c
void enqueue(int front, int *rear, element item) {
    *rear = (*rear + 1) % MAX_QUEUE_SIZE; /* advance */
    if (front == *rear) queue_full();     /* full? */
    queue[*rear] = item;                  /* store */
}
element dequeue(int *front, int rear) {
    if (*front == rear) queue_empty();    /* empty? */
    *front = (*front + 1) % MAX_QUEUE_SIZE; /* advance */
    return queue[*front];                 /* return */
}
```

A subtle point: `enqueue` advances `*rear` first and checks for fullness **afterwards**. Since `queue_full` ends the program, that's harmless here; but in a version that returns an error, `rear` is left corrupted (equal to front, so the queue looks **empty**). The safe version tries the new position in a temporary first (Horowitz–Sahni also roll `rear` back).

INSTRUCTOR "Which is the best university in the world for computer engineering? Harvard? Then you should study as if you were thinking, 'I'm at Harvard, and if I fail I'll be kicked out and have to go to a worse school.'"

Code Trace: A Safe Circular Queue



MAX_QUEUE_SIZE = 5, **front = rear = 0**. Enqueue A, B, C, D, E; dequeue twice; enqueue E, F, G.

Step 1 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

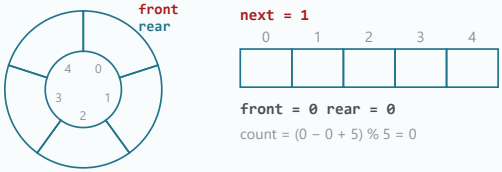
`front = 0` `rear = 0`

CALL STACK

`main()`

OUTPUT

Code Trace: A Safe Circular Queue



next = (0 + 1) % 5 = **1**; 1 ≠ front, there is room.

Step 2 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

front = 0 | *rear = 0 | next = 1 | x = A

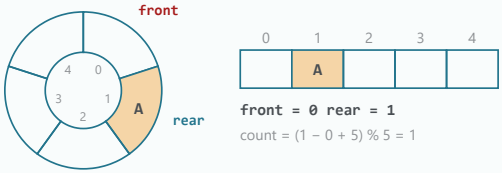
CALL STACK

enqueue(front, &rear, A)

main()

OUTPUT

Code Trace: A Safe Circular Queue



rear = 1, queue[1] = A.

Step 3 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

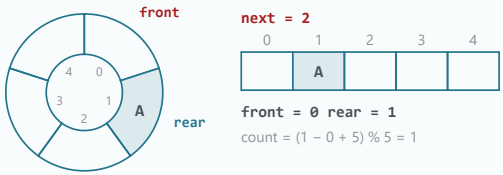
front = 0 | *rear = 1 | next = 1 | x = A

CALL STACK

enqueue(front, &rear, A)
main()

OUTPUT

Code Trace: A Safe Circular Queue



next = (1 + 1) % 5 = **2**; 2 ≠ front, there is room.

Step 4 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 0` `*rear = 1` `next = 2` `x = B`

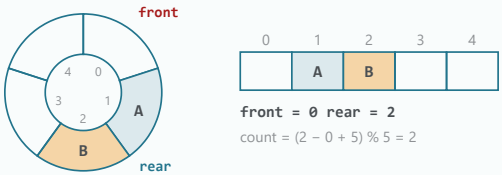
CALL STACK

`enqueue(front, &rear, B)`

`main()`

OUTPUT

Code Trace: A Safe Circular Queue



rear = 2, queue[2] = B.

Step 5 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

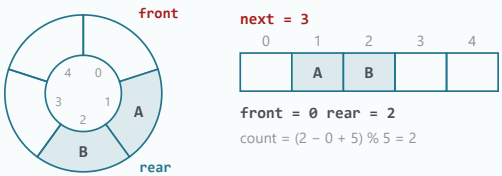
front = 0 | *rear = 2 | next = 2 | x = B

CALL STACK

enqueue(front, &rear, B)
main()

OUTPUT

Code Trace: A Safe Circular Queue



next = (2 + 1) % 5 = **3**; 3 ≠ front, there is room.

Step 6 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 0` `*rear = 2` `next = 3` `x = C`

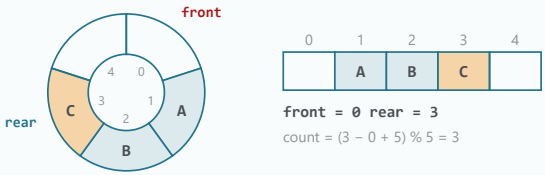
CALL STACK

`enqueue(front, &rear, C)`

`main()`

OUTPUT

Code Trace: A Safe Circular Queue



rear = 3, queue[3] = C.

Step 7 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

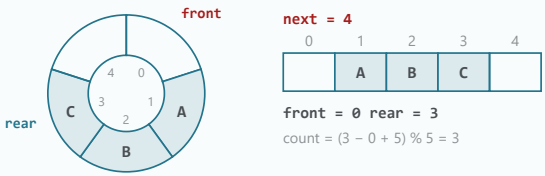
front = 0 | *rear = 3 | next = 3 | x = C

CALL STACK

enqueue(front, &rear, C)
main()

OUTPUT

Code Trace: A Safe Circular Queue



next = (3 + 1) % 5 = **4**; 4 ≠ front, there is room.

Step 8 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 0` `*rear = 3` `next = 4` `x = D`

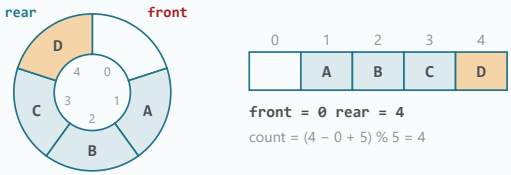
CALL STACK

`enqueue(front, &rear, D)`

`main()`

OUTPUT

Code Trace: A Safe Circular Queue



rear = 4, queue[4] = D.

Step 9 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

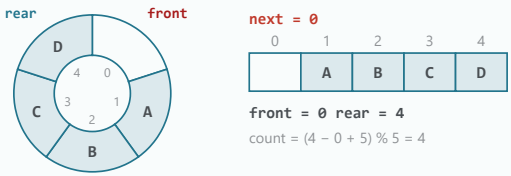
front = 0 | *rear = 4 | next = 4 | x = D

CALL STACK

enqueue(front, &rear, D)
main()

OUTPUT

Code Trace: A Safe Circular Queue



next = (4 + 1) % 5 = 0 == front → **full**, returns false. `rear` **stays** 4. 4 items in 5 cells.

Step 10 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 0` `*rear = 4` `next = 0` `x = E`

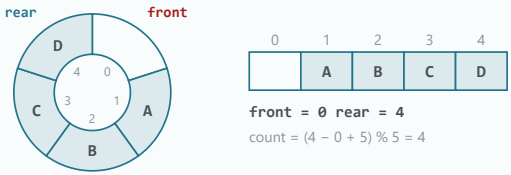
CALL STACK

`enqueue(front, &rear, E)`
`main()`

OUTPUT

```
E: full
```

Code Trace: A Safe Circular Queue



front == rear (0 == 4)? No.

Step 11 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
▶    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`*front = 0` `rear = 4`

CALL STACK

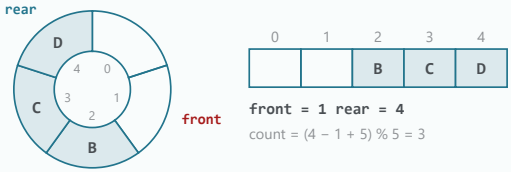
`dequeue(&front, rear, &x)`

`main()`

OUTPUT

```
E: full
```

Code Trace: A Safe Circular Queue



front = 1, *x = queue[1] = A.

Step 12 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
▶    *front = (*front + 1) % MAX_QUEUE_SIZE;
▶    *x = queue[*front];  return true;
11 }
```

VARIABLES

*front = 1 | rear = 4 | *x = A

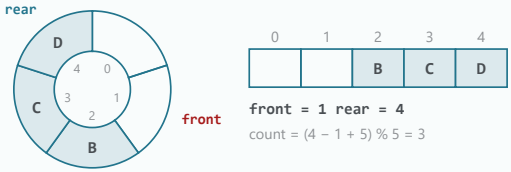
CALL STACK

dequeue(&front, rear, &x)
main()

OUTPUT

```
E: full  dequeue: A
```

Code Trace: A Safe Circular Queue



front == rear (1 == 4)? No.

Step 13 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
▶    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`*front = 1` `rear = 4`

CALL STACK

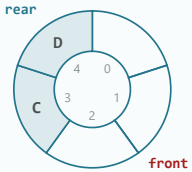
`dequeue(&front, rear, &x)`

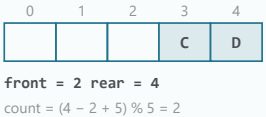
`main()`

OUTPUT

```
E: full  dequeue: A
```

Code Trace: A Safe Circular Queue





front = 2, ***x = queue[2] = B**.

Step 14 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
▶    *front = (*front + 1) % MAX_QUEUE_SIZE;
▶    *x = queue[*front];  return true;
11 }
```

VARIABLES

`*front = 2` `rear = 4` `*x = B`

CALL STACK

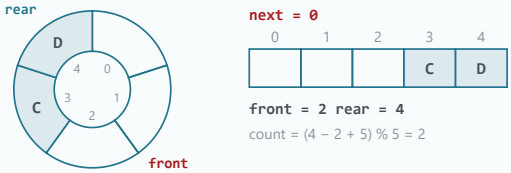
`dequeue(&front, rear, &x)`

`main()`

OUTPUT

```
E: full  dequeue: A  dequeue: B
```

Code Trace: A Safe Circular Queue



next = (4 + 1) % 5 = **0** (wrap!); 0 ≠ front, there is room.

Step 15 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 2` `*rear = 4` `next = 0` `x = E`

CALL STACK

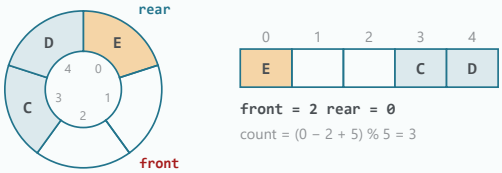
`enqueue(front, &rear, E)`

`main()`

OUTPUT

```
E: full  dequeue: A  dequeue: B
```

Code Trace: A Safe Circular Queue



rear = 0, queue[0] = E.

Step 16 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

front = 2 | *rear = 0 | next = 0 | x = E

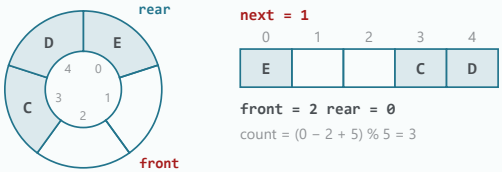
CALL STACK

enqueue(front, &rear, E)
main()

OUTPUT

```
E: full  dequeue: A  dequeue: B
```

Code Trace: A Safe Circular Queue



next = (0 + 1) % 5 = **1**; 1 ≠ front, there is room.

Step 17 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;    /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;   /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 2` `*rear = 0` `next = 1` `x = F`

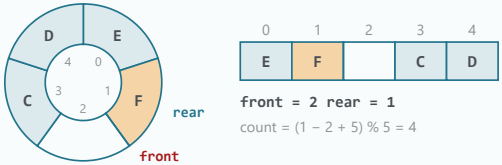
CALL STACK

`enqueue(front, &rear, F)`
`main()`

OUTPUT

```
E: full  dequeue: A  dequeue: B
```

Code Trace: A Safe Circular Queue



rear = 1, queue[1] = F.

Step 18 / 19

```
1  bool enqueue(int front, int *rear, element x) {
2    int next = (*rear + 1) % MAX_QUEUE_SIZE;
3    if (next == front) return false;     /* full */
▶    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

front = **2** | *rear = **1** | next = **1** | x = **F**

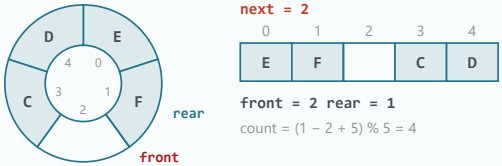
CALL STACK

- enqueue(front, &rear, F)
- main()

OUTPUT

```
E: full  dequeue: A  dequeue: B
```

Code Trace: A Safe Circular Queue



next = (1 + 1) % 5 = 2 == front → **full**, returns false. `rear` **stays** 1. 4 items in 5 cells.

Step 19 / 19

```
1  bool enqueue(int front, int *rear, element x) {
▶    int next = (*rear + 1) % MAX_QUEUE_SIZE;
▶    if (next == front) return false;     /* full */
4    *rear = next;  queue[next] = x;
5    return true;
6  }
7  bool dequeue(int *front, int rear, element *x) {
8    if (*front == rear) return false;    /* empty */
9    *front = (*front + 1) % MAX_QUEUE_SIZE;
10   *x = queue[*front];  return true;
11 }
```

VARIABLES

`front = 2` `*rear = 1` `next = 2` `x = G`

CALL STACK

`enqueue(front, &rear, G)`

`main()`

OUTPUT

```
E: full  dequeue: A  dequeue: B
G: full
```

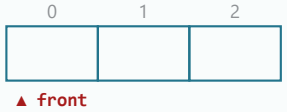
Telling Full from Empty: Three Methods

Method	Empty	Full	Capacity	Cost
1. Leave one cell empty	`front == rear`	`(rear+1)%MAX == front`	MAX − 1	One wasted cell
2. Keep a count	`count == 0`	`count == MAX`	MAX	Update the count on every operation
3. Keep a flag	`front == rear` and last op was dequeue	`front == rear` and last op was enqueue	MAX	A flag on every operation

The root of the problem: with only `front` and `rear` we want to distinguish MAX + 1 fill levels (0, 1, …, MAX items), but the difference of two indices takes only MAX distinct values. Either give up one state (method 1) or keep extra information (methods 2 and 3).

The counted version also gives the size in O(1) and doesn't store `rear` at all: `rear = (front + count − 1) % CAP`.

Code Trace: A Circular Queue with a Count



CAP = 3, front = 0, count = 0. Enqueue 1, 2, 3, 4; dequeue; enqueue 4; dequeue.

Step 1 / 10

```
1  bool enqueue(Queue *q, int x) {
2    if (q->count == CAP) return false;       /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

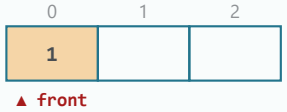
`q.front = 0` `q.count = 0`

CALL STACK

`main()`

OUTPUT

Code Trace: A Circular Queue with a Count



There is room; pos = (front + count) % 3 = **0**, data[0] = 1; count = 1.

Step 2 / 10

```
1  bool enqueue(Queue *q, int x) {
▶    if (q->count == CAP) return false;       /* full */
▶    q->data[(q->front + q->count) % CAP] = x;
▶    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

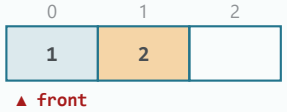
`q→front = 0` `q→count = 1` `x = 1`

CALL STACK

`enqueue(&q, 1)`
`main()`

OUTPUT

Code Trace: A Circular Queue with a Count



There is room; pos = (front + count) % 3 = **1**, data[1] = 2; count = 2.

Step 3 / 10

```
1  bool enqueue(Queue *q, int x) {
▶    if (q->count == CAP) return false;       /* full */
▶    q->data[(q->front + q->count) % CAP] = x;
▶    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

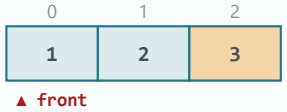
`q→front = 0` `q→count = 2` `x = 2`

CALL STACK

`enqueue(&q, 2)`
`main()`

OUTPUT

Code Trace: A Circular Queue with a Count



There is room; pos = (front + count) % 3 = **2**, data[2] = 3; count = 3.

Step 4 / 10

```
1  bool enqueue(Queue *q, int x) {
▶    if (q->count == CAP) return false;       /* full */
▶    q->data[(q->front + q->count) % CAP] = x;
▶    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

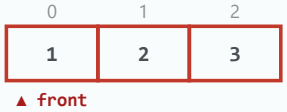
`q→front = 0` `q→count = 3` `x = 3`

CALL STACK

`enqueue(&q, 3)`
`main()`

OUTPUT

Code Trace: A Circular Queue with a Count



count == CAP → full. **All three** cells are in use: no cell is wasted.

Step 5 / 10

```
1  bool enqueue(Queue *q, int x) {
▶    if (q->count == CAP) return false;        /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;          /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

`q->front = 0` `q->count = 3` `x = 4`

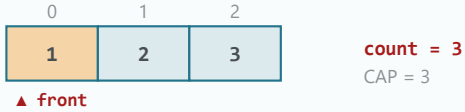
CALL STACK

`enqueue(&q, 4)`
`main()`

OUTPUT

```
enqueue 4: full
```

Code Trace: A Circular Queue with a Count



Not empty; ***x = data[0] = 1**.

Step 6 / 10

```
1  bool enqueue(Queue *q, int x) {
2    if (q->count == CAP) return false;        /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
▶    if (q->count == 0) return false;          /* empty */
▶    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

`q→front = 0` `q→count = 3` `*x = 1`

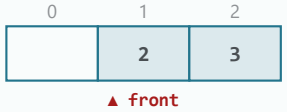
CALL STACK

`dequeue(&q, &x)`
`main()`

OUTPUT

```
enqueue 4: full
```

Code Trace: A Circular Queue with a Count



count = 2
CAP = 3

front = 1, count = 2.

Step 7 / 10

```
1  bool enqueue(Queue *q, int x) {
2    if (q->count == CAP) return false;       /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
▶    q->front = (q->front + 1) % CAP;
▶    q->count--;
12   return true;
13 }
```

VARIABLES

q→front = 1 | q→count = 2 | *x = 1

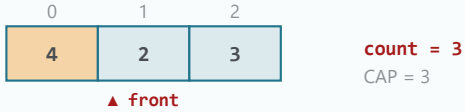
CALL STACK

dequeue(&q, &x)
main()

OUTPUT

```
enqueue 4: full  dequeue: 1
```

Code Trace: A Circular Queue with a Count



There is room; pos = (front + count) % 3 = **0** (wrap), data[0] = 4; count = 3.

Step 8 / 10

```
1  bool enqueue(Queue *q, int x) {
▶    if (q->count == CAP) return false;       /* full */
▶    q->data[(q->front + q->count) % CAP] = x;
▶    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

`q→front = 1` `q→count = 3` `x = 4`

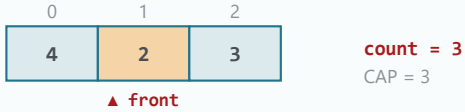
CALL STACK

`enqueue(&q, 4)`
`main()`

OUTPUT

```
enqueue 4: full  dequeue: 1
```

Code Trace: A Circular Queue with a Count



Not empty; ***x = data[1] = 2**.

Step 9 / 10

```
1  bool enqueue(Queue *q, int x) {
2    if (q->count == CAP) return false;       /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
▶    if (q->count == 0) return false;         /* empty */
▶    *x = q->data[q->front];
10   q->front = (q->front + 1) % CAP;
11   q->count--;
12   return true;
13 }
```

VARIABLES

`q→front = 1` `q→count = 3` `*x = 2`

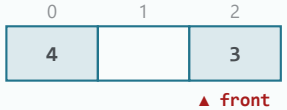
CALL STACK

`dequeue(&q, &x)`
`main()`

OUTPUT

```
enqueue 4: full  dequeue: 1
```

Code Trace: A Circular Queue with a Count



front = 2, count = 2.

Step 10 / 10

```
1  bool enqueue(Queue *q, int x) {
2    if (q->count == CAP) return false;       /* full */
3    q->data[(q->front + q->count) % CAP] = x;
4    q->count++;
5    return true;
6  }
7  bool dequeue(Queue *q, int *x) {
8    if (q->count == 0) return false;         /* empty */
9    *x = q->data[q->front];
▶    q->front = (q->front + 1) % CAP;
▶    q->count--;
12   return true;
13 }
```

VARIABLES

`q→front = 2` `q→count = 2` `*x = 2`

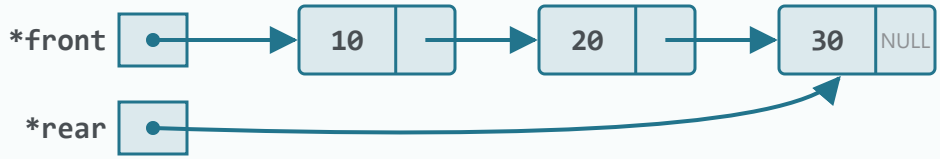
CALL STACK

- `dequeue(&q, &x)`
- `main()`

OUTPUT

```
enqueue 4: full  dequeue: 1
dequeue: 2
```

A Queue as a Linked List

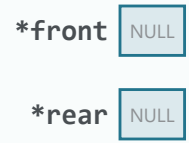


front points to the first node, rear to the last; the last node's link is NULL.

- Insert **at the rear**: `rear→link = new_node; rear = new_node;` → O(1).
- Delete **at the front**: `front = front→link;` → O(1).
- Not the other way round (insert at the front, delete at the rear): in a singly linked list, deleting the last node means finding the one before it, which is O(n).

INSTRUCTOR "– What have I been teaching for two lectures? I put the course aside and taught C, taught pointers. Pointers everyone supposedly knows so well, I taught them too. – I know pointers very well, but when the professor says something, I still don't get it. – That can happen."

Code Trace: Linked `enqueue`



Empty queue: **front = rear = NULL**. Enqueue 10, 20, 30 in turn.

Step 1 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

VARIABLES

`front = NULL` `rear = NULL`

CALL STACK

`main()`

Code Trace: Linked `enqueue`



malloc: a new node is allocated.

Step 2 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = NULL` `new_node = node(?)`

CALL STACK

`enqueue(&front, &rear, 10)`

`main()`

Code Trace: Linked `enqueue`



data = 10, **link = NULL** (it will be the new last node).

Step 3 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
▶   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = NULL`

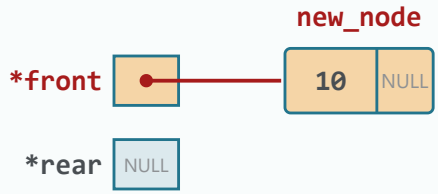
`new_node = node(10)`

CALL STACK

`enqueue(&front, &rear, 10)`

`main()`

Code Trace: Linked `enqueue`



***rear == NULL**: the queue was empty, so the new node is also the **front**: `*front = new_node`.

Step 4 / 14

```
1  bool enqueue(node **front, node **rear, int x) {
2    node *new_node = malloc(sizeof *new_node);
3    if (new_node == NULL) return false;
4    new_node->data = x;  new_node->link = NULL;
▶    if (*rear == NULL) *front = new_node; /* empty queue
   */
6    else (*rear)->link = new_node;     /* link at end */
7    *rear = new_node;
8    return true;
9  }
```

VARIABLES

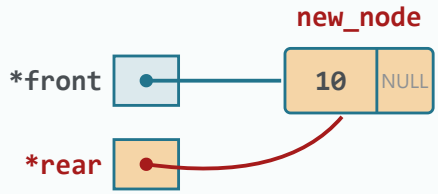
`*front = node(10)` `*rear = NULL` `new_node = node(10)`

CALL STACK

`enqueue(&front, &rear, 10)`

`main()`

Code Trace: Linked `enqueue`



***rear = new_node**: rear points to 10.

Step 5 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
    */
6   else (*rear)->link = new_node;     /* link at end */
▶   *rear = new_node;
8   return true;
9 }
```

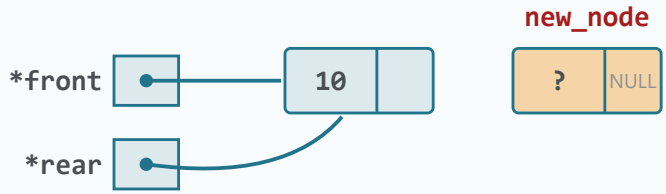
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(10)**
- `new_node` = **node(10)**

CALL STACK

- **`enqueue(&front, &rear, 10)`**
- `main()`

Code Trace: Linked `enqueue`



malloc: a new node is allocated.

Step 6 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

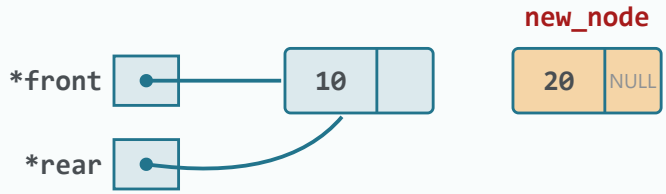
VARIABLES

- `*front = node(10)`
- `*rear = node(10)`
- `new_node = node(?)`

CALL STACK

- `enqueue(&front, &rear, 20)`
- `main()`

Code Trace: Linked `enqueue`



data = 20, **link = NULL** (it will be the new last node).

Step 7 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
▶   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

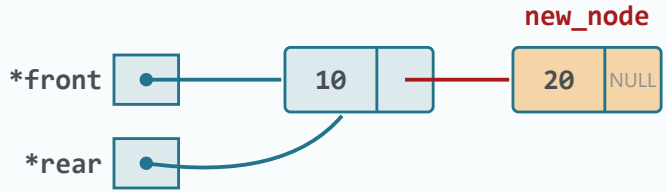
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(10)**
- `new_node` = **node(20)**

CALL STACK

- **`enqueue(&front, &rear, 20)`**
- `main()`

Code Trace: Linked `enqueue`



The queue is not empty: **(*rear)->link = new_node**, so 10 now points to 20.

Step 8 / 14

```
1  bool enqueue(node **front, node **rear, int x) {
2    node *new_node = malloc(sizeof *new_node);
3    if (new_node == NULL) return false;
4    new_node->data = x;  new_node->link = NULL;
5    if (*rear == NULL) *front = new_node; /* empty queue
   */
▶    else (*rear)->link = new_node;     /* link at end */
7    *rear = new_node;
8    return true;
9  }
```

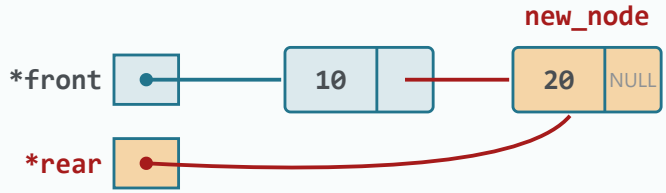
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(10)**
- `new_node` = **node(20)**
- `(*rear)→link` = **node(20)**

CALL STACK

- **`enqueue(&front, &rear, 20)`**
- `main()`

Code Trace: Linked `enqueue`



***rear = new_node**: rear points to 20.

Step 9 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
▶   *rear = new_node;
8   return true;
9 }
```

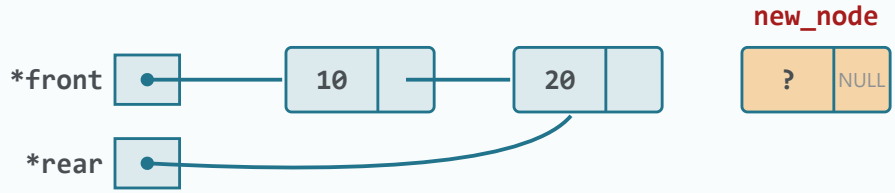
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(20)**
- `new_node` = **node(20)**

CALL STACK

- `enqueue(&front, &rear, 20)`
- `main()`

Code Trace: Linked `enqueue`



malloc: a new node is allocated.

Step 10 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
▶   node *new_node = malloc(sizeof *new_node);
▶   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

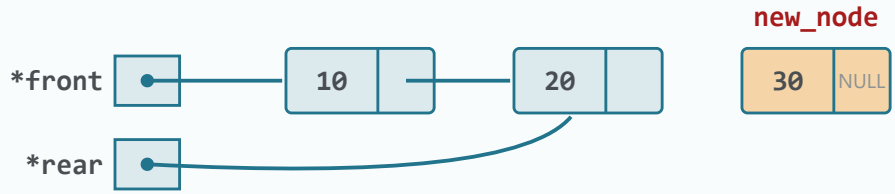
VARIABLES

- `*front = node(10)`
- `*rear = node(20)`
- `new_node = node(?)`

CALL STACK

- `enqueue(&front, &rear, 30)`
- `main()`

Code Trace: Linked `enqueue`



data = 30, **link = NULL** (it will be the new last node).

Step 11 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
▶   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

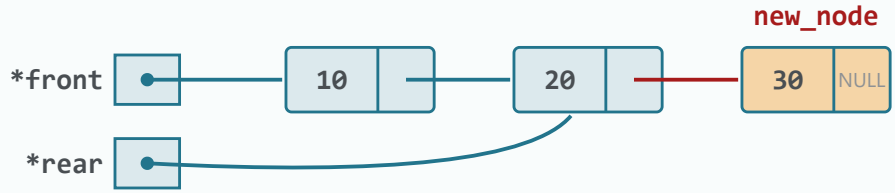
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(20)**
- `new_node` = **node(30)**

CALL STACK

- `enqueue(&front, &rear, 30)`
- `main()`

Code Trace: Linked `enqueue`



The queue is not empty: **(*rear)->link = new_node**, so 20 now points to 30.

Step 12 / 14

```
1  bool enqueue(node **front, node **rear, int x) {
2    node *new_node = malloc(sizeof *new_node);
3    if (new_node == NULL) return false;
4    new_node->data = x;  new_node->link = NULL;
5    if (*rear == NULL) *front = new_node; /* empty queue
   */
▶    else (*rear)->link = new_node;     /* link at end */
7    *rear = new_node;
8    return true;
9  }
```

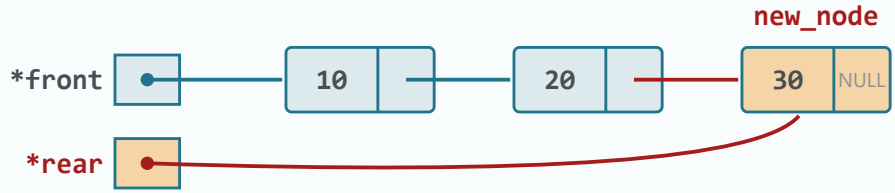
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(20)**
- `new_node` = **node(30)**
- `(*rear)→link` = **node(30)**

CALL STACK

- **`enqueue(&front, &rear, 30)`**
- `main()`

Code Trace: Linked `enqueue`



***rear = new_node**: rear points to 30.

Step 13 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
▶   *rear = new_node;
8   return true;
9 }
```

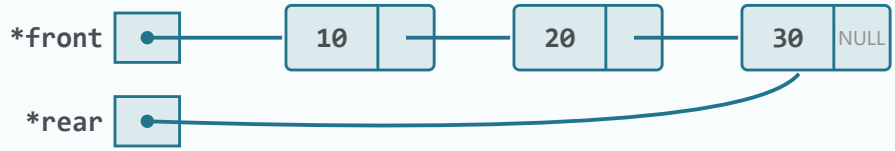
VARIABLES

- `*front` = **node(10)**
- `*rear` = **node(30)**
- `new_node` = **node(30)**

CALL STACK

- `enqueue(&front, &rear, 30)`
- `main()`

Code Trace: Linked `enqueue`



Queue: 10 → 20 → 30. Each enqueue does constant work: **O(1)**; without `rear`, finding the end would be O(n).

Step 14 / 14

```
1 bool enqueue(node **front, node **rear, int x) {
2   node *new_node = malloc(sizeof *new_node);
3   if (new_node == NULL) return false;
4   new_node->data = x;  new_node->link = NULL;
5   if (*rear == NULL) *front = new_node; /* empty queue
  */
6   else (*rear)->link = new_node;     /* link at end */
7   *rear = new_node;
8   return true;
9 }
```

VARIABLES

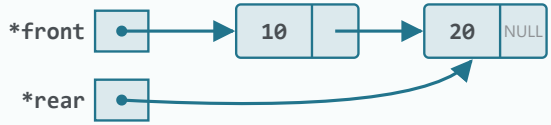
`front` = **node(10)**

`rear` = **node(30)**

CALL STACK

main()

Code Trace: Linked `dequeue`



Queue: 10 → 20. Call `dequeue` three times.

Step 1 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

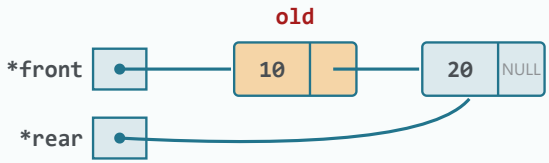
`front = node(10)` `rear = node(20)`

CALL STACK

`main()`

OUTPUT

Code Trace: Linked `dequeue`



old = *front.

Step 2 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
▶   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front` = **node(10)**

`*rear` = **node(20)** `old` = **node(10)**

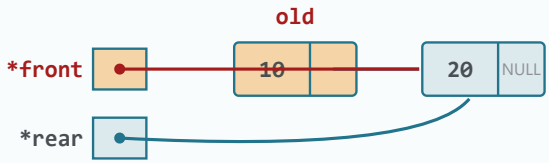
CALL STACK

`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

Code Trace: Linked `dequeue`



***x = 10**; ***front = old->link** → node(20).

Step 3 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;          /* empty */
▶   *x = old->data;
▶   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front` = **node(20)**

`*rear` = **node(20)** `old` = **node(10)**

`*x` = **10**

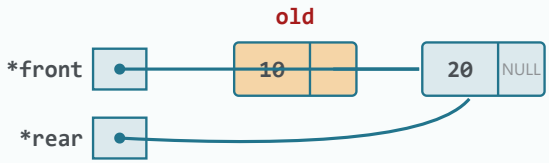
CALL STACK

`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

Code Trace: Linked `dequeue`



*front ≠ NULL: rear is unchanged.

Step 4 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
▶   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front` = **node(20)**

`*rear` = **node(20)** `old` = **node(10)**

`*x` = **10**

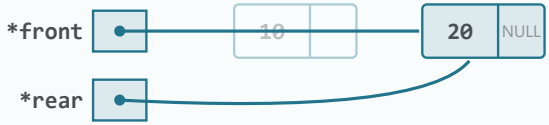
CALL STACK

`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

Code Trace: Linked `dequeue`



free(old); main prints 10.

Step 5 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
▶   free(old);
8   return true;
9 }
```

VARIABLES

`*front = node(20)`

`*rear = node(20)` `*x = 10`

CALL STACK

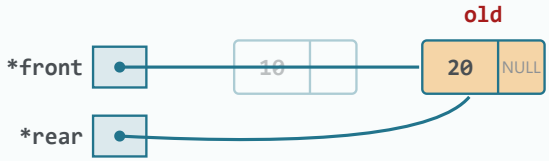
`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

```
10
```

Code Trace: Linked `dequeue`



old = *front.

Step 6 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
▶   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front` = **node(20)**

`*rear` = **node(20)** `old` = **node(20)**

CALL STACK

`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

```
10
```

Code Trace: Linked `dequeue`



***x = 20**; ***front = old->link** → NULL.

Step 7 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;          /* empty */
▶   *x = old->data;
▶   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = node(20)` `old = node(20)` `*x = 20`

CALL STACK

`dequeue(&front, &rear, &x)`
`main()`

OUTPUT

```
10
```

Code Trace: Linked `dequeue`



***front == NULL**: the last item is gone too; without ***rear = NULL**, rear would keep pointing to a node about to be freed (a dangling pointer)!

Step 8 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
▶   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = NULL` `old = node(20)` `*x = 20`

CALL STACK

`dequeue(&front, &rear, &x)`
`main()`

OUTPUT

```
10
```

Code Trace: Linked `dequeue`



free(old); main prints 20.

Step 9 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
2   node *old = *front;
3   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
▶   free(old);
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = NULL` `*x = 20`

CALL STACK

`dequeue(&front, &rear, &x)`

`main()`

OUTPUT

```
10 20
```

Code Trace: Linked `dequeue`



old == NULL → empty, returns false.

Step 10 / 10

```
1 bool dequeue(node **front, node **rear, int *x) {
▶   node *old = *front;
▶   if (old == NULL) return false;        /* empty */
4   *x = old->data;
5   *front = old->link;
6   if (*front == NULL) *rear = NULL; /* last one gone */
7   free(old);
8   return true;
9 }
```

VARIABLES

`*front = NULL` `*rear = NULL` `old = NULL`

CALL STACK

`dequeue(&front, &rear, &x)`
`main()`

OUTPUT

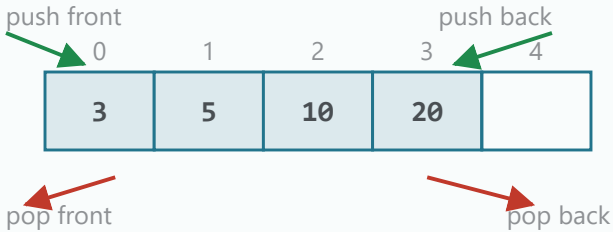
```
10 20
```

Comparing Queue Implementations

Criterion	Linear array	Circular array	Linked list
enqueue / dequeue	O(1), but “false full”	O(1)	O(1) (malloc/free)
Shifting needed?	Yes: O(n)	No	No
Capacity	MAX	MAX − 1 (or MAX with a count)	Up to available memory
Extra memory	None	None	One pointer per item
Special case	front/rear hit the end	Wrap-around and full/empty test	rear = NULL when emptied

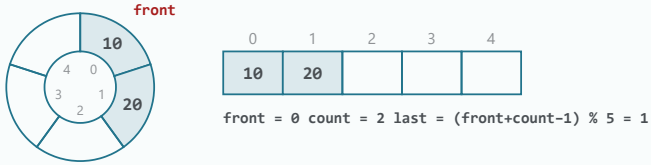
In practice the circular array (a fixed-capacity buffer) and the growable circular array are the most common; a linked queue is preferred when the items already live as nodes or the maximum size is unknown.

The Double-Ended Queue (Deque)



- Deque (*double-ended queue*): insertion and deletion **at both ends**, all in O(1).
- Use only one end and it is a **stack**; insert at one end and remove at the other and it is a **queue**.
- Restricted variants: input-restricted (insert at one end only) and output-restricted (delete at one end only).
- Uses: sliding-window maximum, work stealing, undo + redo history.

Code Trace: A Deque in a Circular Array



CAP = 5; `push_back` has stored 10 and 20. Now push 5 and 3 at the front, then pop the back twice.

Step 1 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

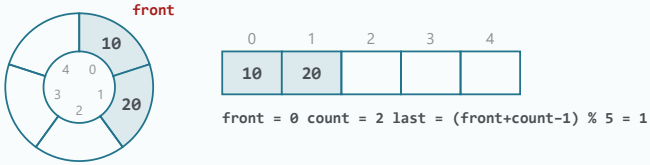
`d.front = 0` `d.count = 2`

CALL STACK

`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



count = 2 < 5, there is room.

Step 2 / 11

```
1  bool push_front(Deque *d, int x) {
▶    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

`d→front = 0` `d→count = 2` `x = 5`

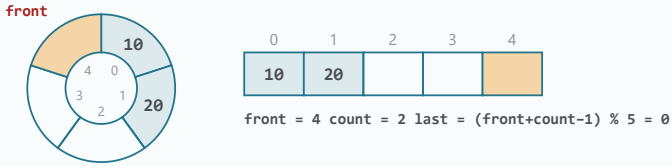
CALL STACK

`push_front(&d, 5)`

`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



front = (0 − 1 + 5) % 5 = 4: one step back from 0 is the **end** of the array.

Step 3 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
▶    d->front = (d->front - 1 + CAP) % CAP;  /* one back */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

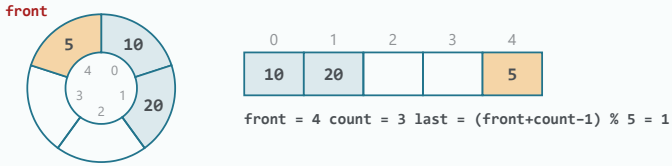
`d->front = 4` `d->count = 2` `x = 5`

CALL STACK

`push_front(&d, 5)`
`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



data[4] = 5; count = 3.

Step 4 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
▶    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

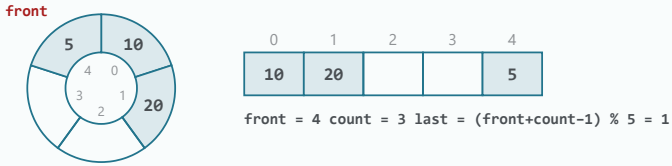
`d→front = 4` `d→count = 3` `x = 5`

CALL STACK

`push_front(&d, 5)`
`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



count = 3 < 5, there is room.

Step 5 / 11

```
1  bool push_front(Deque *d, int x) {
▶    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

`d→front = 4` `d→count = 3` `x = 3`

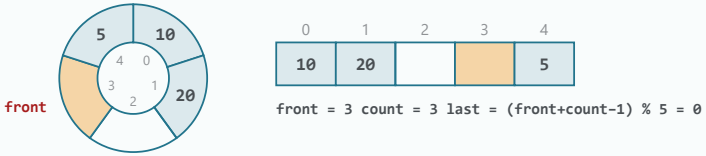
CALL STACK

`push_front(&d, 3)`

`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



front = (4 − 1 + 5) % 5 = 3.

Step 6 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
▶    d->front = (d->front - 1 + CAP) % CAP;  /* one back */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

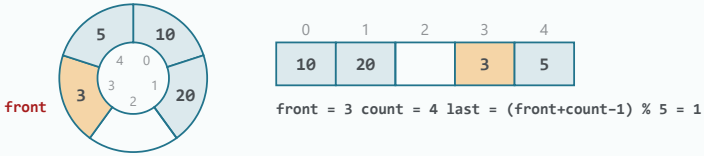
`d→front = 3` `d→count = 3` `x = 3`

CALL STACK

`push_front(&d, 3)`
`main()`

OUTPUT

Code Trace: A Deque in a Circular Array



data[3] = 3; count = 4.

Step 7 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
▶    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

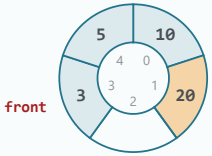
`d→front = 3` `d→count = 4` `x = 3`

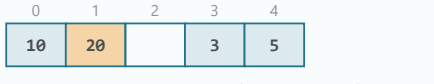
CALL STACK

`push_front(&d, 3)`
`main()`

OUTPUT

Code Trace: A Deque in a Circular Array





Last position = (3 + 4 − 1) % 5 = 1; ***x = 20**.

Step 8 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
▶    if (d->count == 0) return false;
▶    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

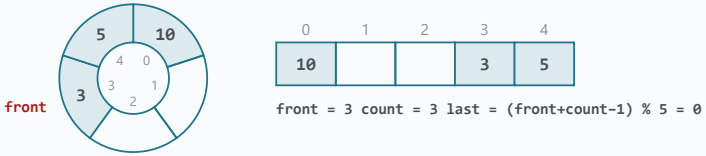
`d→front = 3` `d→count = 4` `*x = 20`

CALL STACK

- `pop_back(&d, &x)`
- `main()`

OUTPUT

Code Trace: A Deque in a Circular Array



count = 3. (`front` is unchanged.)

Step 9 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
▶    d->count--;  return true;
11 }
```

VARIABLES

`d→front = 3` `d→count = 3` `*x = 20`

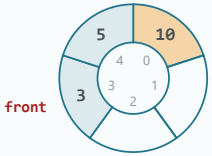
CALL STACK

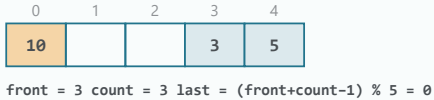
- `pop_back(&d, &x)`
- `main()`

OUTPUT

```
20
```

Code Trace: A Deque in a Circular Array





Last position = (3 + 3 − 1) % 5 = 0; ***x = 10**.

Step 10 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
▶    if (d->count == 0) return false;
▶    *x = d->data[(d->front + d->count - 1) % CAP];
10   d->count--;  return true;
11 }
```

VARIABLES

`d→front = 3` `d→count = 3` `*x = 10`

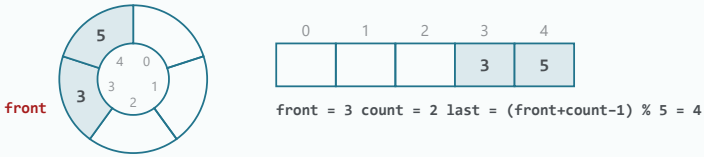
CALL STACK

- `pop_back(&d, &x)`
- `main()`

OUTPUT

```
20
```

Code Trace: A Deque in a Circular Array



count = 2. (`front` is unchanged.)

Step 11 / 11

```
1  bool push_front(Deque *d, int x) {
2    if (d->count == CAP) return false;
3    d->front = (d->front - 1 + CAP) % CAP;  /* one back
   */
4    d->data[d->front] = x;  d->count++;
5    return true;
6  }
7  bool pop_back(Deque *d, int *x) {
8    if (d->count == 0) return false;
9    *x = d->data[(d->front + d->count - 1) % CAP];
▶    d->count--;  return true;
11 }
```

VARIABLES

`d→front = 3` `d→count = 2` `*x = 10`

CALL STACK

`pop_back(&d, &x)`

`main()`

OUTPUT

```
20 10
```

Several Queues in One Array

Splitting into segments

As with multiple stacks, a one-dimensional `memory[]` array is split into segments, and each queue keeps its own `front[i]`, `rear[i]`, and `boundary[i]`.

Two queues

The stack trick of growing from the two ends doesn't work for queues: a queue moves at both of its ends and drifts within its segment.

Circular within a segment

Managing each segment as its own circular queue avoids shifting, but when a segment fills up, room still has to be made (O(m)).

Today

Usually each queue gets its own growable array or linked list.

Code Trace: Round-Robin Scheduling



Three processes need P0 = 5, P1 = 2, P2 = 4 units of work. Time slice Q = 2 (**round robin**).

Step 1 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);               /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

VARIABLES

`Q = 2` `time = 0`

OUTPUT

Code Trace: Round-Robin Scheduling



P0 is taken from the front and runs 2 units; t = 2.

Step 2 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶ while (dequeue(&q, &p)) {
▶   int run = left[p] < Q ? left[p] : Q;
▶   time += run;
▶   left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

VARIABLES

`p = 0` `run = 2` `time = 2` `left[0] = 3`

OUTPUT

Code Trace: Round-Robin Scheduling



P0 has 3 units left → goes back to the **rear**.

Step 3 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
▶    if (left[p] > 0)
▶      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

VARIABLES

`p = 0` `run = 2` `time = 2` `left[0] = 3`

OUTPUT

Code Trace: Round-Robin Scheduling



P1 is taken from the front and runs 2 units; t = 4.

Step 4 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶ while (dequeue(&q, &p)) {
▶   int run = left[p] < Q ? left[p] : Q;
▶   time += run;
▶   left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

VARIABLES

p = 1 run = 2 time = 4 left[1] = 0

OUTPUT

Code Trace: Round-Robin Scheduling



P1 is finished; it doesn't rejoin the queue.

Step 5 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
▶      printf("P%d done: t=%d\n", p, time);
10 }
```

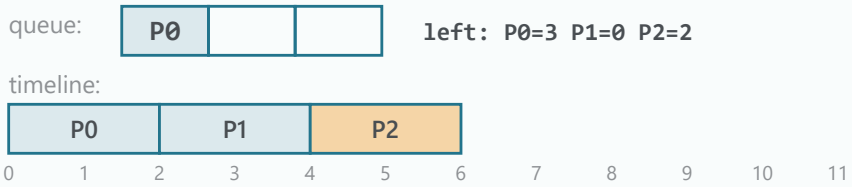
VARIABLES

`p = 1` `run = 2` `time = 4` `left[1] = 0`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P2 is taken from the front and runs 2 units; t = 6.

Step 6 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶ while (dequeue(&q, &p)) {
▶   int run = left[p] < Q ? left[p] : Q;
▶   time += run;
▶   left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

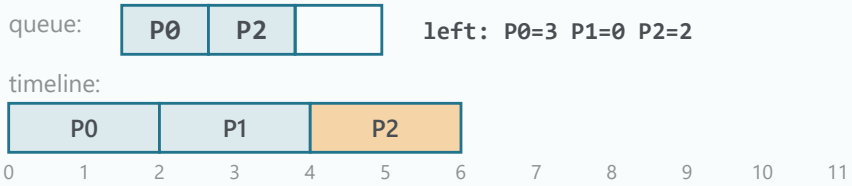
VARIABLES

`p = 2` `run = 2` `time = 6` `left[2] = 2`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P2 has 2 units left → goes back to the **rear**.

Step 7 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
▶    if (left[p] > 0)
▶      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

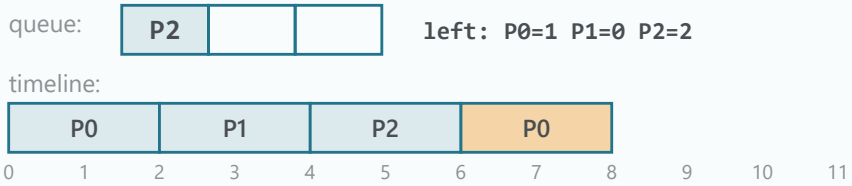
VARIABLES

`p = 2` `run = 2` `time = 6` `left[2] = 2`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P0 is taken from the front and runs 2 units; t = 8.

Step 8 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶  while (dequeue(&q, &p)) {
▶    int run = left[p] < Q ? left[p] : Q;
▶    time += run;
▶    left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

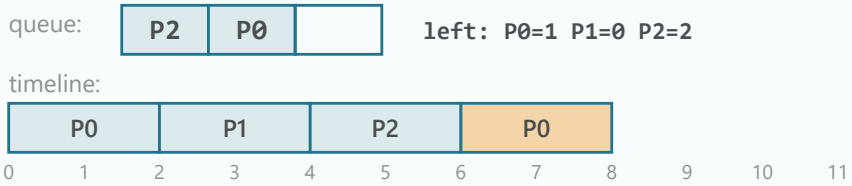
VARIABLES

`p = 0` `run = 2` `time = 8` `left[0] = 1`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P0 has 1 unit left → goes back to the **rear**.

Step 9 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
▶    if (left[p] > 0)
▶      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

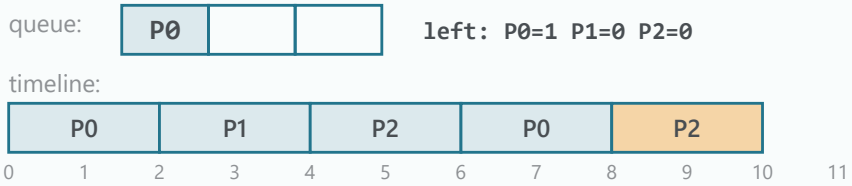
VARIABLES

`p = 0` `run = 2` `time = 8` `left[0] = 1`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P2 is taken from the front and runs 2 units; t = 10.

Step 10 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶ while (dequeue(&q, &p)) {
▶   int run = left[p] < Q ? left[p] : Q;
▶   time += run;
▶   left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

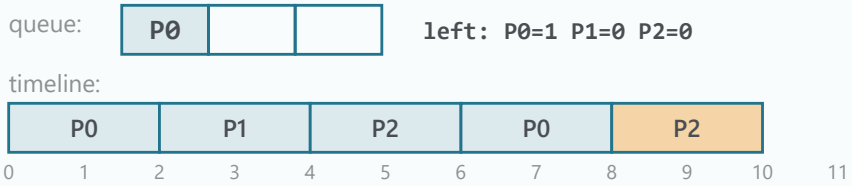
VARIABLES

`p = 2` `run = 2` `time = 10` `left[2] = 0`

OUTPUT

```
P1 done: t=4
```

Code Trace: Round-Robin Scheduling



P2 is finished; it doesn't rejoin the queue.

Step 11 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
▶      printf("P%d done: t=%d\n", p, time);
10 }
```

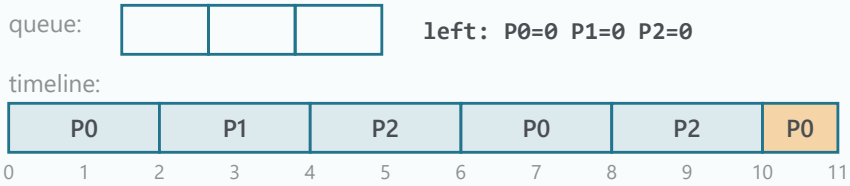
VARIABLES

`p = 2` `run = 2` `time = 10` `left[2] = 0`

OUTPUT

```
P1 done: t=4
P2 done: t=10
```

Code Trace: Round-Robin Scheduling



P0 is taken from the front and runs 1 unit; t = 11.

Step 12 / 13

```
1  /* q: queue of process ids; Q: time slice */
▶ while (dequeue(&q, &p)) {
▶   int run = left[p] < Q ? left[p] : Q;
▶   time += run;
▶   left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
9      printf("P%d done: t=%d\n", p, time);
10 }
```

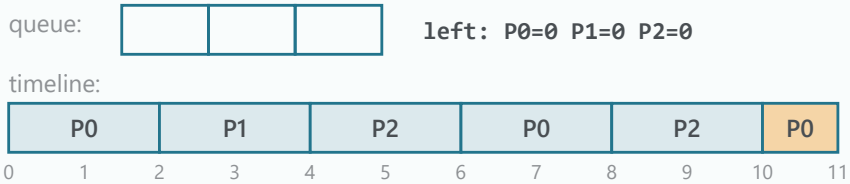
VARIABLES

`p = 0` `run = 1` `time = 11` `left[0] = 0`

OUTPUT

```
P1 done: t=4
P2 done: t=10
```

Code Trace: Round-Robin Scheduling



P0 is finished; it doesn't rejoin the queue.

Step 13 / 13

```
1  /* q: queue of process ids; Q: time slice */
2  while (dequeue(&q, &p)) {
3    int run = left[p] < Q ? left[p] : Q;
4    time += run;
5    left[p] -= run;
6    if (left[p] > 0)
7      enqueue(&q, p);              /* back of line */
8    else
▶      printf("P%d done: t=%d\n", p, time);
10 }
```

VARIABLES

`p = 0` `run = 1` `time = 11` `left[0] = 0`

OUTPUT

```
P1 done: t=4
P2 done: t=10
P0 done: t=11
```

Code Trace: A Queue Built from Two Stacks



The queue is made of two stacks. Enqueue 1, 2, 3; dequeue; enqueue 4; dequeue three times.

Step 1 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))      /* out is empty */
7      while (pop(&q->in, &t))      /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = []` `out = []`

CALL STACK

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



enqueue(1): **push** onto the in stack, O(1).

Step 2 / 15

```
1  void enqueue(Queue2 *q, int x) {
▶    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1]` `out = []`

CALL STACK

`enqueue(&q, 1)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



enqueue(2): **push** onto the in stack, O(1).

Step 3 / 15

```
1  void enqueue(Queue2 *q, int x) {
▶    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))      /* out is empty */
7      while (pop(&q->in, &t))      /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1, 2]` `out = []`

CALL STACK

`enqueue(&q, 2)`
`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



enqueue(3): **push** onto the in stack, O(1).

Step 4 / 15

```
1  void enqueue(Queue2 *q, int x) {
▶    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))      /* out is empty */
7      while (pop(&q->in, &t))      /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1, 2, 3]` `out = []`

CALL STACK

`enqueue(&q, 3)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



The out stack is empty → transfer everything from the in stack.

Step 5 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
▶    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1, 2, 3]` `out = []`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



Pop 3 from in, push it onto out.

Step 6 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
▶      while (pop(&q->in, &t))       /* move them all */
▶        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1, 2]` `out = [3]`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



Pop 2 from in, push it onto out.

Step 7 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
▶      while (pop(&q->in, &t))       /* move them all */
▶        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [1]` `out = [3, 2]`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



Pop 1 from in, push it onto out.

Step 8 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
▶      while (pop(&q->in, &t))       /* move them all */
▶        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = []` `out = [3, 2, 1]`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

Code Trace: A Queue Built from Two Stacks



The top of the out stack is **1**, the oldest item. FIFO achieved.

Step 9 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
▶    return pop(&q->out, x);
10 }
```

VARIABLES

`in = []` `out = [3, 2]` `*x = 1`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

```
1
```

Code Trace: A Queue Built from Two Stacks



enqueue(4): **push** onto the in stack, O(1).

Step 10 / 15

```
1  void enqueue(Queue2 *q, int x) {
▶    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))      /* out is empty */
7      while (pop(&q->in, &t))      /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [4]` `out = [3, 2]`

CALL STACK

`enqueue(&q, 4)`
`main()`

OUTPUT

```
1
```

Code Trace: A Queue Built from Two Stacks



The top of the out stack is **2**, the oldest item. FIFO achieved.

Step 11 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
▶    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [4]` `out = [3]` `*x = 2`

CALL STACK

`dequeue(&q, &x)`
`main()`

OUTPUT

```
1 2
```

Code Trace: A Queue Built from Two Stacks



The top of the out stack is **3**, the oldest item. FIFO achieved.

Step 12 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
▶    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [4]` `out = []` `*x = 3`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

```
1 2 3
```

Code Trace: A Queue Built from Two Stacks



The out stack is empty → transfer everything from the in stack.

Step 13 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
▶    if (stack_empty(&q->out))       /* out is empty */
7      while (pop(&q->in, &t))       /* move them all */
8        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = [4]` `out = []`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

```
1 2 3
```

Code Trace: A Queue Built from Two Stacks



Pop 4 from in, push it onto out.

Step 14 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))       /* out is empty */
▶      while (pop(&q->in, &t))       /* move them all */
▶        push(&q->out, t);
9    return pop(&q->out, x);
10 }
```

VARIABLES

`in = []` `out = [4]`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

```
1 2 3
```

Code Trace: A Queue Built from Two Stacks



The top of the out stack is **4**, the oldest item. FIFO achieved. Each item is transferred at most once → **amortized O(1)**.

Step 15 / 15

```
1  void enqueue(Queue2 *q, int x) {
2    push(&q->in, x);
3  }
4  bool dequeue(Queue2 *q, int *x) {
5    int t;
6    if (stack_empty(&q->out))      /* out is empty */
7      while (pop(&q->in, &t))      /* move them all */
8        push(&q->out, t);
▶    return pop(&q->out, x);
10 }
```

VARIABLES

`in = []` `out = []` `*x = 4`

CALL STACK

`dequeue(&q, &x)`

`main()`

OUTPUT

```
1 2 3 4
```

Which One Fits?

Problem	Structure
Documents sent to a printer	▶ show
Ctrl+Z in a text editor	▶ show
Printing a tree level by level	▶ show
Checking nested parentheses	▶ show
Function calls	▶ show
Sickest patient first in the ER	▶ show

INSTRUCTOR "You know something called a queue, and the word queue appears here too. But in the queue we know, you add at the back and take from the front. This one is different…"

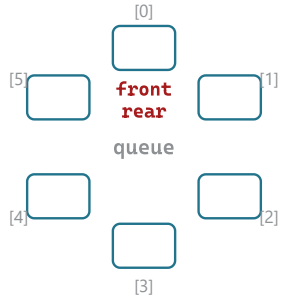
Lab: Circular Queue

Wrapping around

Input (number = enqueue, dequeue) `10, 20, 30, dequeue, dequeue, 40, 50, 60, 70, dequeue` Apply

CODE

```
 1 #define MAX_QUEUE_SIZE 6
 2 bool enqueue(int front, int *rear,
 3              element x) {
 4   int next = (*rear + 1) % MAX_QUEUE_SIZE;
 5   if (next == front) return false;  /* full …
 6   *rear = next;  queue[next] = x;
 7   return true;
 8 }
 9 bool dequeue(int *front, int rear,
10              element *x) {
11   if (*front == rear) return false; /* empty…
12   *front = (*front + 1) % MAX_QUEUE_SIZE;
13   *x = queue[*front];  return true;
14 }
```



`MAX_QUEUE_SIZE 6` `front 0` `rear 0`

OUTPUT (none yet)

The queue starts empty: `front = rear = 0`. The array has 6 slots; the circular queue holds at most 5 items, because one slot is kept empty to tell a full queue from an empty one.

SUMMARY

Complexity Summary

Operation	Linear array	Circular array	Linked	Deque (circular)
enqueue	O(1)*	O(1)	O(1)	O(1) at both ends
dequeue	O(1)	O(1)	O(1)	O(1) at both ends
front / empty / full	O(1)	O(1)	O(1)	O(1)
item count	rear − front	(rear−front+MAX)%MAX	O(n) or a count	count
memory	MAX	MAX	n · (item + pointer)	CAP

* Enqueue in a linear queue is O(1), but O(n) if it shifts on a “false full.” The round-robin simulation and level-order traversal perform O(n) queue operations for n items.

SUMMARY

Common Mistakes

1. Taking `rear == MAX-1` in a linear queue as truly full.
2. Forgetting `% MAX` in a circular queue and writing past the array.
3. Advancing `rear` first and not rolling it back when full: the queue then looks empty.
4. Writing `(i - 1) % MAX` to step back: in C the result can be negative.
5. Not setting `rear = NULL` when the last item leaves a linked queue.
6. Forgetting to update `front` on the first insert into an empty queue.

SUMMARY

Exercises

1. MAX = 6, front = 4, rear = 1: how many items are there, and at which indices?
2. Write `enqueue` and `dequeue` for a circular queue that uses the flag method.
3. Build a queue from two stacks. What is the amortized cost of `dequeue`?
4. Implement a queue as a circular linked list with only a **rear** pointer.
5. Run the round-robin example with Q = 3: when does each process finish?

▶ **Hints / answers**

SUMMARY

Chapter Summary

FIFO

Insert at the rear, remove at the front; first in, first out.

Linear queue

Simple, but once rear hits the end the free space at the front is lost; shifting is expensive.

Circular queue

Wrap with `% MAX`; empty: front == rear, full: (rear+1)%MAX == front.

Full vs. empty

Leave one cell empty, or keep a count or a flag.

Linked queue

front and rear pointers; rear = NULL when the last item leaves.

Applications

Deques, job scheduling, buffers, a queue from two stacks, BFS.

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.3 and 4.3.
- T. H. Cormen et al., *Introduction to Algorithms*, 3rd ed., Section 10.1 (queues, deque exercises).
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Chapter 4.

← Chapter 6: Stacks

Course page

Chapter 8: Stack and Queue Applications

The End

Chapter 7: Queues

Next chapter: Chapter 8: Stack and Queue Applications · Course page