UNIT 4 · LINEAR STRUCTURES

13. Queue ADT and Implementation

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

Agenda

Q

The Queue Concept & FIFO

First In, First Out — the rule that defines every queue

A

The Queue ADT

enqueue, dequeue, peek, isEmpty

[]

Array-Based Implementation

Two indices, and the wasted-space limitation it exposes

!

Full / Empty Edge Cases

Why both checks are mandatory, not optional

->

Linked-List Implementation

Sidesteps wasted space entirely

O

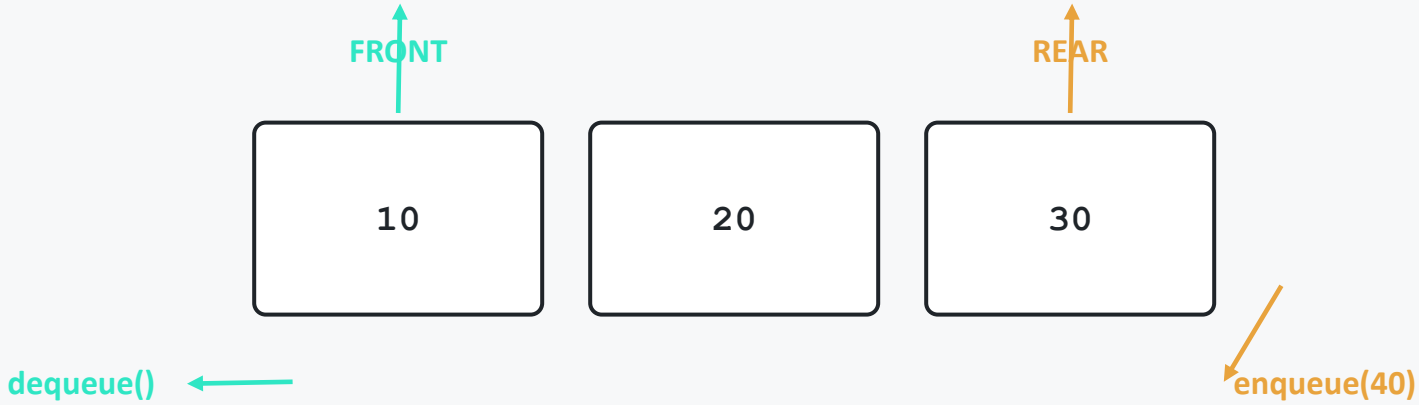
Comparison & Complexity

Array vs. linked list, and O(1) for every operation

THE QUEUE CONCEPT

FIFO: First In, First Out

New elements join at the REAR. Elements leave from the FRONT. Whichever element has waited longest leaves next.



O(1)

enqueue / dequeue time, both ends touched independently

2

positions tracked: frontIndex (read) and rearIndex (write)

THE INTERFACE

The Queue ADT

`enqueue(value)`	Add value at the rear of the queue
`dequeue()`	Remove and return the value at the front
`peek() / front()`	Return the front value without removing it
`isEmpty()`	Report whether the queue has any elements

FIRST ATTEMPT

Array-Based Implementation

Two indices track the two ends independently: frontIndex reads, rearIndex writes.

array_queue.cpp

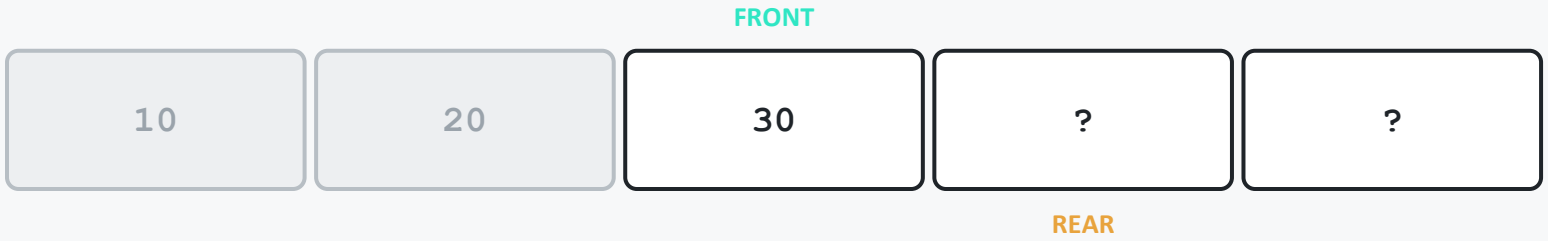
```
class ArrayQueue {
    int data[CAPACITY];
    int frontIndex, rearIndex;   // rearIndex = next free slot

    bool isFull() const { return rearIndex == CAPACITY; }

    void enqueue(int value) {
        if (isFull()) throw overflow_error("Queue is full");
        data[rearIndex++] = value;
    }

    int dequeue() {
        if (isEmpty()) throw underflow_error("Queue is empty");
        return data[frontIndex++];
    }
};
```

WATCH THE INDICES MOVE

Tracing enqueue / dequeue Against the Array

Step 1: enqueue(10), enqueue(20), enqueue(30)

Step 2: dequeue() -> 10, dequeue() -> 20 (slots 0,1 now stale)



** stale — still physically in the array, but logically dequeued and no longer reachable*

THE LIMITATION THIS EXPOSES

A Naive Array Queue Wastes Space

rearIndex only ever climbs toward CAPACITY. Once frontIndex moves past a slot, that slot is abandoned forever — even though it's physically free. The queue can report "full" while most of the array sits empty.

2 of 5

slots permanently wasted after just two dequeues (CAPACITY = 5)

isFull()

reports true once rearIndex == CAPACITY, regardless of real content

Fix -> L14

the circular queue reuses freed slots via modulo wraparound

TWO CHECKS, EVERY TIME

Queue-Full and Queue-Empty Edge Cases

the two guards

```
bool isEmpty() const {
    return frontIndex == rearIndex;
}
bool isFull() const {
    return rearIndex == CAPACITY;
}
```

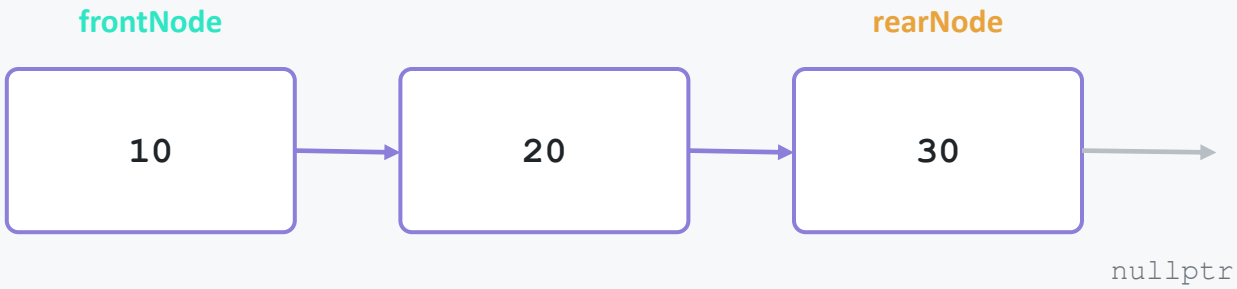
isEmpty() compares frontIndex == rearIndex (nothing between them).

isFull() compares rearIndex == CAPACITY (no room to grow).

They check genuinely different conditions — skipping either lets enqueue write past the array, or dequeue/peek read uninitialized memory.

CAPACITY = 3 trace: dequeue()/peek() on empty throw underflow_error → enqueue(1,2,3) fills it → enqueue(4) throws overflow_error → draining back to empty, dequeue() throws underflow_error again.

Linked-List Implementation



linked_queue.cpp

```
void enqueue(int value) {
    Node* newNode = new Node(value);
    if (isEmpty()) { frontNode = rearNode = newNode; return; }
    rearNode->next = newNode;  rearNode = newNode;
}
int dequeue() {
    Node* oldFront = frontNode;
    int value = oldFront->data;
    frontNode = frontNode->next;
    if (frontNode == nullptr) rearNode = nullptr;
    delete oldFront;  return value;
}
```

SAME ADT, DIFFERENT TRADE-OFFS

Array-Based vs. Linked-List-Based

Array-Based (ArrayQueue)

- Contiguous memory — cache-friendly
- Fixed CAPACITY, set at compile time
- frontIndex only moves forward — wastes slots once dequeued

Linked-List-Based (LinkedQueue)

- Scattered nodes — worse cache locality
- Grows one node at a time, no fixed limit
- Every dequeued node is genuinely freed — nothing wasted

Maximum size	Fixed (CAPACITY)	Limited by memory
Wasted space after dequeues	Yes, until Lecture 14	No — nodes freed
Good fit when...	Max size known ahead of time	Size unpredictable

THE BOTTOM LINE

Complexity of Queue Operations

O(1)

enqueue() — array-based and linked-list-based alike

O(1)

dequeue() — array-based and linked-list-based alike

O(1)

peek() — array-based and linked-list-based alike

Both implementations achieve O(1) for every core operation. The real difference — exactly as with stacks — is about capacity and wasted space, not raw speed. Lecture 14's circular queue closes that gap for the array-based version entirely.

KEY TAKEAWAYS

Key Takeaways

- A queue enforces FIFO — elements are added at the rear and removed from the front.
- enqueue, dequeue, and peek are all O(1), in both array-based and linked-list-based implementations.
- An array-based queue needs two independent indices — frontIndex and rearIndex — because the two ends drift apart independently.
- isFull() and isEmpty() check genuinely different conditions, and both must be checked before every array-touching operation.
- A naive array-based queue wastes space permanently; a linked-list-based queue avoids this at the cost of cache locality and pointer overhead.