Queues

First-in and First-served

Objectives

- Examine queue processing
- Define a queue abstract data type
- Demonstrate how a queue can be used to solve problems
- Examine various queue implementations
- Compare queue implementations

Queue

- **Queue**: a collection whose elements are added at one end (the ***rear*** or ***tail*** of the queue) and removed from the other end (the ***front*** or ***head*** of the queue)
- A queue is a ***FIFO*** (first in, first out) data structure
- Any waiting line is a queue:
  - The check-out line at a grocery store
  - The cars at a stop light
  - An assembly line

Uses of Queues in Computing

- For any kind of problem involving FIFO data
- Printer queue (e.g. printer in MC 235)
- Keyboard input buffer
- GUI event queue (click on buttons, menu items)
- In ***simulation studies***, where the goal is to reduce waiting times:
  - Optimize the flow of traffic at a traffic light

Queue Operations

- ***enqueue*** : add an element to the tail of a queue
- ***dequeue*** : remove an element from the head of a queue
- ***first*** : examine the element at the head of the queue ("peek")
- Other useful operations (e.g. is the queue empty)
- It is ***not*** *a queue operation if one is* to access the elements in the middle of the queue.

Operations on a Queue

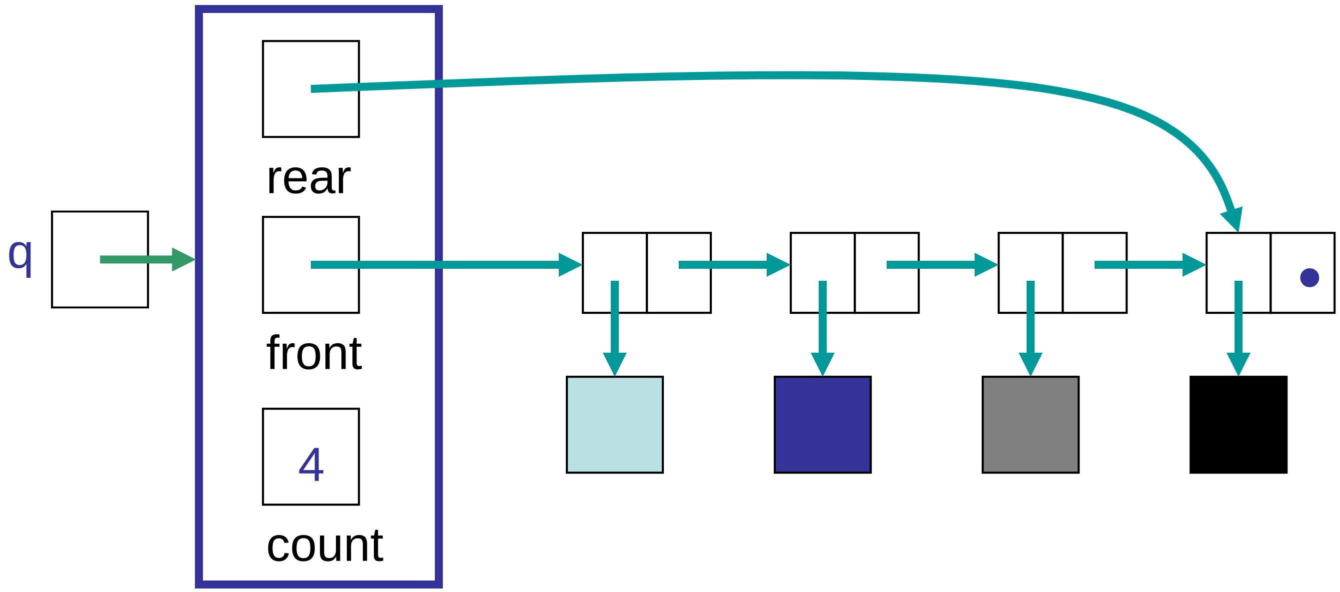
Operation	Description
dequeue	Removes an element from the front of the queue
enqueue	Adds an element to the rear of the queue
first	Examines the element at the front of the queue
isEmpty	Determines whether the queue is empty
size	Determines the number of elements in the queue
toString	Returns a string representation of the queue

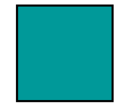
Queue Implementation Using a Linked List

- Internally, the queue is represented as a ***linked list of nodes,*** with each node containing a data element
- We need *two* pointers for the linked list
  - A pointer to the beginning of the linked list (***front*** of queue)
  - A pointer to the end of the linked list (***rear*** of queue)
- We will also have a ***count*** of the number of items in the queue

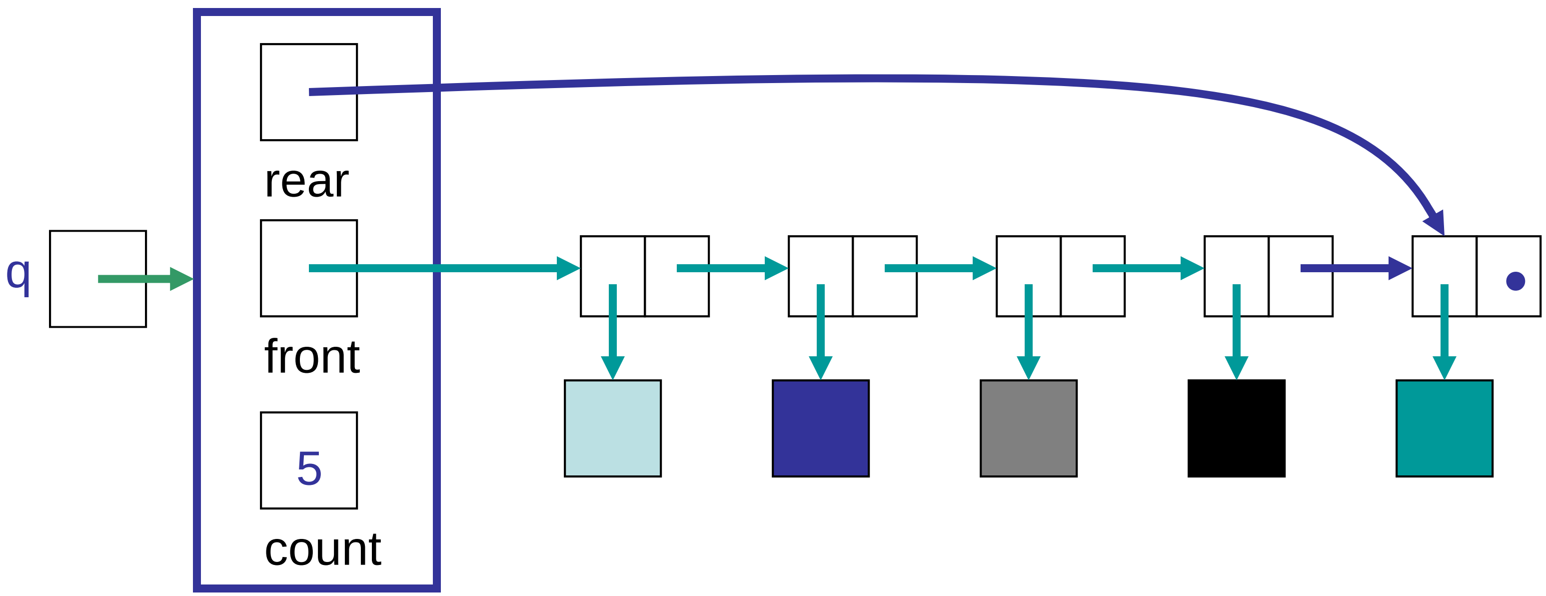
Linked Implementation of a Queue

- A queue q containing four elements

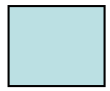


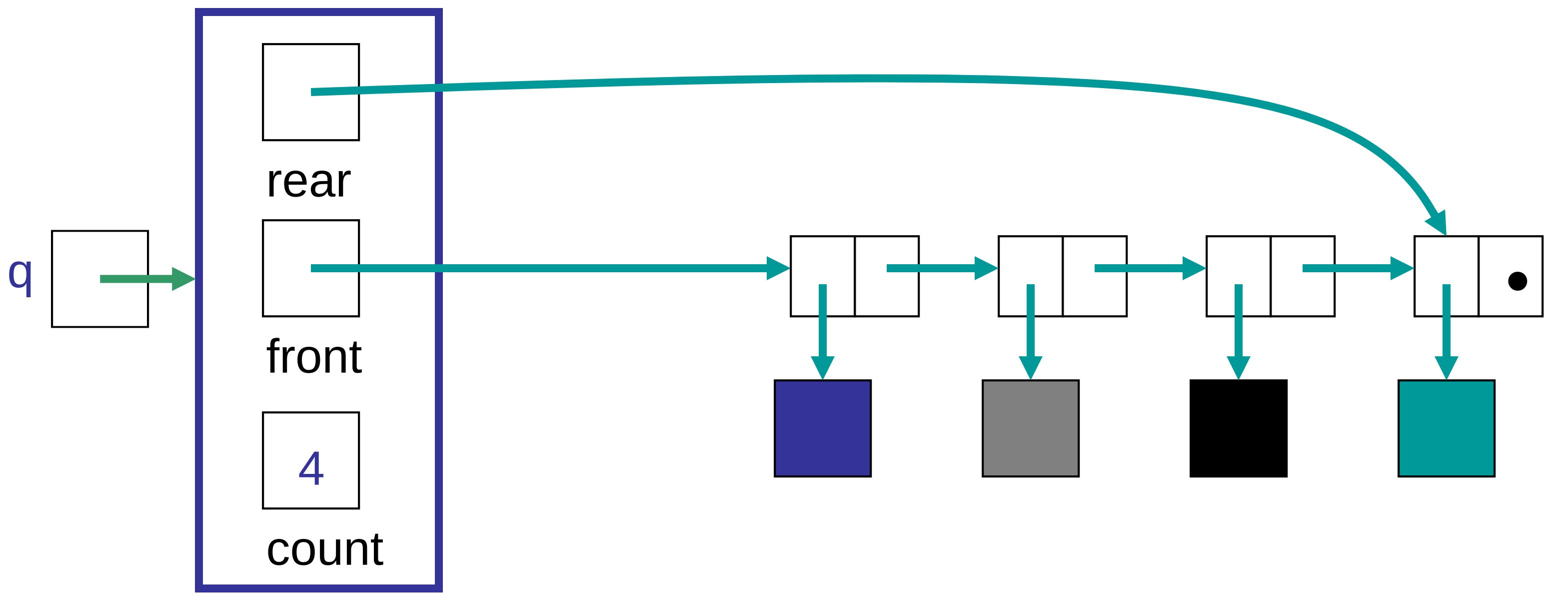
Queue After Adding Element 

New element is added in a node at the end of the list, rear points to the new node, and count is incremented



Queue After a dequeue Operation

Node containing  is removed from the front of the list (see previous slide), front now points to the node that was formerly second, and count has been decremented.

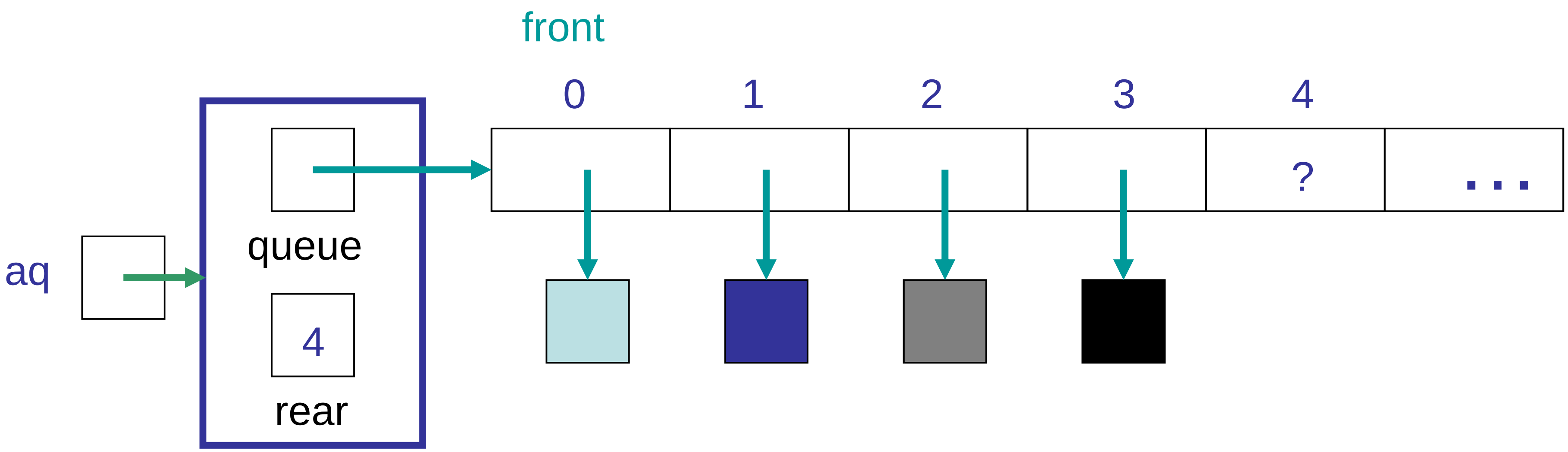


Array Implementation of a Queue

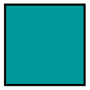
- ***First Approach***:
  - Use an array in which index 0 represents one end of the queue (the *front*)
  - Integer value ***rear*** represents the next open slot in the array (and also the number of elements currently in the queue)

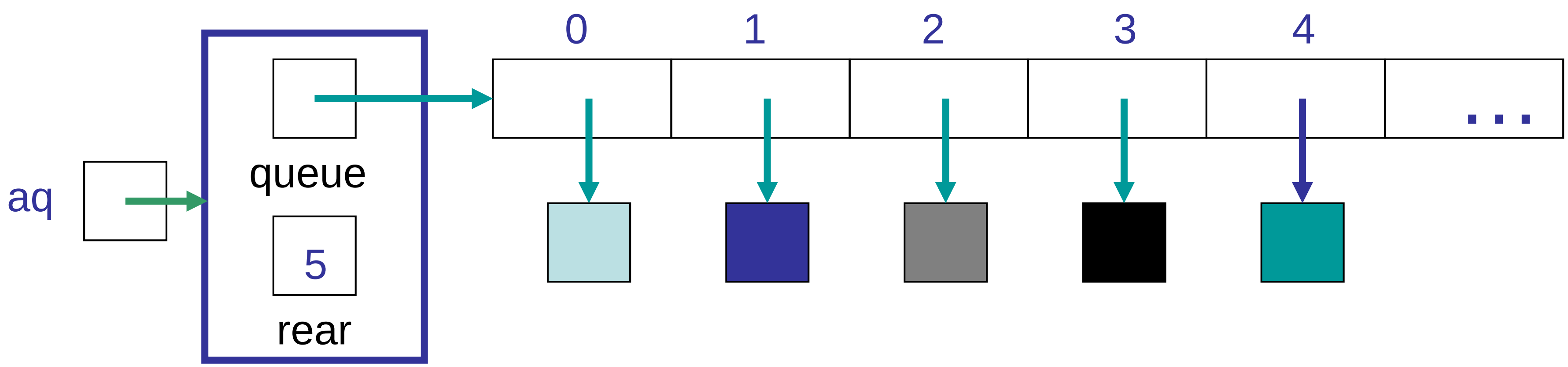
An Array Implementation of a Queue

A queue aq containing four elements



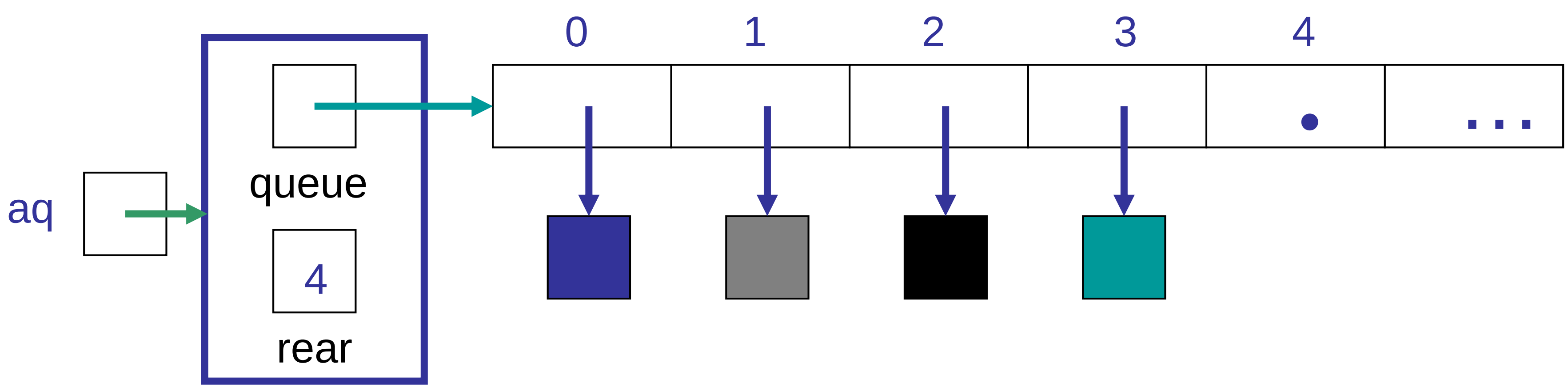
Queue After Adding an Element

Element  is added at the array location given by the (old) value of rear, and then rear is incremented.



Queue After Removing an Element

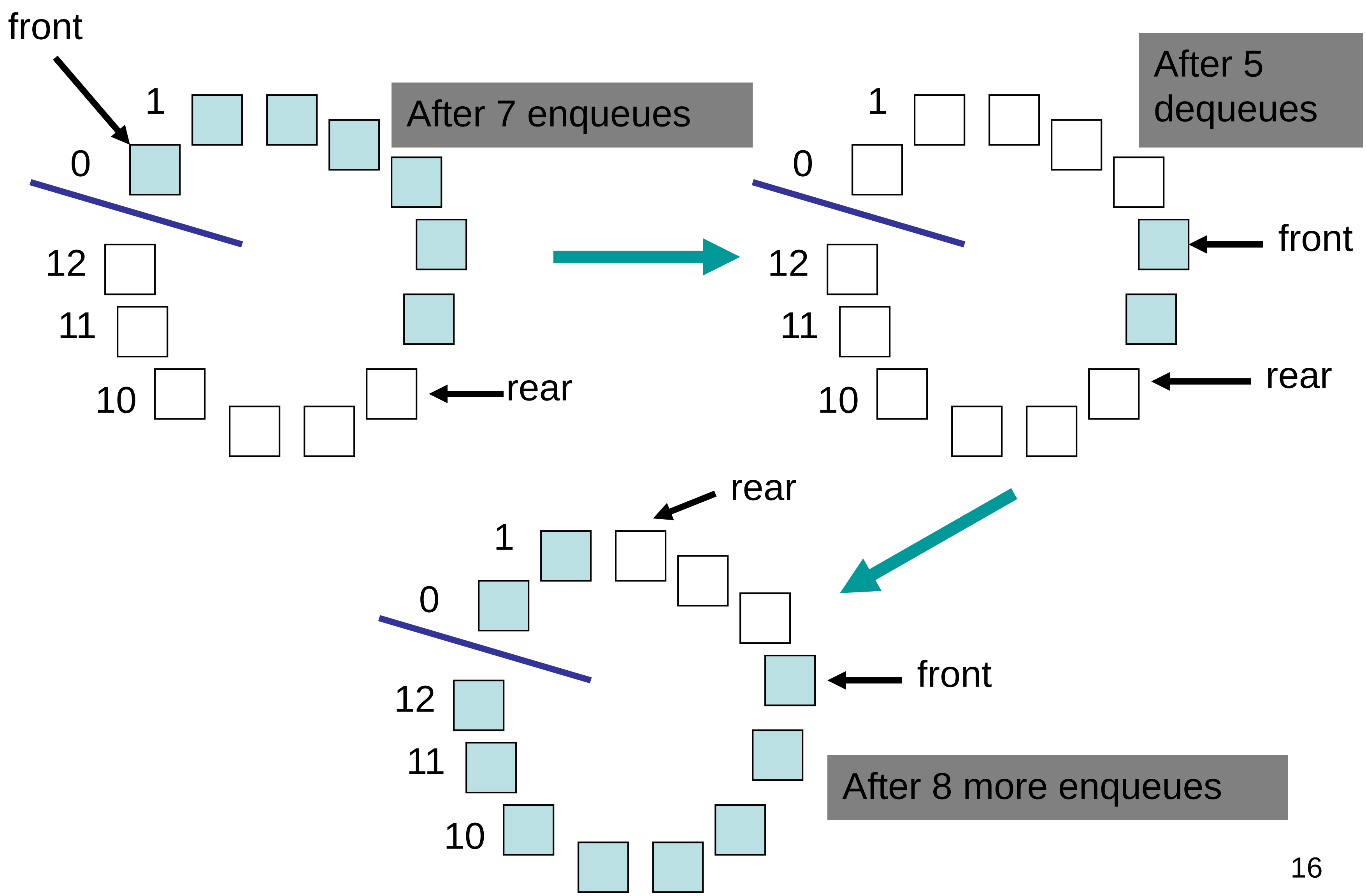
Element is removed from array location 0, remaining elements are shifted forward one position in the array, and then rear is decremented.



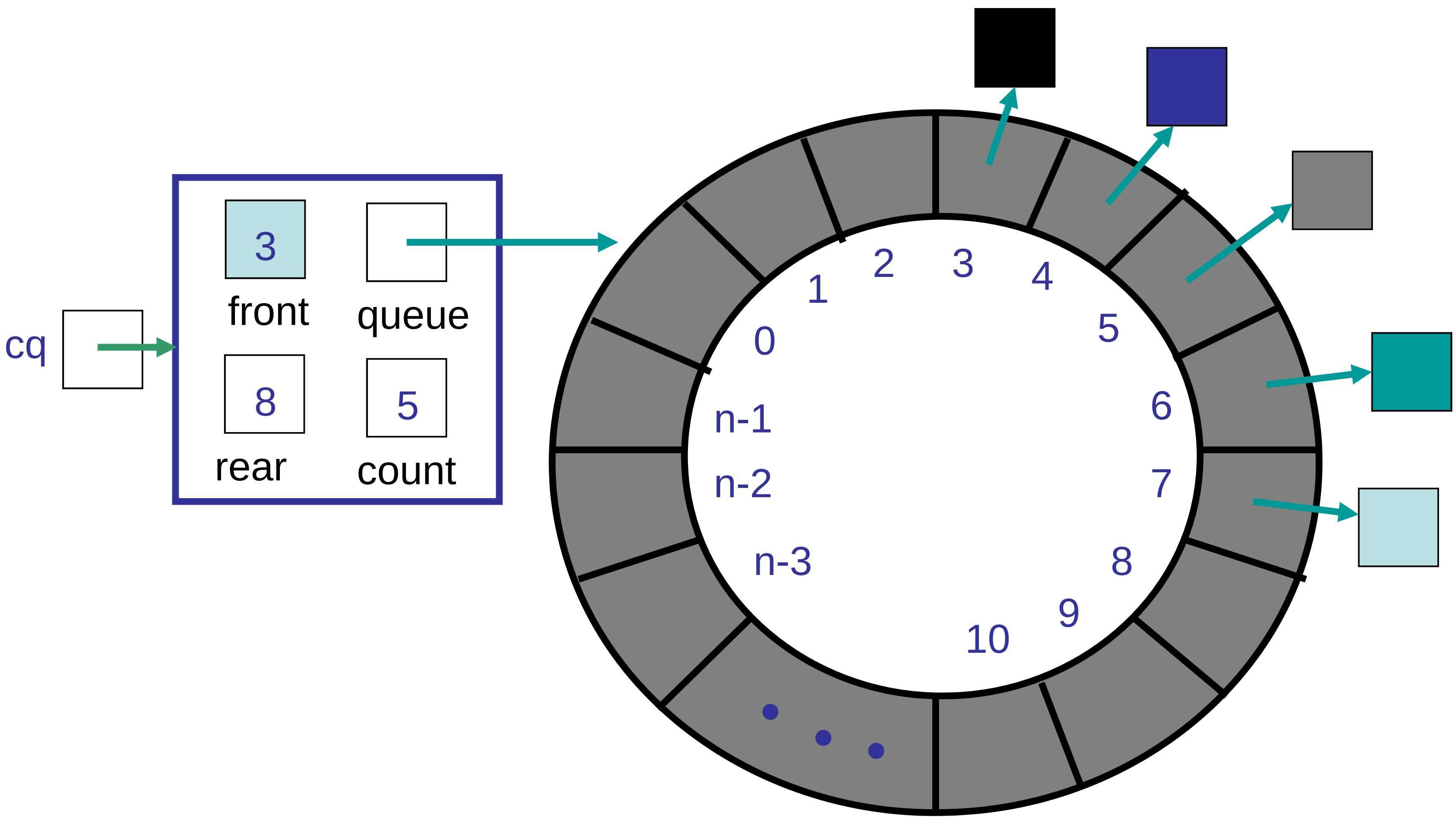
Second Approach: Queue as a *Circular Array*

- If we don't fix one end of the queue at index 0, we won't have to shift elements
- ***Circular array*** is an array that conceptually loops around on itself
  - The last index is thought to "***precede***" index 0
  - In an array whose last index is **n**, the location "***before***" index **0** is index **n**; the location "***after***" index **n** is index **0**
- Need to keep track of where the ***front*** as well as the ***rear*** of the queue are at any given time

Conceptual Example of a Circular Queue

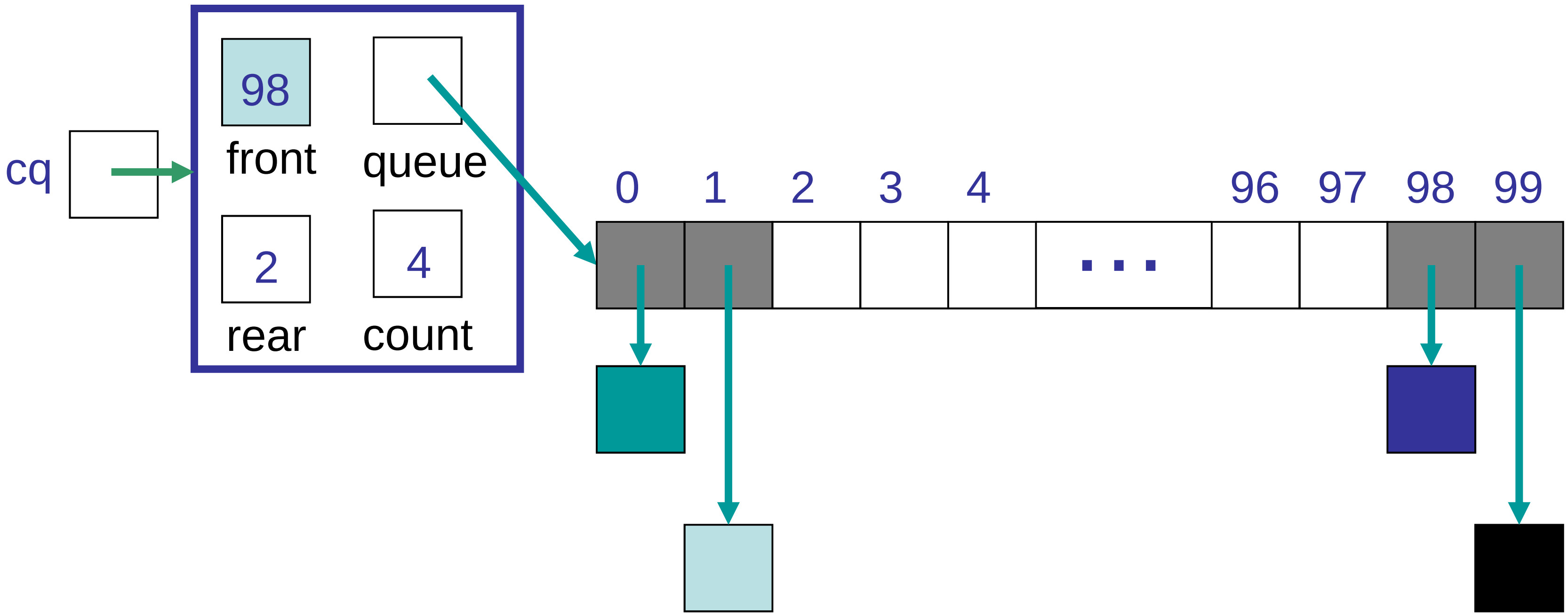


Circular Array Implementation of a Queue



Circular Queue Drawn Linearly

(Queue from previous slide)



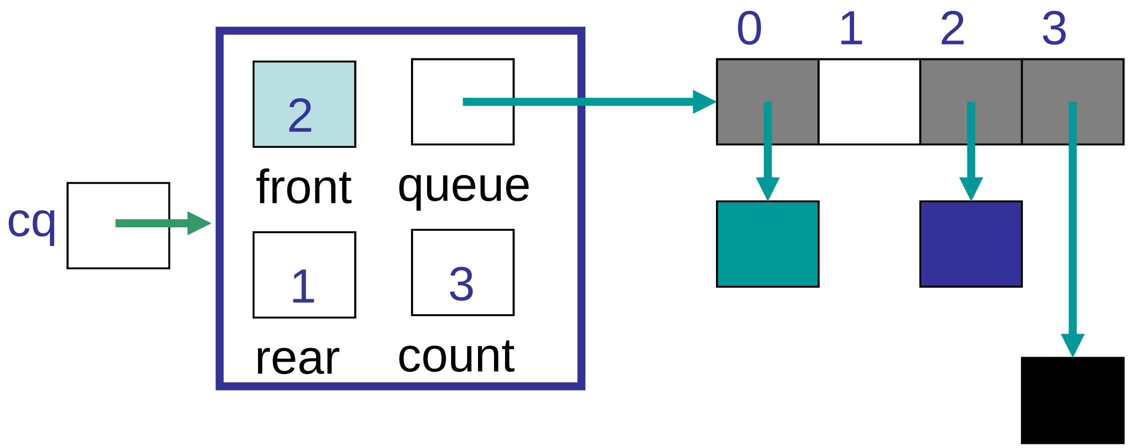
Circular Array Implementation

- When an element is enqueued, the value of **rear** is incremented
- But it must take into account the need to loop back to index 0:

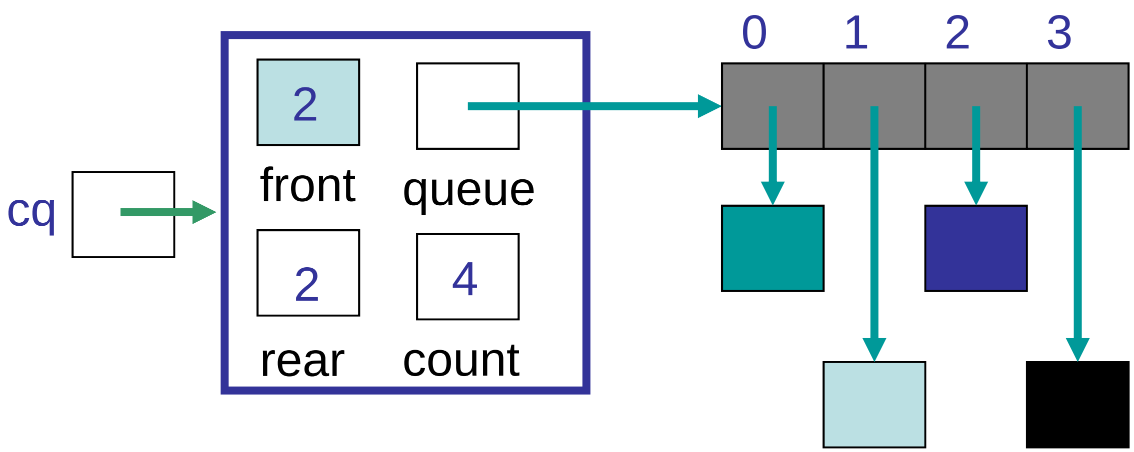
rear = (rear+1) % queue.length;

- Can this array implementation also reach capacity?

Example: array of length 4
What happens?

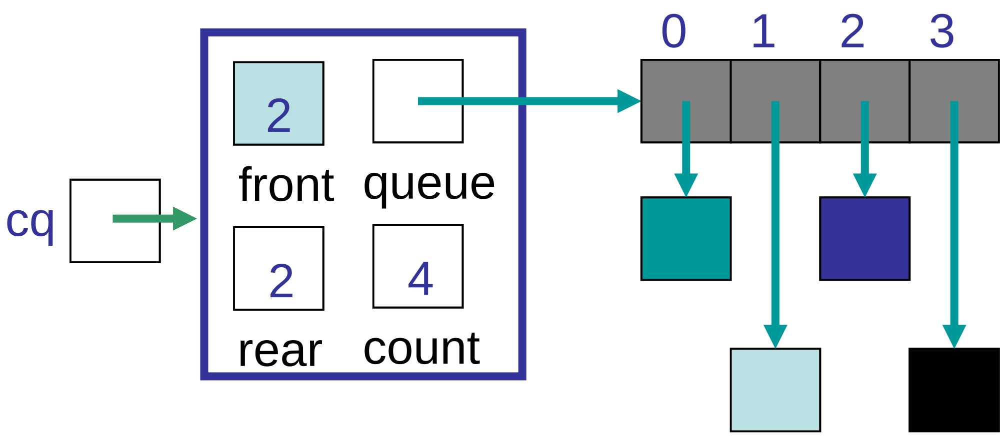


Suppose we try to add one more item to a queue implemented by an array of length 4

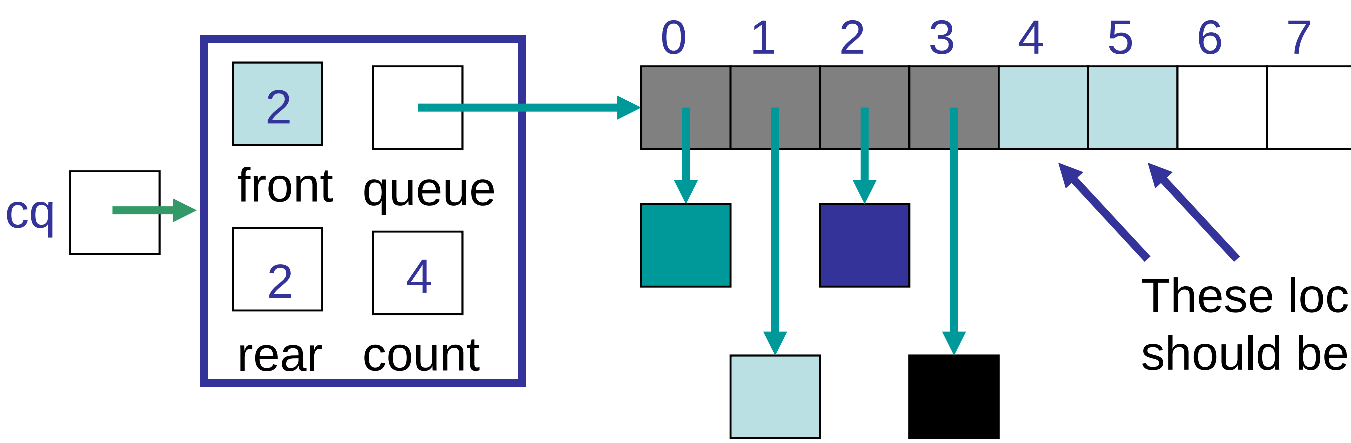


The queue is now full.
How can you tell?

Add another item.
Need to expand capacity…

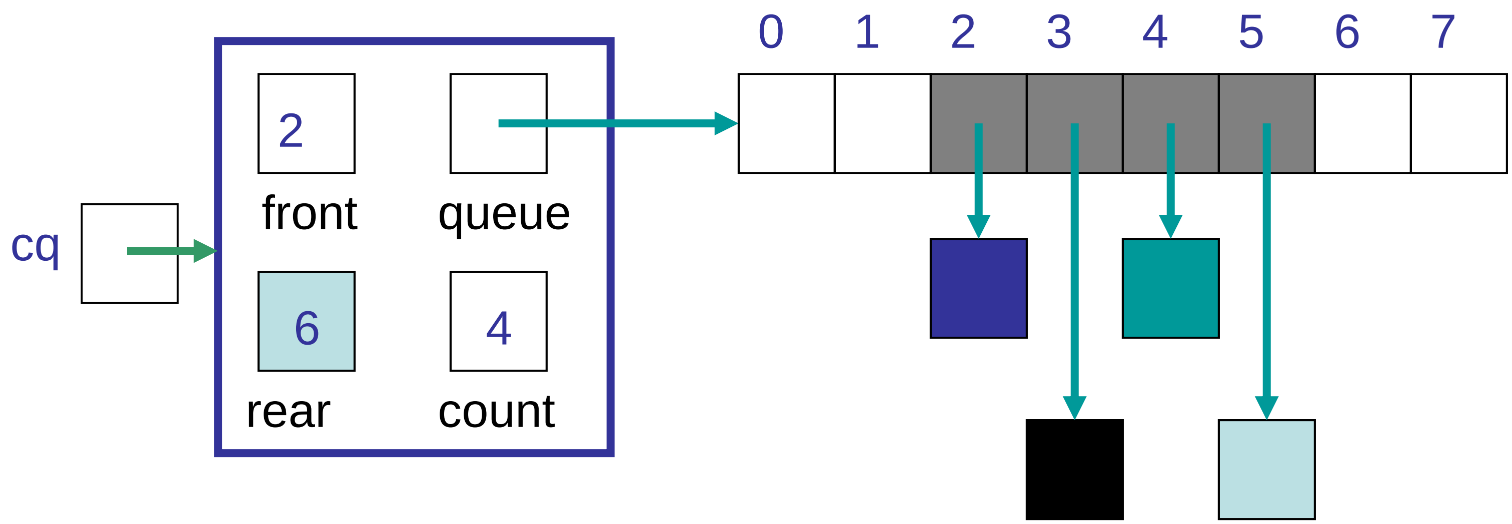


We can't just double the size of the array: circular properties of the queue will be lost

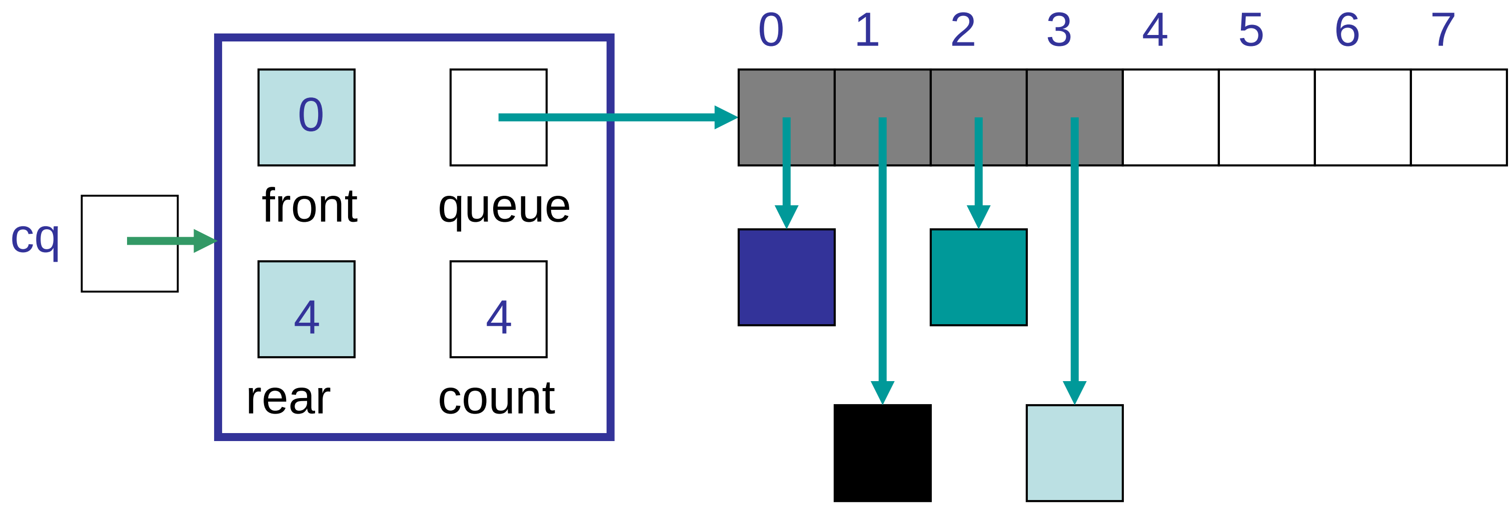


These locations should be in use

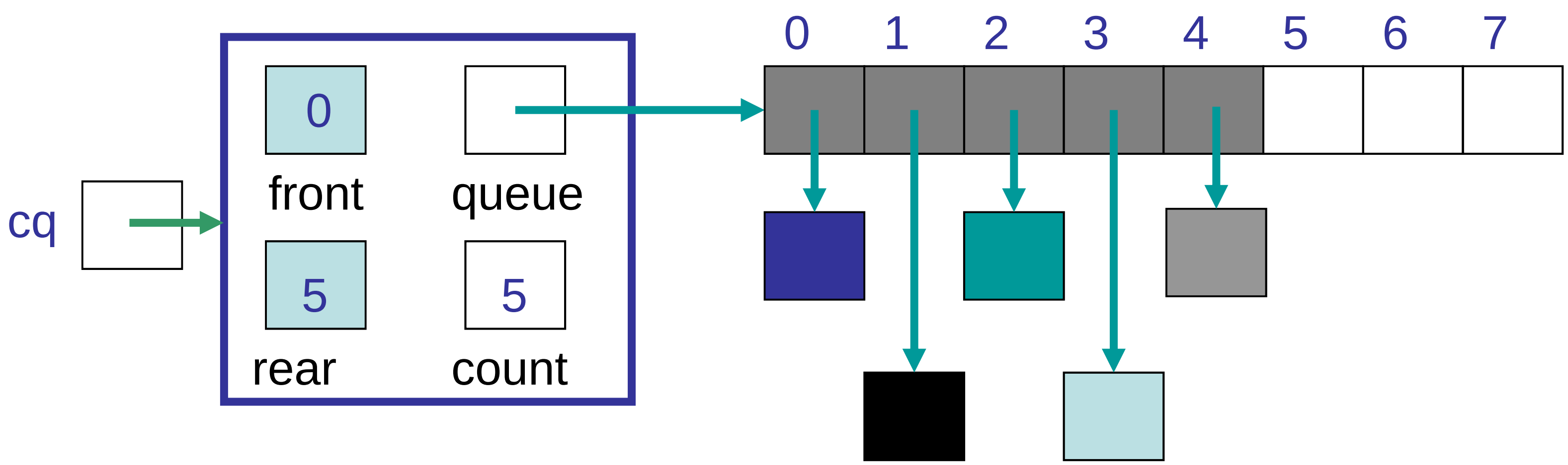
We *could* build the new array, and copy the queue elements into contiguous locations beginning at location front:



Better: copy the queue elements in order to the *beginning* of the new array



New element is added at rear = (rear+1) % queue.length



Analysis of Queue Operations

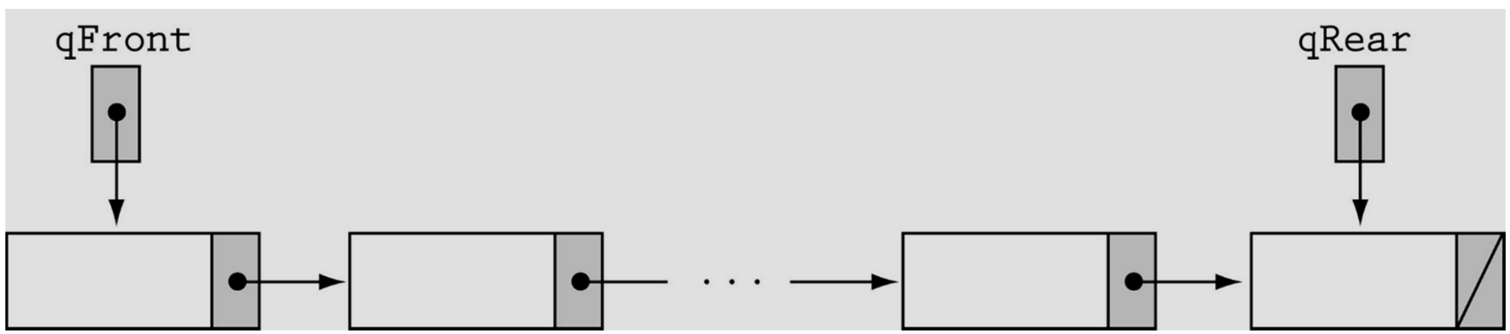
- The linked implementation of a queue does not suffer because of the need to operate on both ends of the queue
- **enqueue** operation:
  - **O(1)** for linked implementation
  - **O(n)** for circular array implementation if need to expand capacity, **O(1)** otherwise

Analysis of Queue Operations

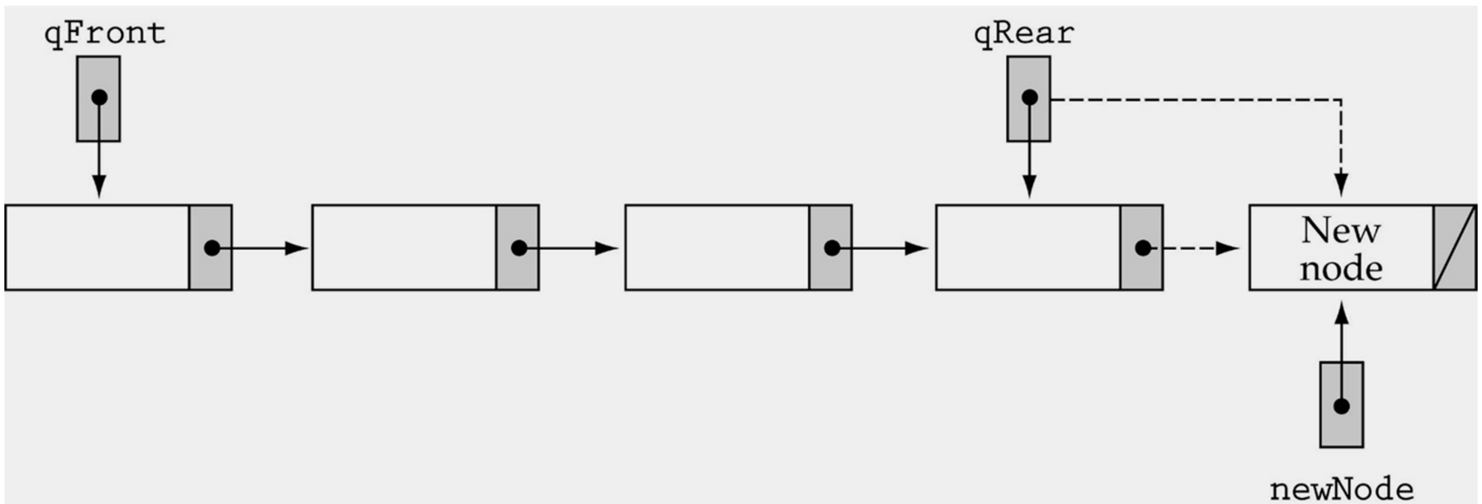
- **dequeue** operation:
  - **O(1)** for linked implementation
  - **O(1)** for circular array implementation

Queue using a linked list

- Allocate memory for each new element dynamically
- Link the queue elements together
- Use two pointers, *qFront* and *qRear*, to mark the front and rear of the queue

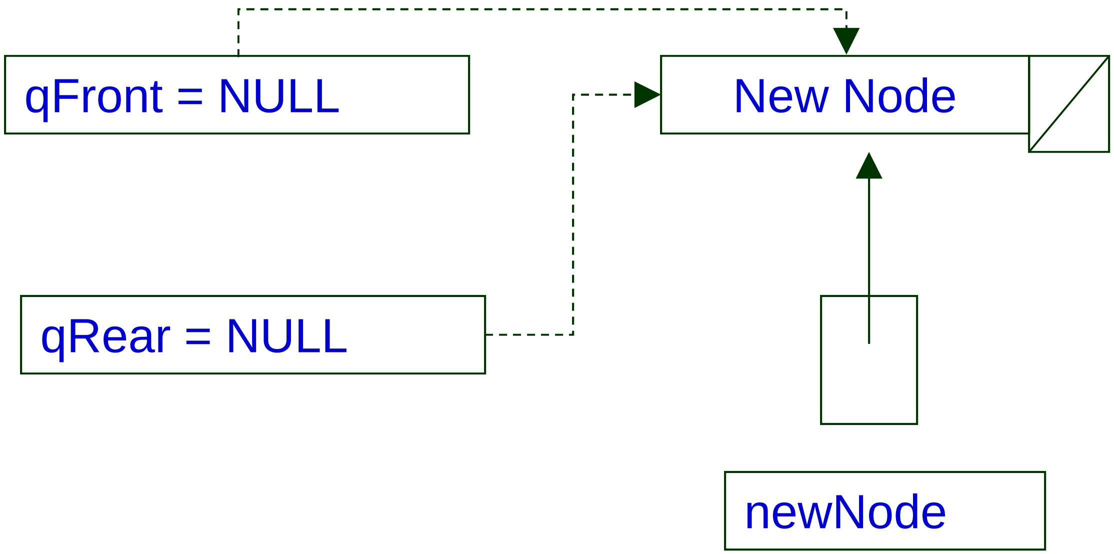


Enqueuing (non-empty queue)

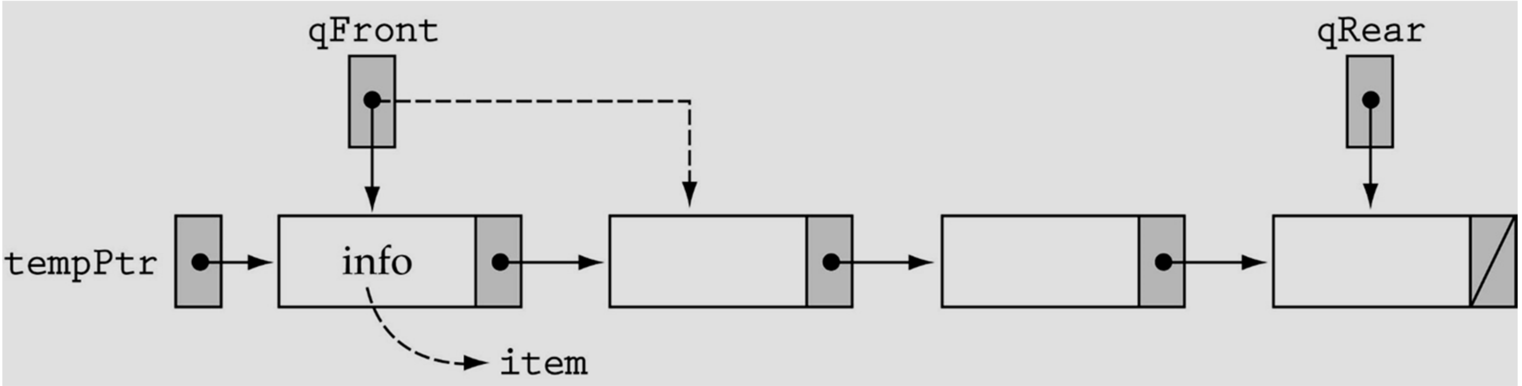


Enqueuing (empty queue)

Also need to make *qFront* point to the new node

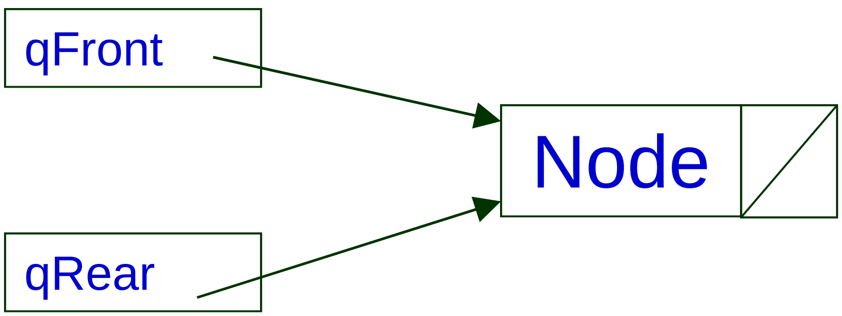


Dequeueing (the queue contains more than one element)



Dequeueing (the queue contains only one element)

Need to reset *qRear* to NULL

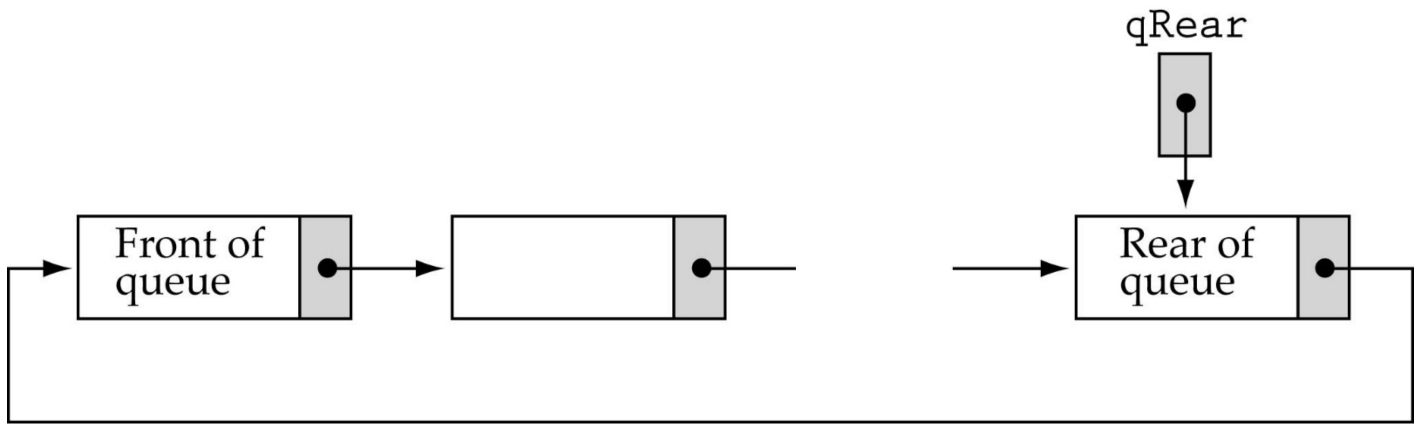


After dequeue:

qFront = NULL

qRear = NULL

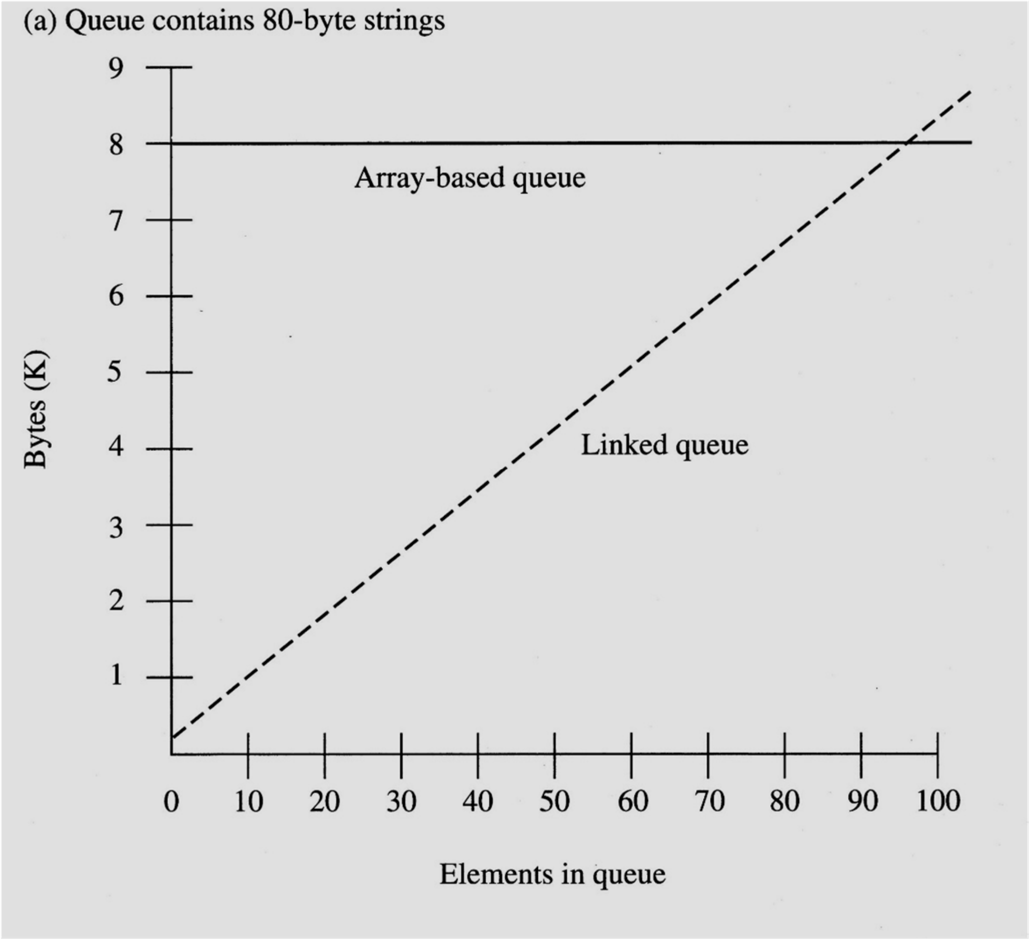
A circular linked queue design



Comparing queue implementations

- Memory requirements
  - Array-based implementation
    - Assume a queue (size: 100) of strings (80 bytes each)
    - Assume indices take 2 bytes
    - Total memory: (80 bytes x 101 slots) + (2 bytes x 2 indexes) = 8084 bytes
  - Linked-list-based implementation
    - Assume pointers take 4 bytes
    - Total memory per node: 80 bytes + 4 bytes = 84 bytes

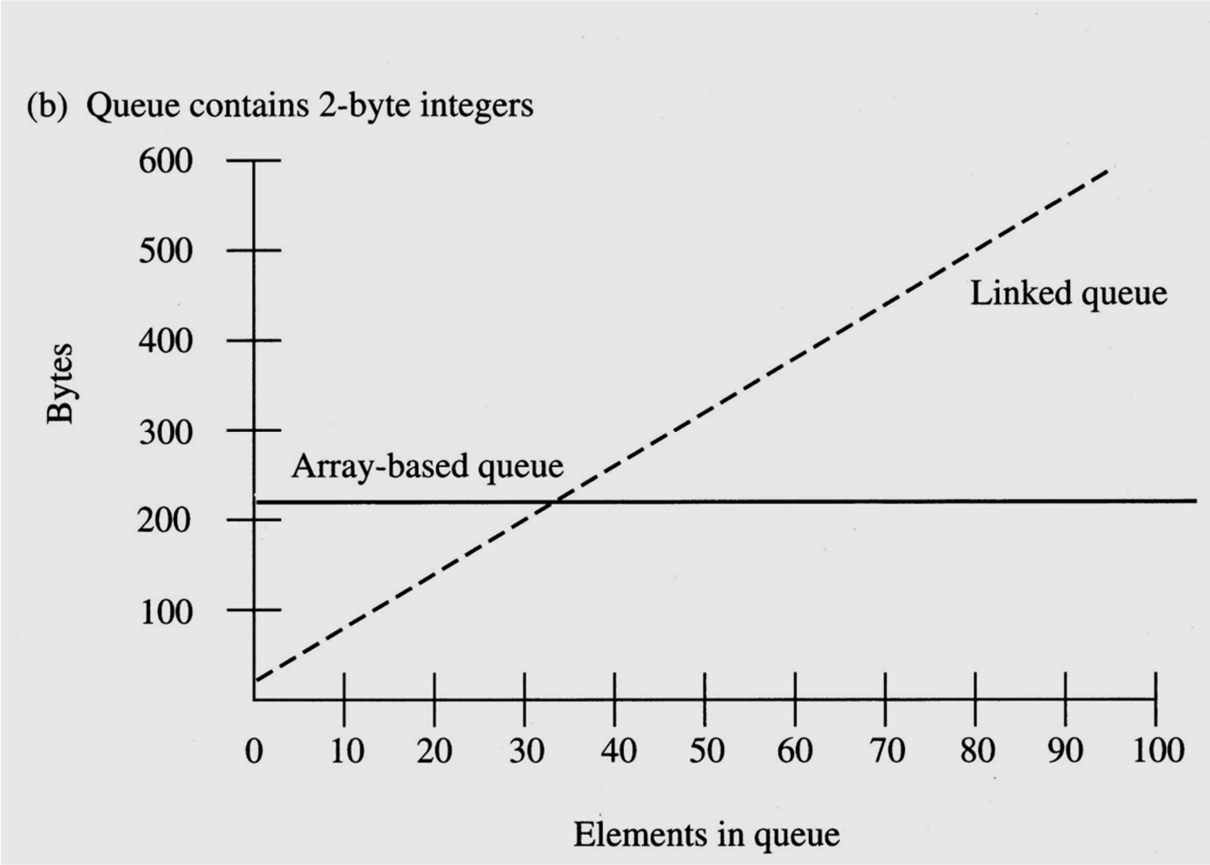
Comparing queue implementation



Comparing queue implementations

- Memory requirements
  - <u>Array-based implementation</u>
    - Assume a queue (size: 100) of short integers (2 bytes each)
    - Assume indices take 2 bytes
    - Total memory: (2 bytes x 101 slots) + (2 bytes x 2 indexes) = 206 bytes
  - <u>Linked-list-based implementation</u>
    - Assume pointers take 4 bytes
    - Total memory per node: 2 bytes + 4 bytes = 6 bytes

Comparing queue implementations



Comparing queue implementations

Big-O Comparison of Queue Operations

Operation	Array Implementation	Linked Implementation
Class constructor	O(1)	O(1)
MakeEmpty	O(1)	O(N)
IsFull	O(1)	O(1)
IsEmpty	O(1)	O(1)
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Destructor	O(1)	O(N)